

Modelos Generativos Profundos



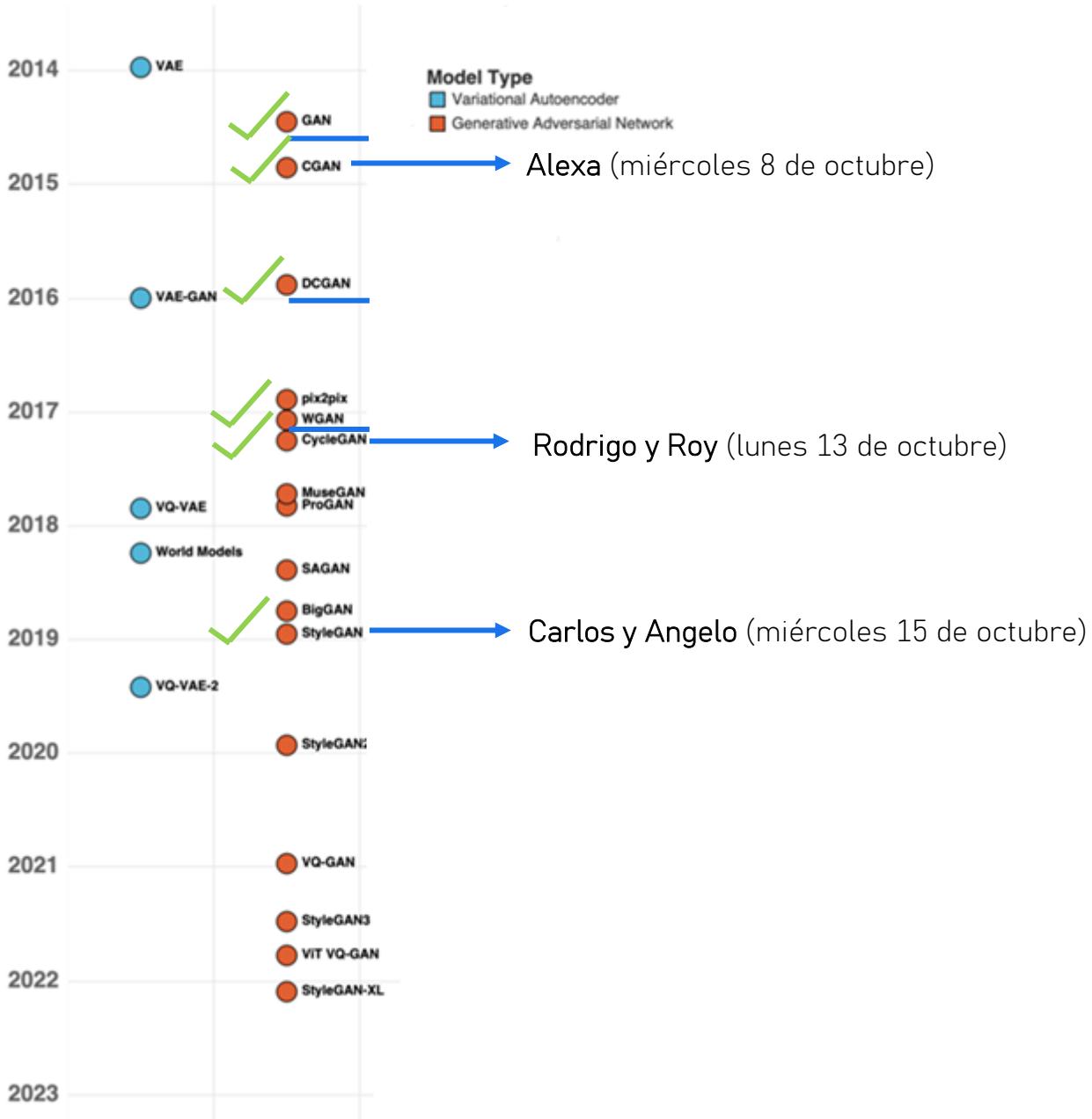
GANs V

Clase 16

Dra. Wendy Aguilar

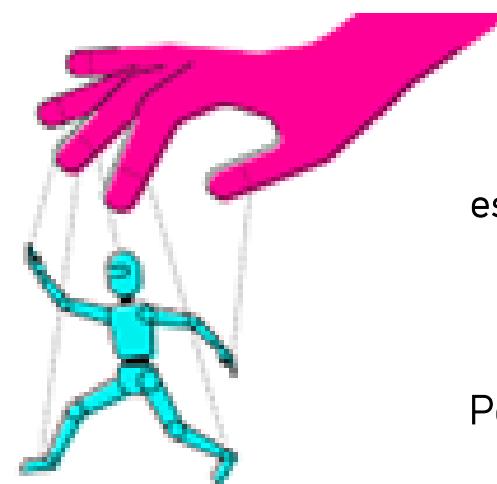
UN ENFOQUE DESDE LA
CREATIVIDAD
COMPUTACIONAL

Generative AI Timeline



Estrategias de control generativo en GANs

- Buscan que el modelo deje de producir **muestras al azar**
- Podamos dirigir el proceso de **generación** hacia resultados específicos, interpretables o guiados por intención.



estrategias_control_stylegan.ipynb

Google
colab

Poner a ejecutar la celda 1, 2 y 3

Estrategias de control generativo en GANs

4 Niveles conceptuales del control generativo

1. Control Condicional

Se guía la generación con información externa (p.ej. etiquetas).

Estrategia	Principio de control	Ejemplo representativo
Condisional explícito	Se incorporan etiquetas o variables semánticas (como clase o categoría) en el generador y discriminador para guiar la generación.	cGAN 
Control semántico guiado por otra modalidad	Se usa otra modalidad como condición de entrada (texto, layout, atributos) para dirigir la generación sin fusión completa entre modalidades.	AttnGAN, CLIP-GAN, Layout2Image 

Estrategias de control generativo en GANs

4 Niveles conceptuales del control generativo

1. Control Condicional

Se guía la generación con información externa (clase, texto, atributos, estilo).

2. Control Latente

Se manipula el espacio latente para modificar o combinar características internas.

Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación y mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Estrategias de control generativo en GANs

4 Niveles conceptuales del control generativo

1. Control Condicional

Se guía la generación con información externa (clase, texto, atributos, estilo).

2. Control Latente

Se manipula el espacio latente para modificar o combinar características internas.

3. Multimodal

Se integran distintas modalidades de información —como texto, bocetos o imágenes de referencia— para guiar y dirigir el proceso generativo hacia resultados coherentes con cada tipo de entrada.

Estrategia	Principio de control	Ejemplo representativo	
Integración entre modalidades	Se combinan distintas fuentes de información (texto, imagen, layout, audio o atributos) para guiar la generación en múltiples dominios.	CLIP-GAN, TediGAN, StyleCLIP	
Traducción y reconstrucción cruzada	El modelo aprende representaciones compartidas entre modalidades, permitiendo traducir o reconstruir una a partir de otra.	CycleGAN, MUNIT, DRIT++	

Estrategias de control generativo en GANs

4 Niveles conceptuales del control generativo

4. Control mediante Retroalimentación u Objetivos Externos en GANs

Introduce una **capa de control superior**, donde el generador **aprende a optimizar respecto a una meta externa**, no solo respecto a los datos o condiciones de entrada.

Estrategia	Principio de control	Ejemplo representativo
Pérdidas perceptuales	Se incorporan métricas perceptuales o semánticas (por ejemplo, basadas en VGG o CLIP) para ajustar la generación hacia mayor realismo o coherencia.	Perceptual GAN, SRGAN 
Recompensas o criterios externos	Se utiliza retroalimentación basada en recompensas (automáticas o humanas) para optimizar propiedades como estética, novedad o creatividad.	RL-GAN, Creative Adversarial Network (CAN), Human-in-the-loop GANs 

estrategias_control_stylegan.ipynb



2. Control Latente

Se manipula el espacio latente para modificar o combinar características internas.

Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación y mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

StyleGAN

2019

estrategias_control_stylegan.ipynb



- **Control Latente**
Se manipula el espacio latente para modificar o combinar características internas.

Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación y mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Meta: permitir *edición semántica controlada* de las imágenes generadas.

- **Control explícito**
 - Cambiar el tono cálido/frío de una foto sin alterar la forma de la cara.
 - Ajustar la intensidad del maquillaje sin modificar los rasgos estructurales.
- **Control jerárquico**
 - El efecto del estilo depende de la profundidad de la capa en la red generadora:
 - Las **capas tempranas** controlan atributos **globales** o **estructurales**.
 - Las **capas intermedias** controlan rasgos **faciales** o **morfológicos**.
 - las **capas finales** controlan **detalles locales** como texturas o ruido fino.
- **Control continuo**
 - Tomar dos vectores de estilo w_1 y w_2 (por ejemplo, uno de una persona joven y otro de una persona mayor).
 - Al interpolar $w = (1 - \alpha)w_1 + \alpha w_2$, con $0 \leq \alpha \leq 1$, el modelo genera un cambio gradual y suave de edad en el mismo rostro.

Conditional GANs (cGANs).

Condicionan la generación mediante **etiquetas externas** (por ejemplo, "rubio", "feliz", "invierno") y por tanto ofrecen un **control discreto y supervisado**.

Controla *qué* se genera (según una etiqueta).

VS

StyleGAN

Logra un **control no supervisado y estructuralmente localizado**: cada capa del generador recibe información de estilo (mediante *Adaptive Instance Normalization*) que regula distintos aspectos visuales del resultado.

Controla *cómo* se genera (según el estilo interno del espacio latente).

StyleGAN

Arquitectura del Generador

1. Espacio latente original:

vector $z \sim \mathcal{N}(0, I)$.

2. Mapping Network (8 capas densas):

Cada una con 512 neuronas

$$z \in \mathbb{R}^{512} \xrightarrow[\text{8 capas densas}]{\text{Mapping Network}} w \in \mathbb{R}^{512}$$

Transforma z en un vector de estilo w en un nuevo espacio latente desacoplado \mathcal{W} .

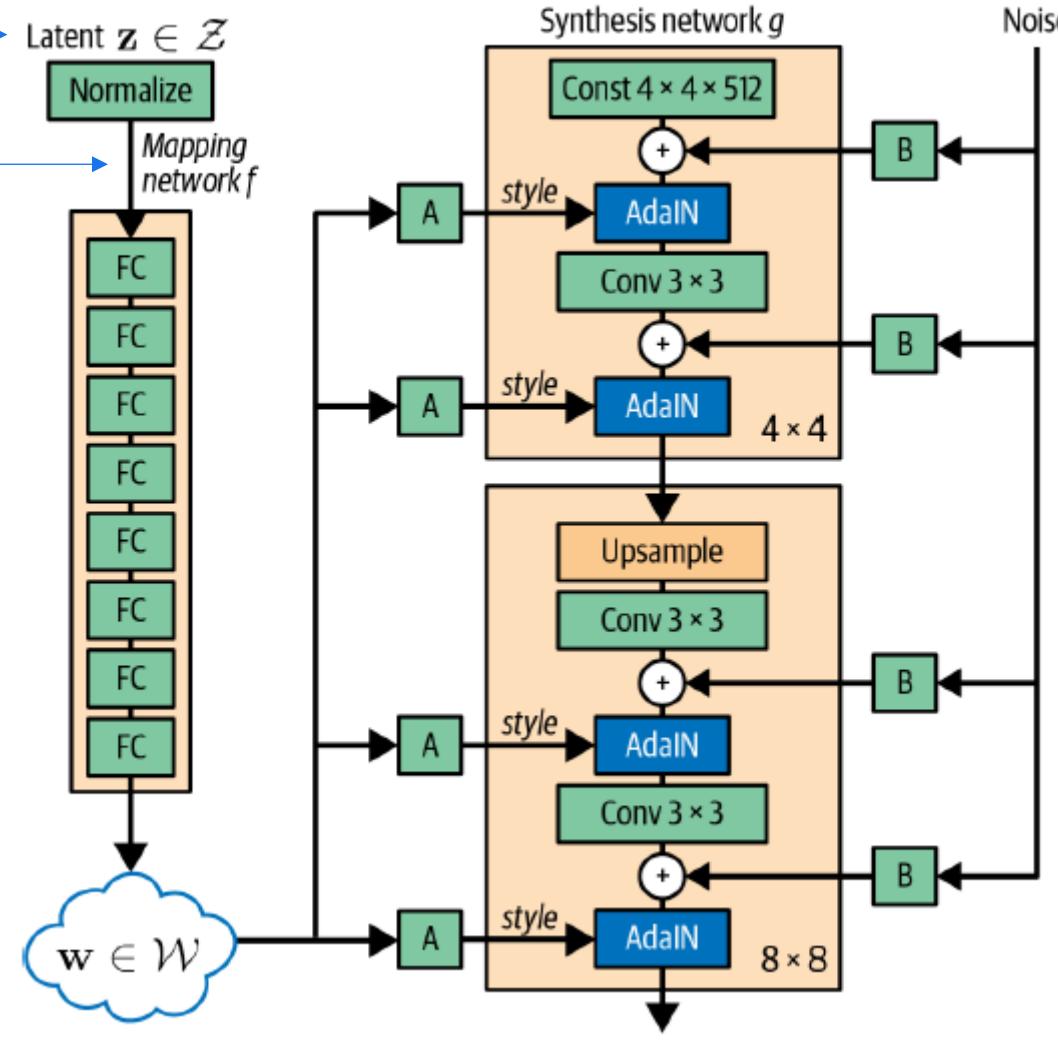
De modo que distintos atributos visuales (como edad, pose, iluminación, etc.) correspondan a **direcciones separadas** en el espacio \mathcal{W} .

Dirección horizontal cambios en la edad:
mover a la derecha → más joven,
a la izquierda → más viejo.

Dirección vertical cambios en la expresión facial:
arriba → sonrisa, abajo → rostro serio.

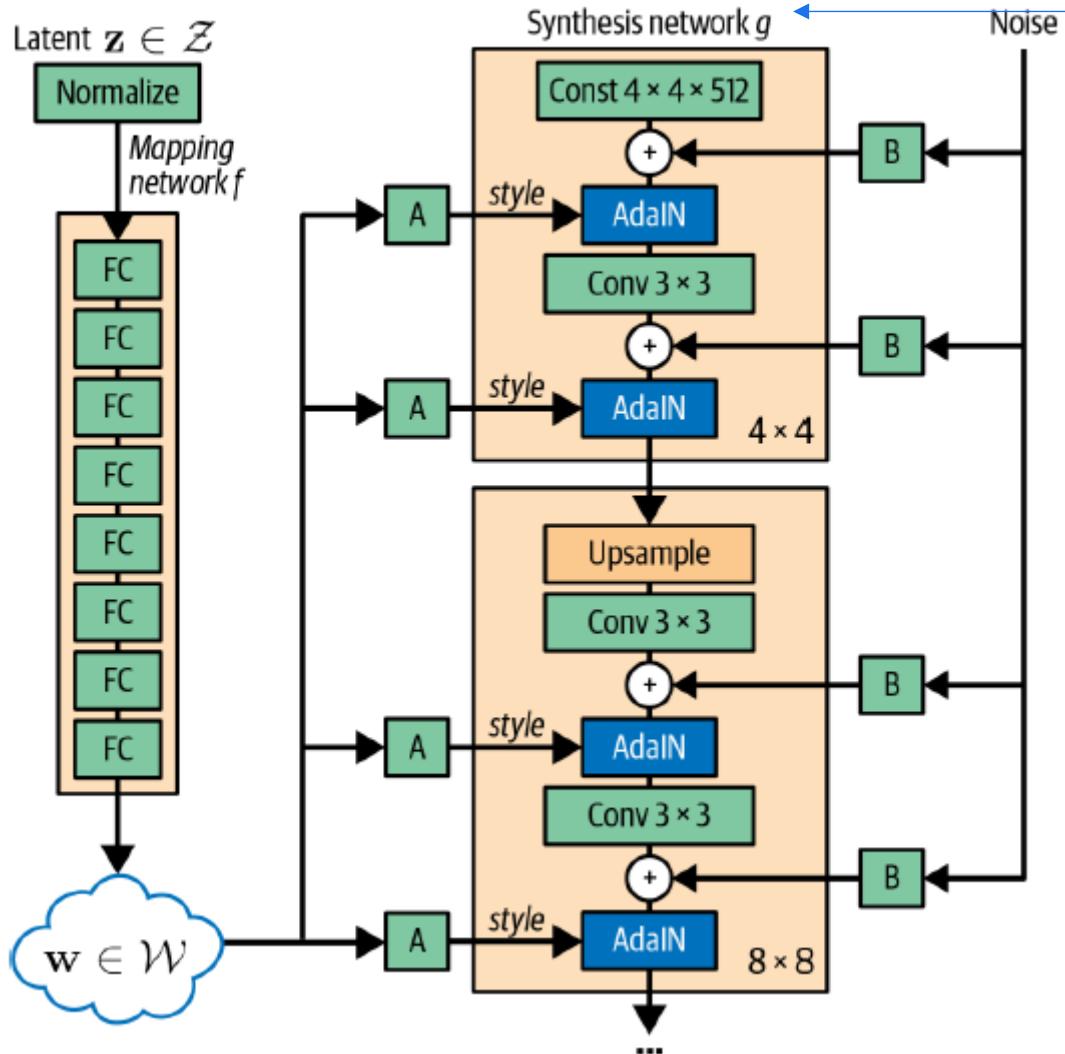
Otra dirección oblicua

alterar la iluminación o el color de cabello.



StyleGAN

Arquitectura del Generador



3. Synthesis Network

Construye la imagen a partir del vector de estilo w

- No parte directamente del vector z .
- Inicia desde un tensor aprendido (un “constant input”) — una pequeña matriz de $4 \times 4 \times 512$ parámetros entrenados.
- Luego va expandiendo la resolución capa por capa hasta llegar a la imagen final (por ejemplo, 1024×1024).

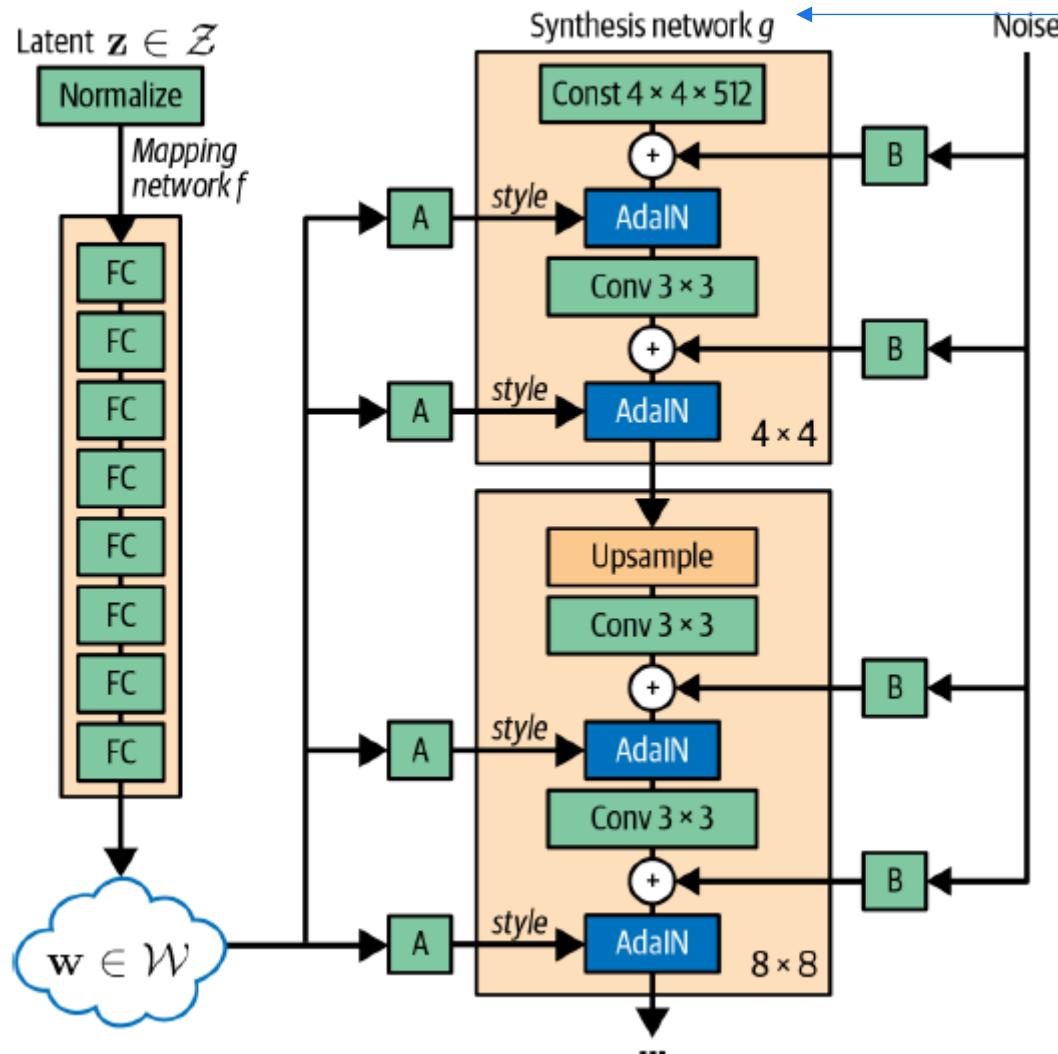
Crecimiento progresivo de la resolución

- Cada bloque de resolución incluye:
 - Dos capas convolucionales (3×3)
 - Upsampling (duplicar resolución)
 - AdaIN y ruido estocástico por capa

$4 \times 4 \rightarrow 8 \times 8 \rightarrow 16 \times 16 \rightarrow 32 \times 32 \rightarrow \dots \rightarrow 1024 \times 1024$

StyleGAN

Arquitectura del Generador



Inyección de estilo por capa (AdaIN)

- Cada capa recibe parámetros de estilo (y_s, y_b) derivados del vector \mathbf{w} mediante una transformación afín:

$$(y_s, y_b) = A_i \mathbf{w}$$

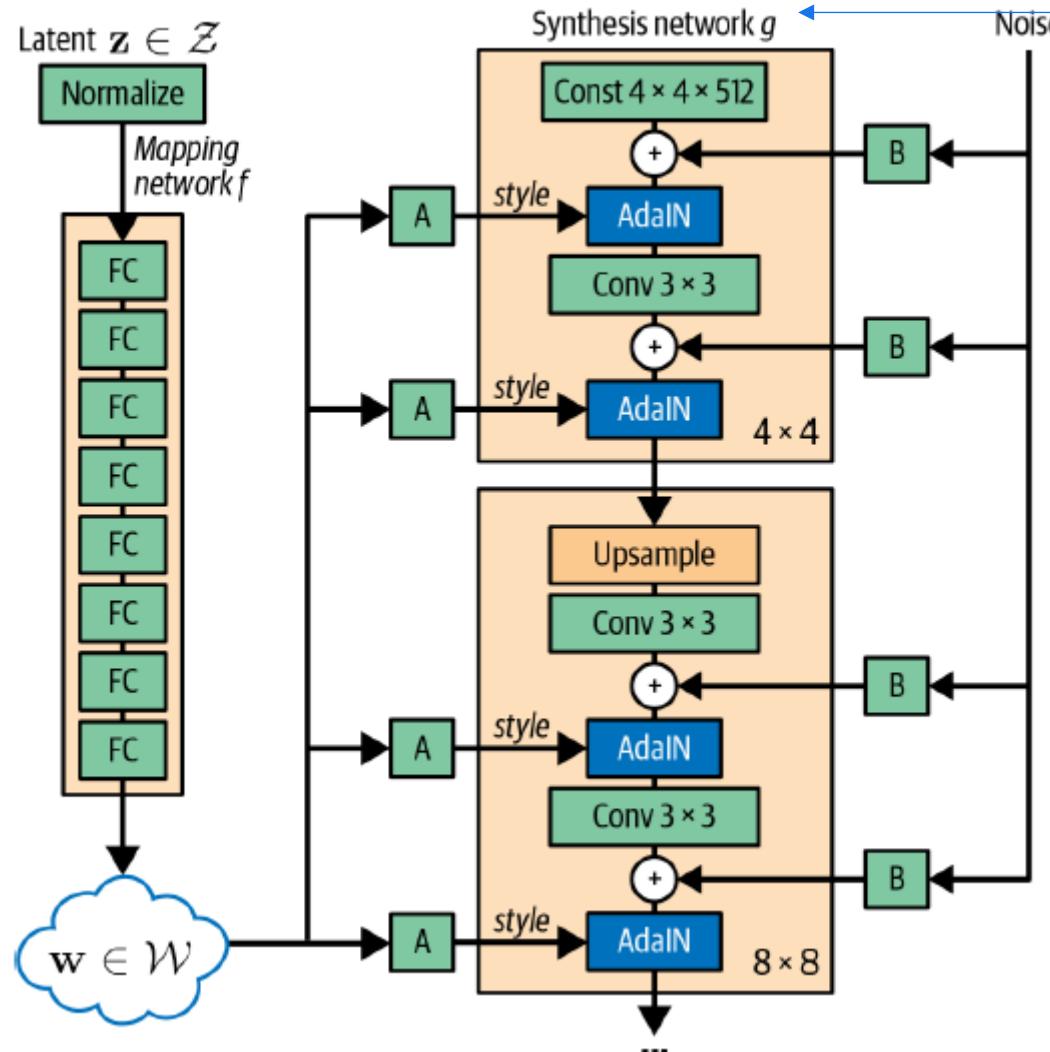
- Estos parámetros modulan las características intermedias de la capa mediante Adaptive Instance Normalization (AdaIN):

$$\text{AdaIN}(x_i, y) = y_{s,i} \frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i}$$

- Así, cada capa controla un nivel distinto del estilo (estructura, rasgos o textura).

StyleGAN

Arquitectura del Generador



Inyección de ruido

- Además del estilo global controlado por \mathbf{w} , cada capa recibe un **mapa de ruido** independiente:

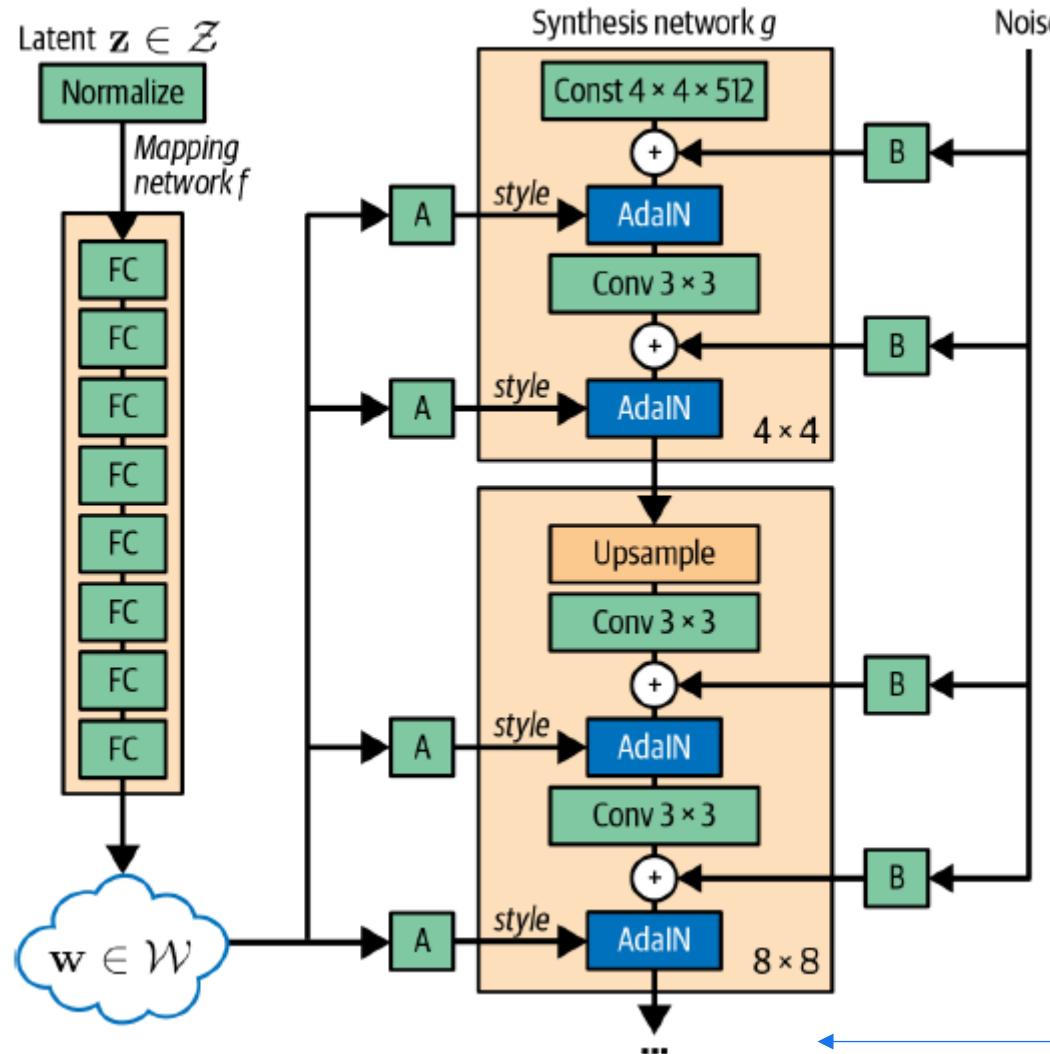
$$\mathbf{x}_i = \mathbf{x}_i + b_i \cdot \text{noise}_i$$

Este ruido introduce **variaciones locales aleatorias**, como poros, arrugas, textura del cabello o microdetalles de fondo.

- El **estilo** define el aspecto general (quién es la persona).
- El **ruido** aporta diversidad visual (pequeños detalles únicos).

StyleGAN

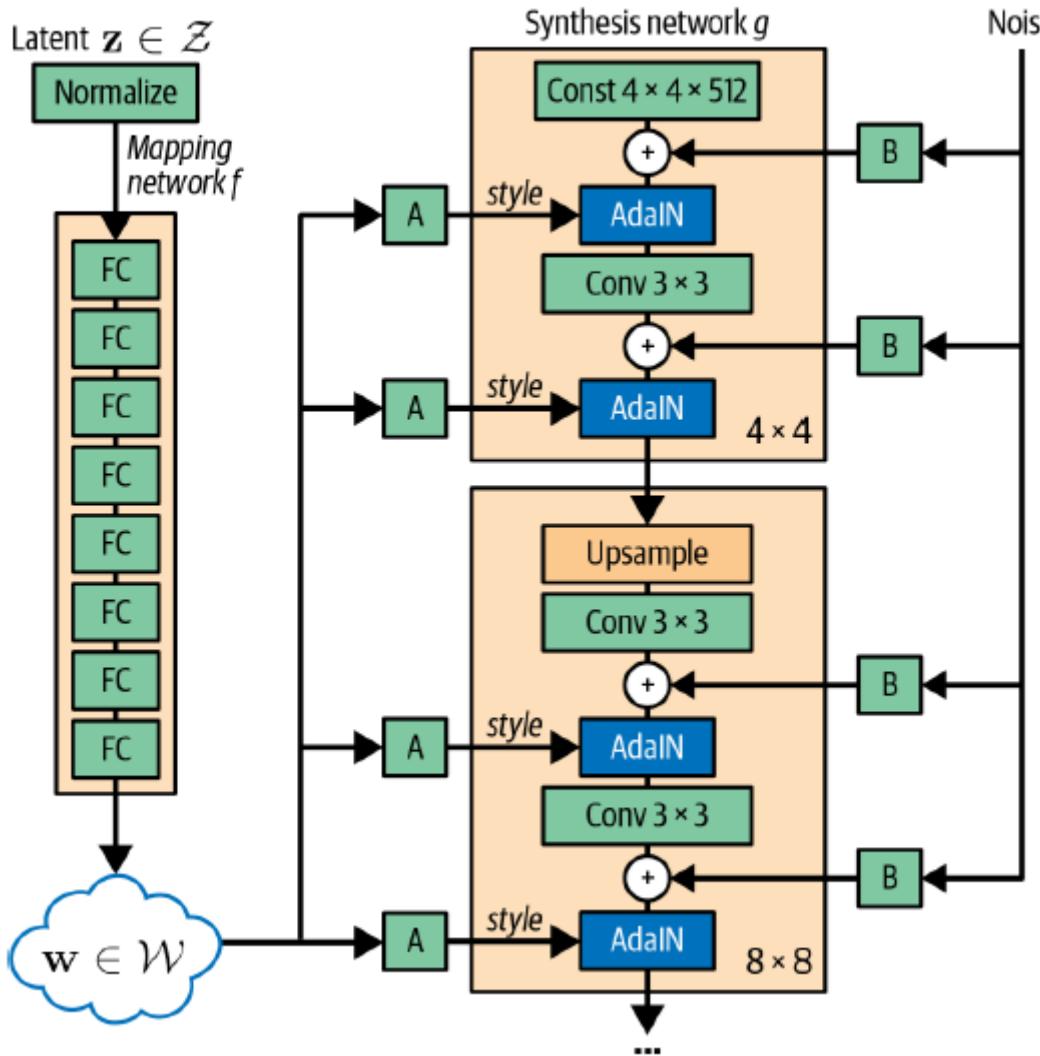
Arquitectura del Generador



La red termina con una convolución 1×1 para combinar todas las salidas RGB y producir la imagen final, usualmente de 3 canales (RGB) y rango $[-1, 1]$.

StyleGAN

Arquitectura del Discriminador



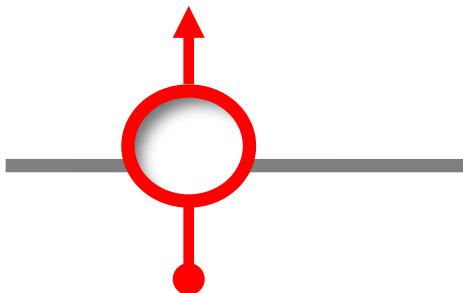
Sigue una arquitectura convolucional jerárquica que reduce progresivamente la resolución de la imagen hasta llegar a un vector escalar (probabilidad real/falsa).

Su diseño es, conceptualmente, el inverso del *Synthesis Network*.

Línea de tiempo de IA Generativa



2020



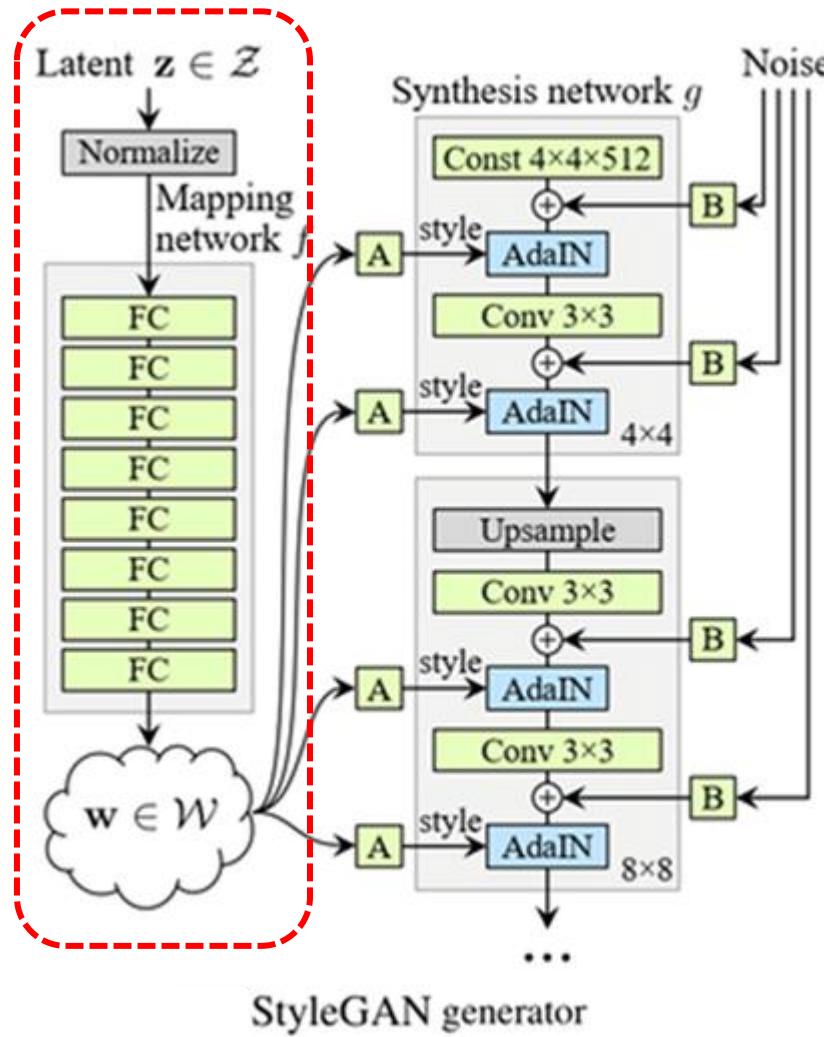
StyleGAN2

Se desarrolló para eliminar los artefactos generados por AdalN, mejorar la coherencia espacial y lograr un control jerárquico más interpretable sobre los atributos visuales.

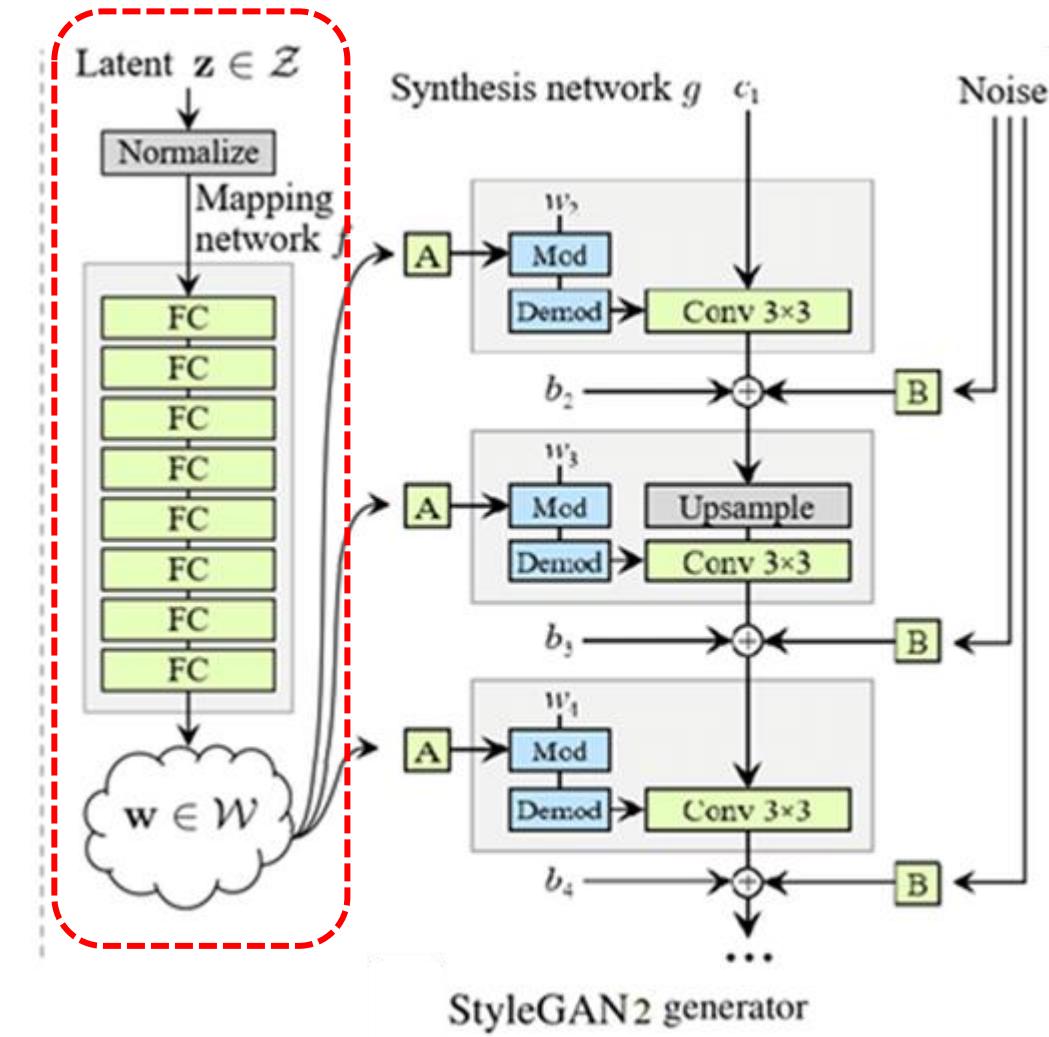


Karras, T., Laine, S., & Aila, T. (2019). A Style-Based Generator Architecture for Generative Adversarial Networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2019), 4401–4410.

<https://arxiv.org/abs/1812.04948>



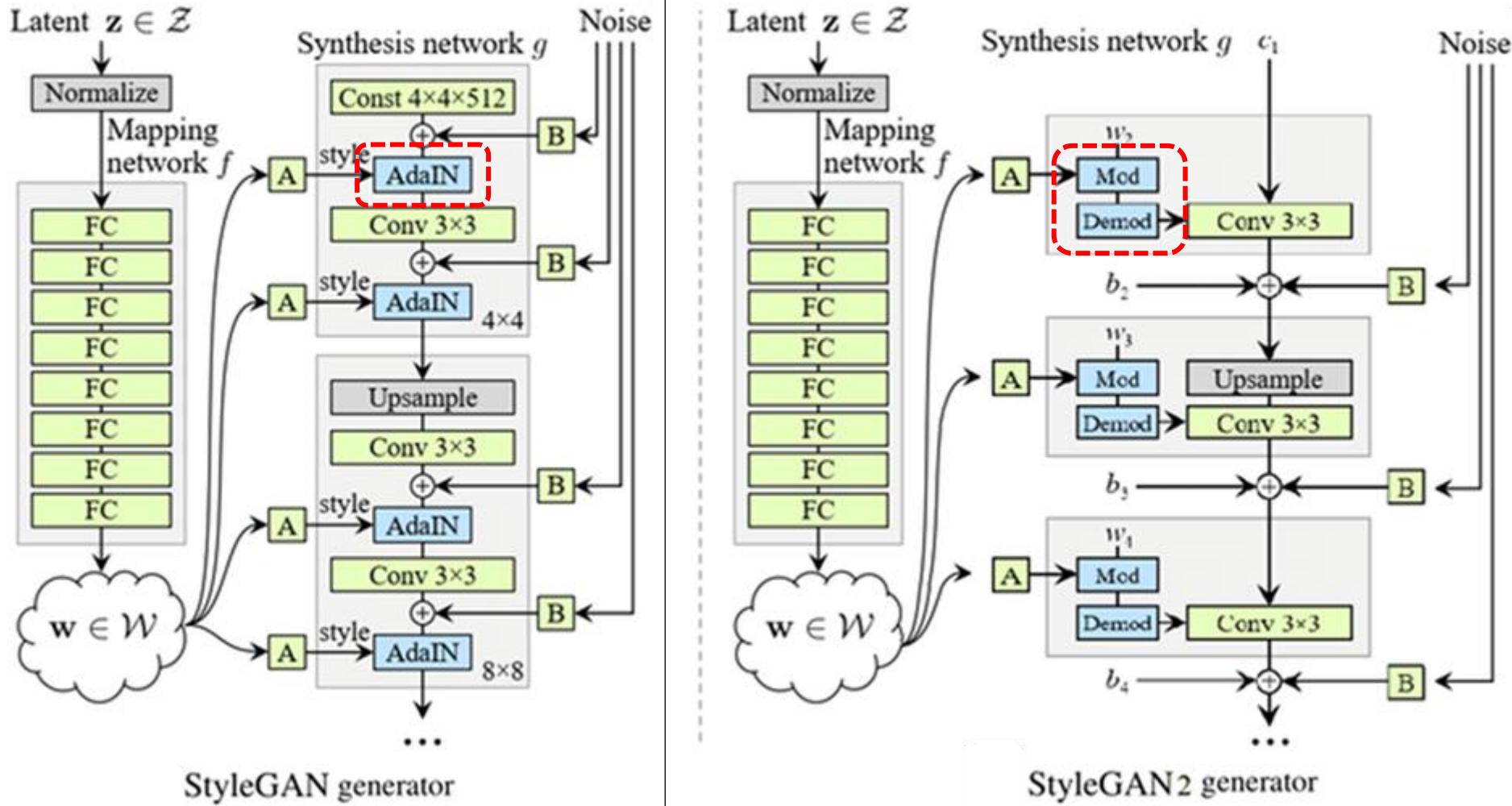
StyleGAN generator



StyleGAN2 generator

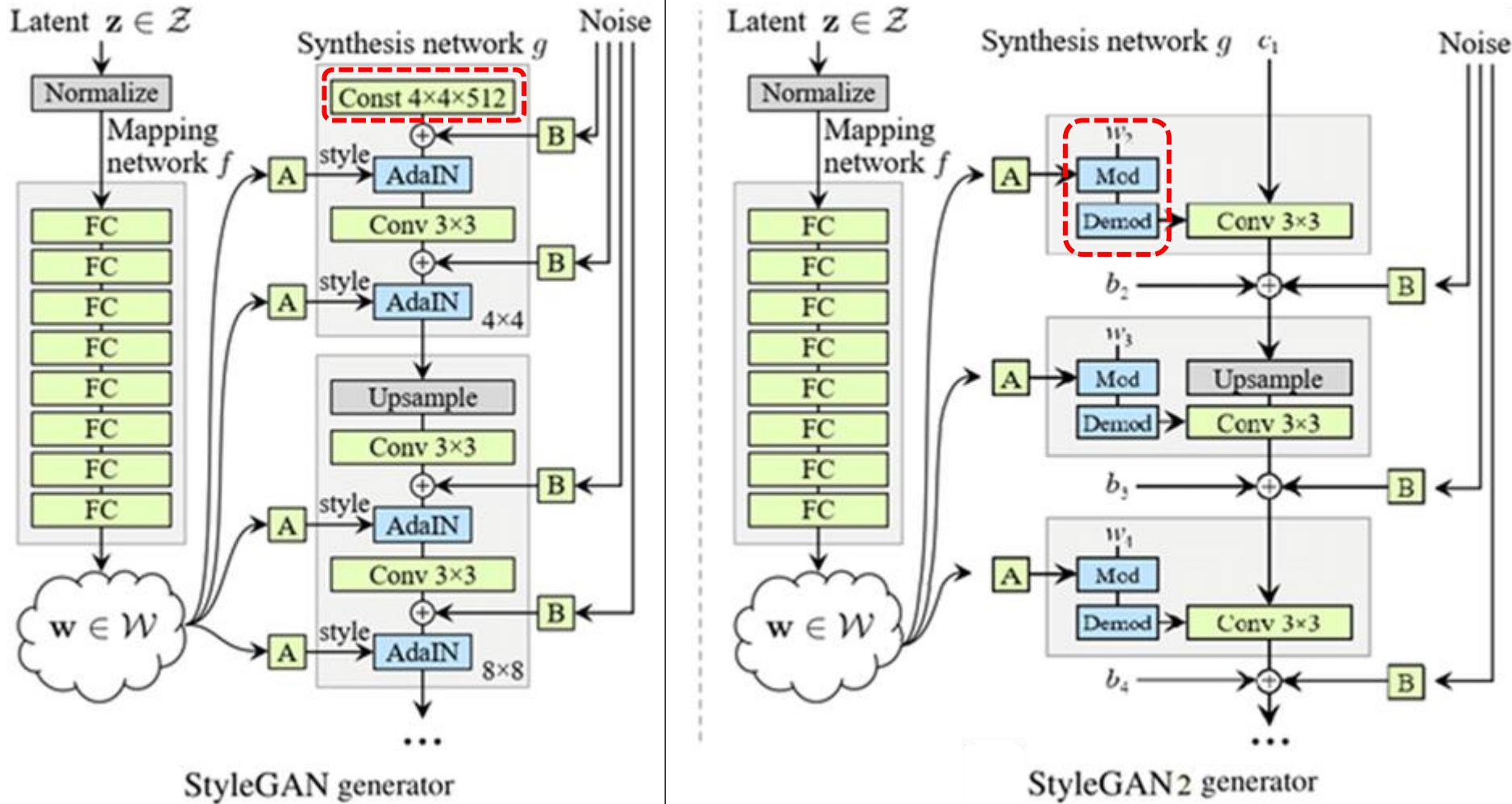
1. Mapping Network (f)

- Ambas versiones tienen una red de mapeo totalmente conectada que transforma el vector latente $z \in \mathcal{Z}$ en $w \in \mathcal{W}$.
- Esta red busca **desentrelazar factores latentes**, haciendo que el espacio \mathcal{W} sea más interpretable.



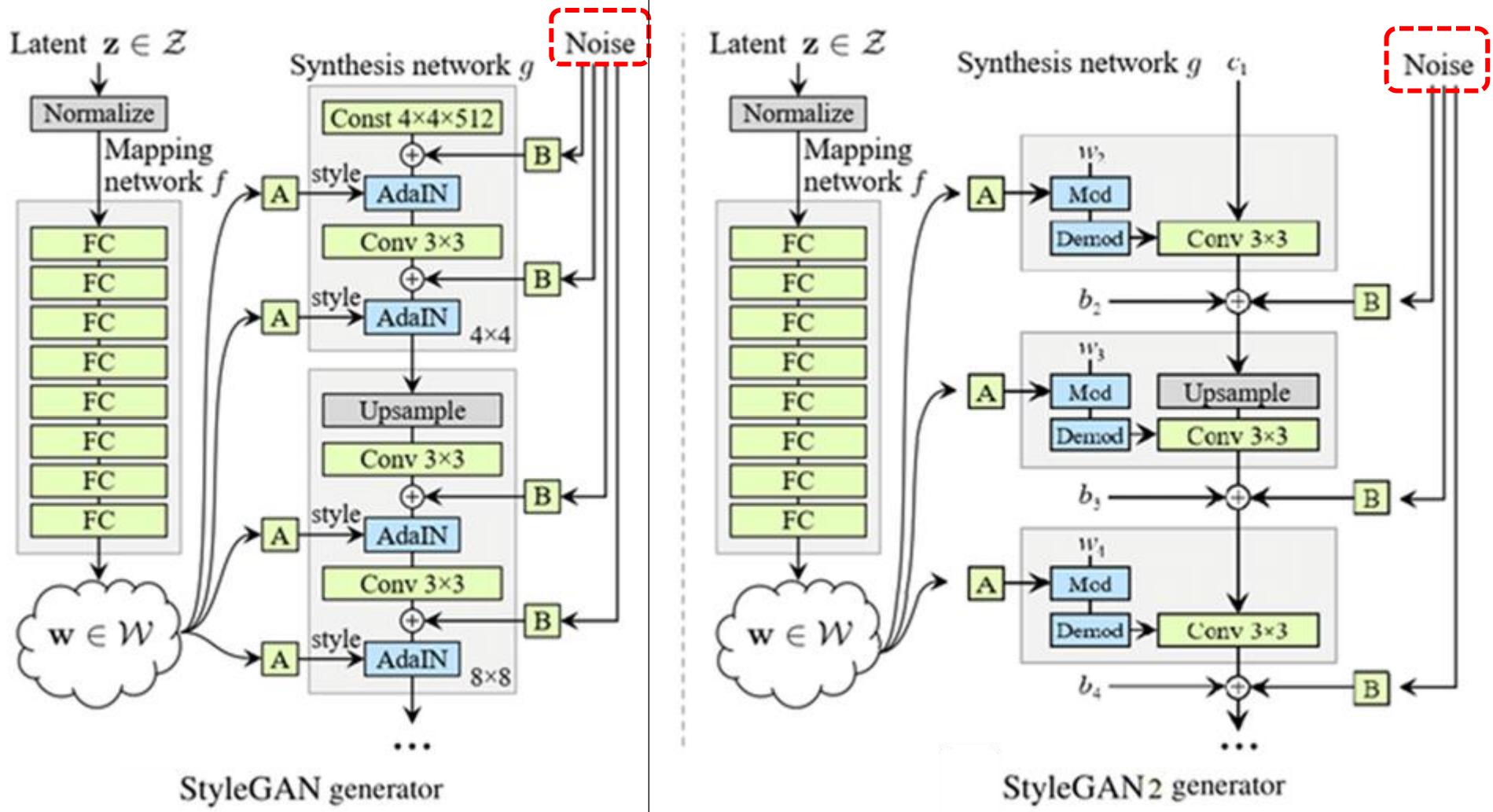
2. Aplicación del estilo

- En StyleGAN1, el vector w modula cada capa mediante AdaIN (Adaptive Instance Normalization).
- En StyleGAN2, AdaIN se elimina y se sustituye por Modulated Convolution + Demodulation, lo que:
 - Evita artefactos visuales tipo "burbujas" (causados por normalizar estadísticos por canal).
 - Permite un control más estable y local del estilo.



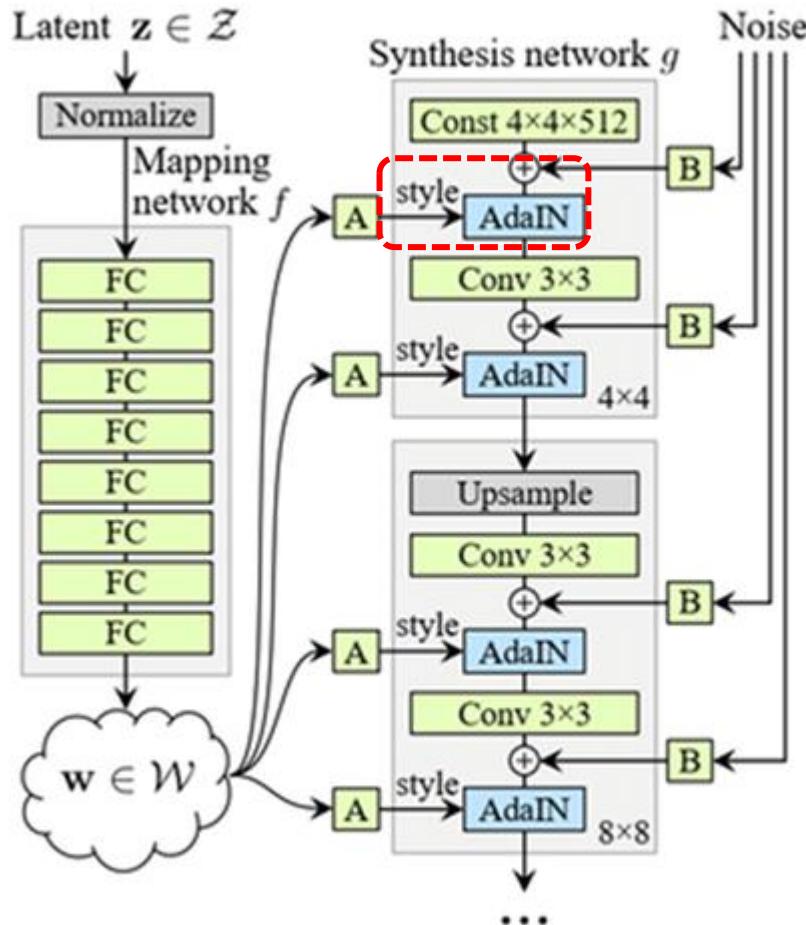
3. Synthesis Network (g)

- En StyleGAN1, el proceso comienza desde una **constante aprendida** ($4 \times 4 \times 512$) y se expande mediante *upsampling + convoluciones*.
- En StyleGAN2, se mantiene la estructura, pero las convoluciones ahora son **moduladas por el estilo** y no por normalización, produciendo imágenes con **mejor coherencia espacial**.



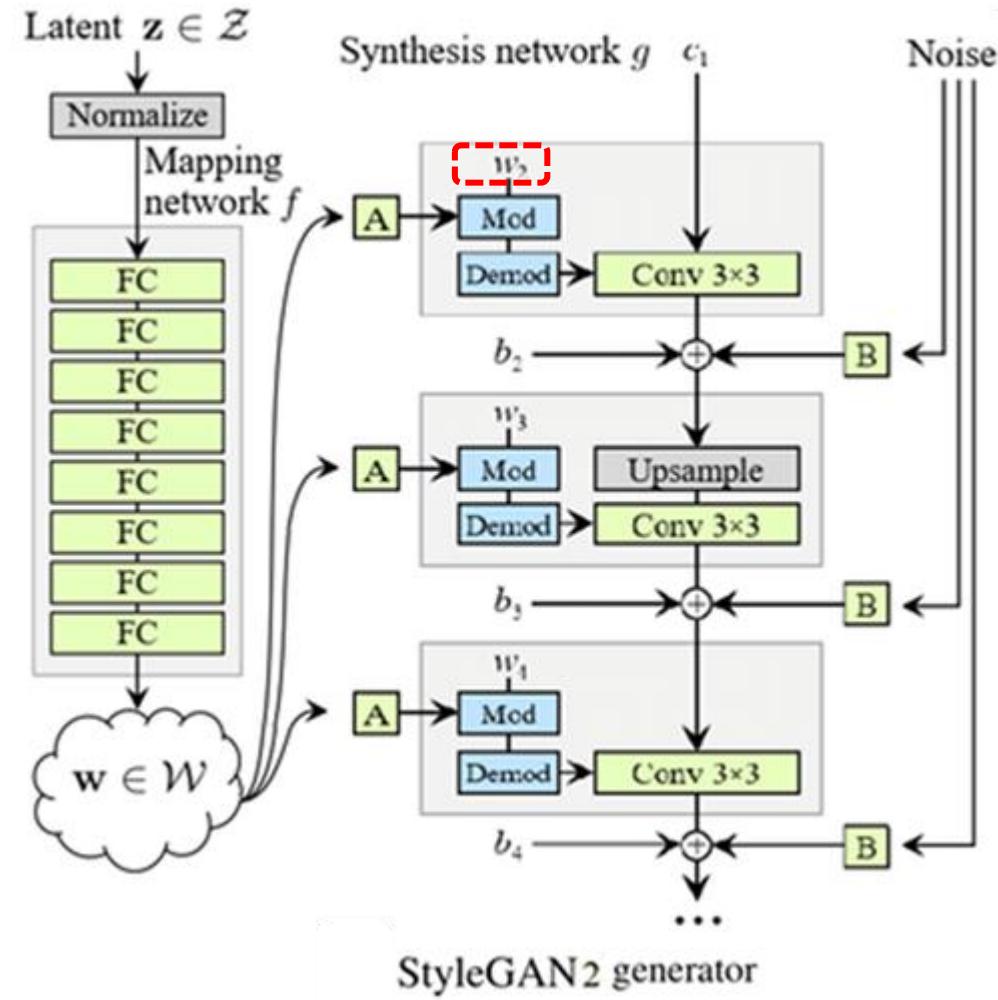
4. Inyección de ruido

- En ambas versiones se inyecta ruido gaussiano independiente por capa para generar detalles estocásticos (poros, cabello, textura de piel).
- StyleGAN2 lo mantiene pero con mejor regularización.



5. Espacio intermedio \mathcal{W} StyleGAN generator

- En StyleGAN1, w controla el estilo mediante AdaIN.
- En StyleGAN2, cada capa recibe su propio w_i , (lo que permite **un control más granular** de atributos locales y globales).
 - El vector \mathcal{W} generado por la *mapping network* se expande a una secuencia de estilos $[w_1, w_2, \dots, w_L]$, uno por cada capa del generador.
 - Esto define el espacio extendido \mathcal{W}^+ , donde cada w_i controla rasgos a distintas escalas (globales en capas bajas, locales en capas altas), permitiendo un control jerárquico y más disentangled sobre los atributos visuales.



StyleGAN2 generator

StyleGAN2

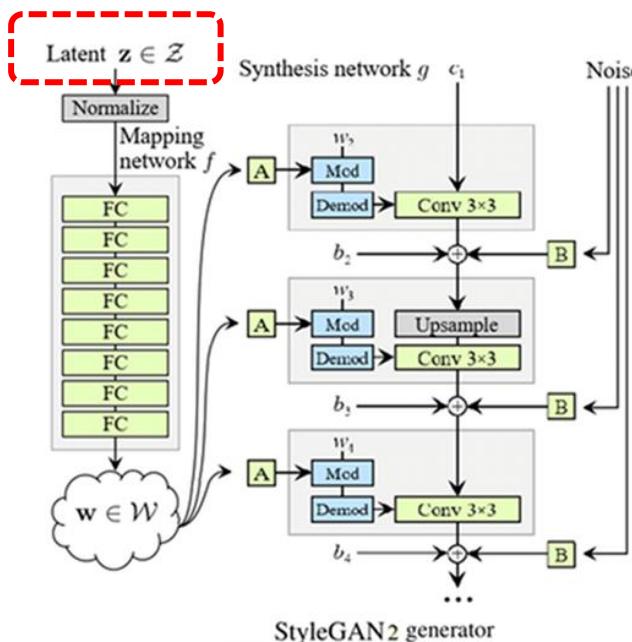
estrategias_control_stylegan.ipynb



• Control Latente

Se manipula el espacio latente para modificar o combinar características internas.

Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación y mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN



1. Instalar ninja y pycuda

ninja: Sistema de compilación optimizado que coordina la construcción rápida de módulos en C++ y CUDA, utilizado por PyTorch y otras librerías para acelerar la generación de extensiones nativas durante la instalación o el entrenamiento en GPU.

pycuda: Proporciona una interfaz directa entre Python y CUDA, permitiendo ejecutar kernels en la GPU, gestionar memoria y realizar operaciones de cómputo paralelo de bajo nivel necesarias para modelos como StyleGAN.

2. Cargar StyleGAN2-ADA preentrenado (FFHQ) - NVIDIA oficial

Entrenado a resolución 1024×1024.

3. Generar y visualizar imágenes base

1. **Latentes aleatorios:** crea `n_images` vectores `z` en el espacio latente (`G.z_dim`).
2. **Inferencia sin gradientes:** ejecuta `G(z, c)` dentro de `torch.no_grad()` para ahorrar memoria y acelerar la ejecución ya que estamos en modo inferencia.
3. **Truncation:** aplica `truncation_psi = 0.7` para equilibrar realismo y diversidad en las imágenes generadas.
 - Valores bajos (≈ 0.5) producen imágenes más realistas y estables.
 - Valores altos (≈ 1.0) producen imágenes más variadas pero a veces menos coherentes.
 - En este caso, 0.7 busca un equilibrio entre realismo y diversidad.
4. **Ruido determinista:** usa `noise_mode="const"` para resultados reproducibles.
5. **Escalado y visualización:** reescalas el tensor de `[-1, 1]` a `[0, 1]` y muestra las imágenes con Matplotlib.

Salida: una cuadrícula de rostros realistas generados por StyleGAN2-FFHQ, listos para inspección visual.

```
n_images = 4
z_dim = G.z_dim
z = torch.randn(n_images, z_dim, device=device) # vectores latentes aleatorios
c = None # sin etiquetas (modelo no condicional)

with torch.no_grad(): #Desactiva el cálculo del gradiente porque estamos en modo inferencia
    imgs = G(z, c, truncation_psi=0.7, noise_mode='const')
```

Imágenes generadas con StyleGAN2-FFHQ (NVIDIA, PyTorch)



Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 4

4. Control latente

4.1 Control latente continuo por interpolación entre dos códigos z

Idea: observar cómo cambia la identidad o combinar rasgos de distintos códigos latentes.

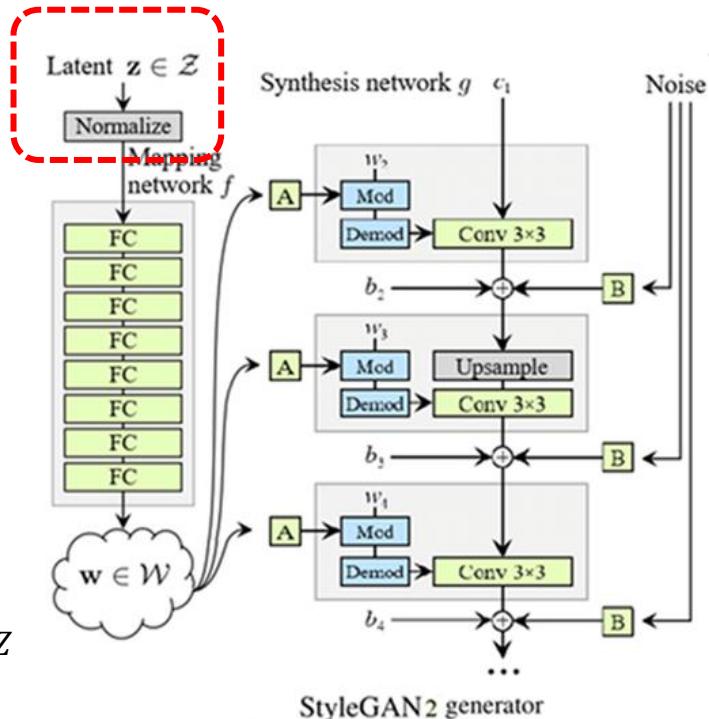
```
# Control latente por interpolación entre dos códigos z
z1 = torch.randn(1, G.z_dim, device=device)
z2 = torch.randn(1, G.z_dim, device=device) } Se generan dos vectores aleatorios distintos del espacio latente Z

steps = 7
alphas = np.linspace(0, 1, steps)
imgs = []

with torch.no_grad():
    for a in alphas:
        z_interp = (1 - a) * z1 + a * z2
        img = G(z_interp, None, truncation_psi=0.7, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)

plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, steps, i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")
plt.suptitle("Interpolación lineal entre dos códigos latentes z", fontsize=14)
plt.show()
```

$$\dim(Z) = 512$$



Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable manipulable.	StyleGAN
Interpolación mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 4

4. Control latente

4.1 Control latente continuo por interpolación entre dos códigos z

Idea: observar cómo cambia la identidad o combinar rasgos de distintos códigos latentes.

```
# Control latente por interpolación entre dos códigos z
```

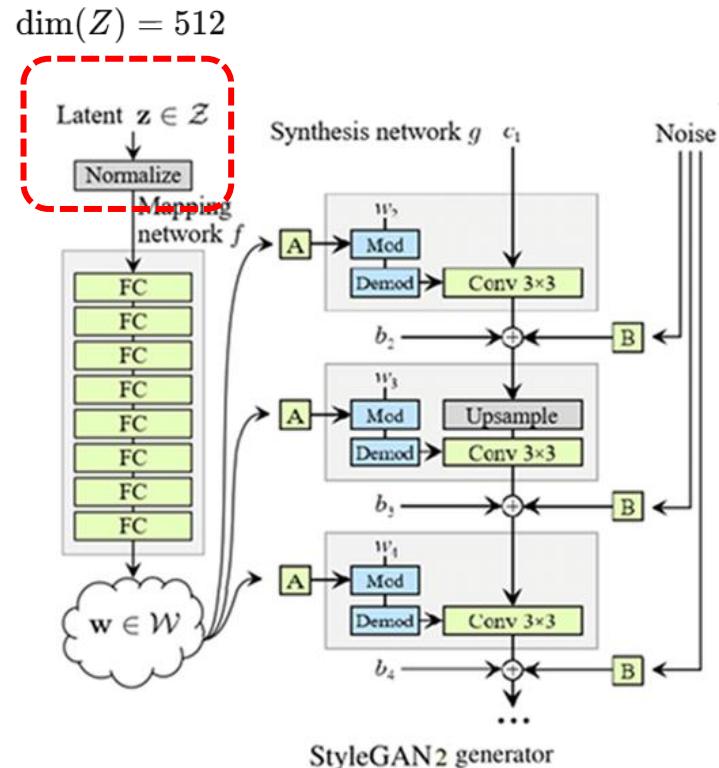
```
z1 = torch.randn(1, G.z_dim, device=device)
z2 = torch.randn(1, G.z_dim, device=device)

steps = 7
alphas = np.linspace(0, 1, steps)
imgs = []

with torch.no_grad():
    for a in alphas:
        z_interp = (1 - a) * z1 + a * z2
        img = G(z_interp, None, truncation_psi=0.7, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)
```

```
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, steps, i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")
plt.suptitle("Interpolación lineal entre dos códigos latentes z", fontsize=14)
plt.show()
```

estراتيجias_control_stylegan.ipynb



Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable manipulable.	StyleGAN
Interpolación mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 4

4. Control latente

4.1 Control latente continuo por interpolación entre dos códigos z

Idea: observar cómo cambia la identidad o combinar rasgos de distintos códigos latentes.

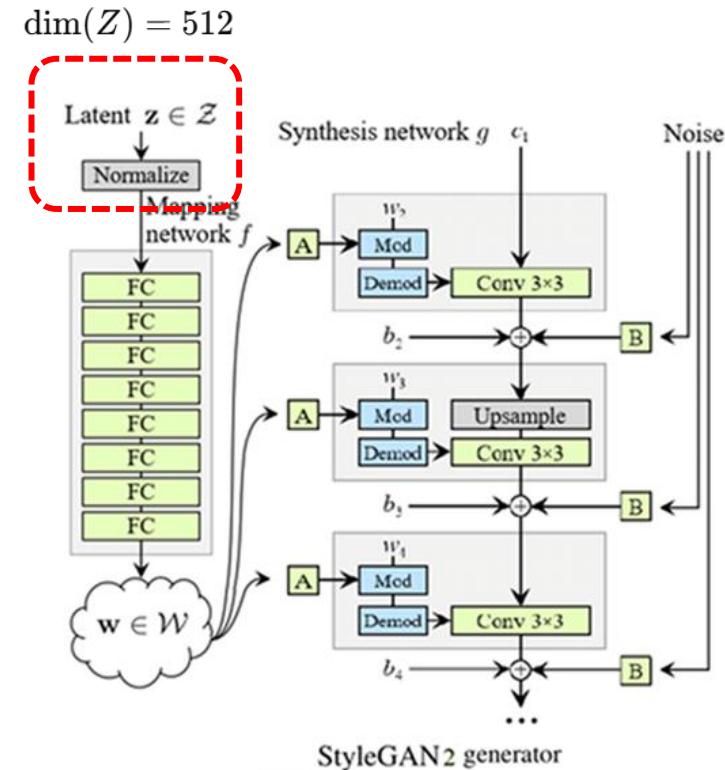
```
# Control latente por interpolación entre dos códigos z

z1 = torch.randn(1, G.z_dim, device=device)
z2 = torch.randn(1, G.z_dim, device=device)

steps = 7
alphas = np.linspace(0, 1, steps)
imgs = []

with torch.no_grad():
    for a in alphas:
        z_interp = (1 - a) * z1 + a * z2
        img = G(z_interp, None, truncation_psi=0.7, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)

plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, steps, i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")
plt.suptitle("Interpolación lineal entre dos códigos latentes z", fontsize=14)
plt.show()
```



- En cada paso, se calcula el punto intermedio: $z_{\text{interp}} = (1 - a)z_1 + az_2$
- Es decir, una **combinación lineal** entre los dos códigos.
- Luego, se pasa ese código interpolado al generador G para producir la imagen correspondiente.
 - **truncation_psi=0.7** controla qué tan cerca del "centro" del espacio latente se generan las muestras (evita artefactos).
 - **noise_mode='const'** fija el ruido estocástico para que los cambios se deban solo al desplazamiento en el espacio latente.
 - **clamp(-1, 1)** y la división por 2 normalizan la imagen al rango [0, 1].

Interpolación lineal entre dos códigos latentes z



Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación y mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 5

4.2 Espacio disentangled

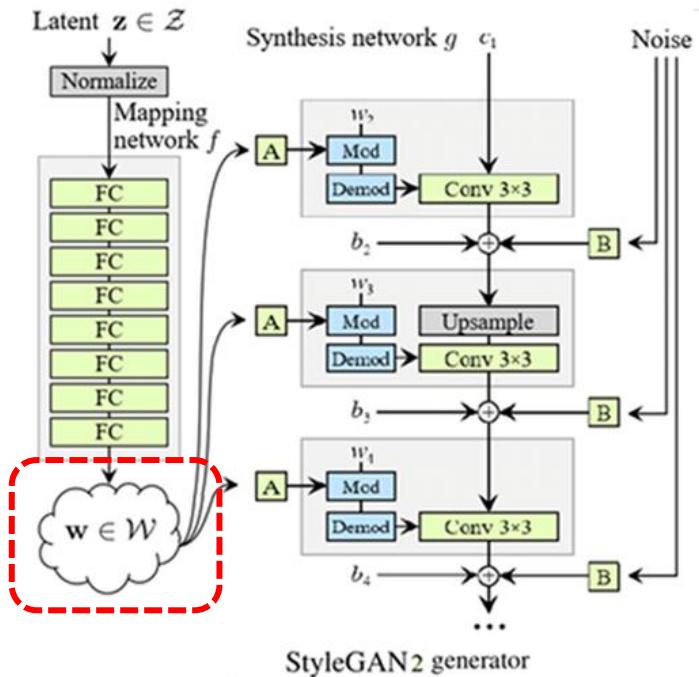
4.2.1 Descubrir direcciones semánticas

Idea: el espacio intermedio W de StyleGAN2 se puede descomponer en direcciones que corresponden a atributos visuales. Usaremos PCA para encontrar direcciones dominantes — sin etiquetas — y observar cómo cada una afecta el resultado.

estrategias_control_stylegan.ipynb



1. **El espacio latente “tradicional” Z $\dim(Z) = 512$**
 - Es el espacio donde se generan los vectores latentes iniciales $z \sim \mathcal{N}(0, I)$.
 - Se usa en la mayoría de las GANs clásicas (DCGAN).
 - Cada punto z representa una “semilla” de generación y se pasa directamente al generador.
 - En este espacio, la relación entre dimensiones y atributos visuales suele ser **altamente entrelazada (entangled)**: cambiar una coordenada puede alterar simultáneamente varios rasgos.



2. **El espacio intermedio W $\dim(W) = 512$**
 - En StyleGAN, el vector z se transforma mediante la *mapping network* f :

$$w = f(z)$$
 - El resultado $w \in \mathcal{W}$ también es un **vector latente**, pero pertenece a un **espacio latente intermedio**, no directamente al de entrada.
 - Es decir, W es un **espacio latente aprendido**, diseñado para tener una estructura más semánticamente organizada.

```

# =====
# Control latente disentangled con PCA en el espacio W
# =====

from sklearn.decomposition import PCA
import numpy as np
import torch
import matplotlib.pyplot as plt

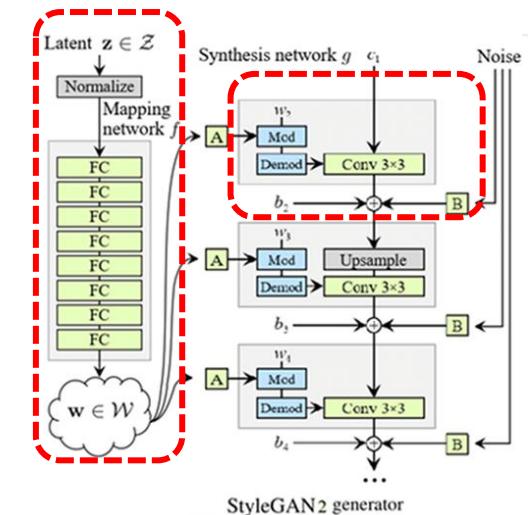
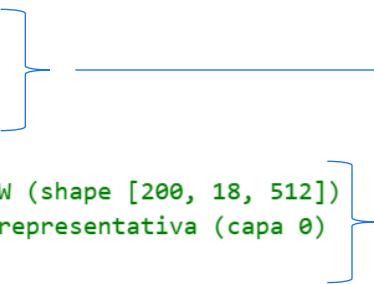
# 1. Generar vectores latentes z y proyectarlos a W
n_samples = 200
torch.manual_seed(42)
z = torch.randn(n_samples, G.z_dim, device=device)

with torch.no_grad():
    w = G.mapping(z, None) # proyección al espacio W (shape [200, 18, 512])
w_np = w[:, 0, :].cpu().numpy() # tomamos una capa representativa (capa 0)

# 2. PCA: descubrir direcciones semánticas no supervisadas
pca = PCA(n_components=5)
pca.fit(w_np)
components = pca.components_

print("PCA aplicado sobre W. Varianza explicada:", np.round(pca.explained_variance_ratio_, 3))

```



- Se generan 200 vectores latentes $z \in \mathbb{R}^{512}$.
- Cada uno se proyecta mediante la *mapping network* al espacio intermedio W , produciendo $w \in \mathbb{R}^{200 \times 18 \times 512}$.
- Se toma solo la **primera capa** ($w[:, 0, :]$) para simplificar el análisis (las demás son similares).

<https://youtu.be/k3CWA2GBb8o>

PCA encuentra direcciones ortogonales de máxima varianza.

```
# =====
# Control latente disentangled con PCA en el espacio W
# =====

from sklearn.decomposition import PCA
import numpy as np
import torch
import matplotlib.pyplot as plt

# 1. Generar vectores latentes z y proyectarlos a W
n_samples = 200
torch.manual_seed(42)
z = torch.randn(n_samples, G.z_dim, device=device)

with torch.no_grad():
    w = G.mapping(z, None) # proyección al espacio W (shape [200, 18, 512])
w_np = w[:, 0, :].cpu().numpy() # tomamos una capa representativa (capa 0)
```

```
# 2. PCA: descubrir direcciones semánticas no supervisadas
```

```
pca = PCA(n_components=5)
```

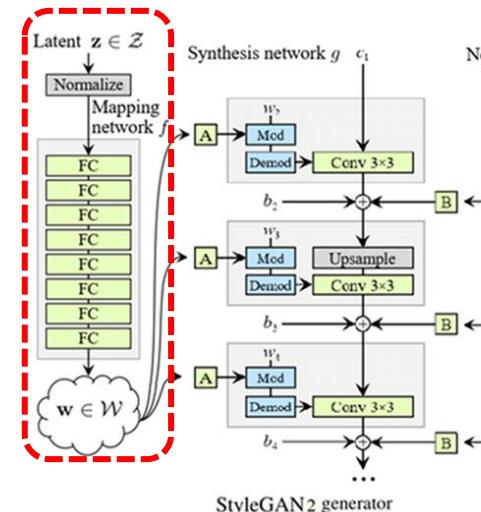
```
pca.fit(w_np)
```

```
components = pca.components_
```

```
print("PCA aplicado sobre W. Varianza explicada:", np.round(pca.explained_variance_ratio_, 3))
```

PCA aplicado sobre W. Varianza explicada: [0.094 0.075 0.067 0.064 0.061]

- Las cinco primeras componentes principales, en conjunto, explican aproximadamente el 35 % de la variabilidad total presente en los vectores W ($0.090 + 0.073 + 0.068 + 0.064 + 0.055 \approx 0.35$).
- Eso implica que el espacio W de StyleGAN2 es **rico y de alta dimensionalidad efectiva**: no hay unas pocas direcciones que dominen completamente la variación, sino muchas direcciones que aportan gradualmente cambios visuales.
- Los cambios más evidentes (edad, pose, expresión) están probablemente asociados con las primeras componentes, mientras que otras características visuales más finas se distribuyen entre muchas direcciones menores.



StyleGAN2 generator

3. Visualizar el efecto de una componente PCA (dirección semántica fija)

```
# Seleccionar componente (0 a 4)
ic = 0
```

Generar nuevos vectores base z → w (sin semilla fija)

```
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7)
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)
```

```
imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)
```

4. Mostrar resultados

```
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")

plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()
```

ic contiene el valor de la componente principal que nos interesa visualizar.

```
# 3. Visualizar el efecto de una componente PCA (dirección semántica fija)
```

```
# Seleccionar componente (0 a 4)
```

```
ic = 0
```

```
# Generar nuevos vectores base z → w (sin semilla fija)
```

```
n_images = 5
```

```
z = torch.randn(n_images, G.z_dim, device=device)
```

```
with torch.no_grad():
```

```
    w_base = G.mapping(z, None)
```

```
# Definir valores de desplazamiento
```

```
alpha_values = np.linspace(-4, 4, 7)
```

```
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)
```

```
imgs = []
```

```
with torch.no_grad():
```

```
    for a in alpha_values:
```

```
        w_mod = w_base.clone()
```

```
        w_mod += a * direction # aplicar desplazamiento
```

```
        img = G.synthesis(w_mod, noise_mode='const')
```

```
        imgs.append((img.clamp(-1, 1) + 1) / 2)
```

```
# 4. Mostrar resultados
```

```
plt.figure(figsize=(14, 3))
```

```
for i, img in enumerate(imgs):
```

```
    plt.subplot(1, len(imgs), i + 1)
```

```
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
```

```
    plt.axis("off")
```

```
plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
```

```
plt.show()
```

Genera nuevos puntos de partida w_{base} (uno por cada rostro distinto) a partir de vectores aleatorios z , usando la *mapping network* de StyleGAN2.

3. Visualizar el efecto de una componente PCA (dirección semántica fija)

```
# Seleccionar componente (0 a 4)
ic = 0

# Generar nuevos vectores base z → w (sin semilla fija)
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7) →
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)

imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)

# 4. Mostrar resultados
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")

plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()
```

- Se generan **7 valores equiespaciados** entre -4 y $+4$:
[-4,-2.6667,-1.3333,0,1.3333,2.6667,4]
- Cada valor α representa **cuánto te mueves en una dirección PCA**.
 - Valores negativos → mueves w en un sentido.
 - Valores positivos → mueves w en el sentido opuesto.
 - $\alpha = 0 \rightarrow$ es el punto base (la media del espacio).
- En términos semánticos, si la dirección representa "sonrisa", entonces α negativo podría producir rostros más serios y α positivo, rostros más sonrientes.

```

# 3. Visualizar el efecto de una componente PCA (dirección semántica fija)

# Seleccionar componente (0 a 4)
ic = 0

# Generar nuevos vectores base z → w (sin semilla fija)
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7)
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512) —————→

imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)

# 4. Mostrar resultados
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")

plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()

```

Selecciona la dirección PCA correspondiente y la adapta al tamaño de w para poder sumarla.

3. Visualizar el efecto de una componente PCA (dirección semántica fija)

```

# Seleccionar componente (0 a 4)
ic = 0

# Generar nuevos vectores base z → w (sin semilla fija)
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

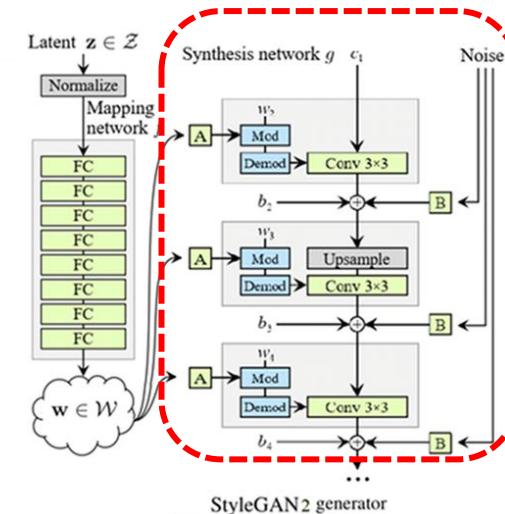
# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7)
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)

img = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        img.append((img.clamp(-1, 1) + 1) / 2)

# 4. Mostrar resultados
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")

plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()

```



- Para cada valor a (cada desplazamiento):
 - Se clona w_{base} → mantienes las 5 identidades originales.
 - A cada una se le aplica el **mismo desplazamiento** $\Delta w = a \cdot \text{dirección}$
- Se llama al generador $G.\text{synthesis}(\dots)$ para reconstruir las imágenes correspondientes.
- $\text{noise_mode}=\text{'const'}$ mantiene el ruido fijo, evitando variaciones no semánticas (como texturas aleatorias o granulado).
- $\text{img.clamp}(-1, 1) + 1 / 2$ normaliza los valores al rango $[0, 1]$ para visualizarlos con matplotlib.

```

# 3. Visualizar el efecto de una componente PCA (dirección semántica fija)

# Seleccionar componente (0 a 4)
ic = 0

# Generar nuevos vectores base z → w (sin semilla fija)
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7)
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)

imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)

# 4. Mostrar resultados
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")

plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()

```



- Se colocan todas las imágenes generadas en una fila.
- Cada imagen corresponde a un valor distinto de α .
- Al recorrerlas de izquierda a derecha, se observa una **interpolación continua** a lo largo de esa dirección PCA.

3. Visualizar el efecto de una componente PCA (dirección semántica fija)

```
# Seleccionar componente (0 a 4)
ic = 0
```

```
# Generar nuevos vectores base z → w (sin semilla fija)
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7)
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)
```

```
imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)
```

```
# 4. Mostrar resultados
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")
```

```
plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()
```

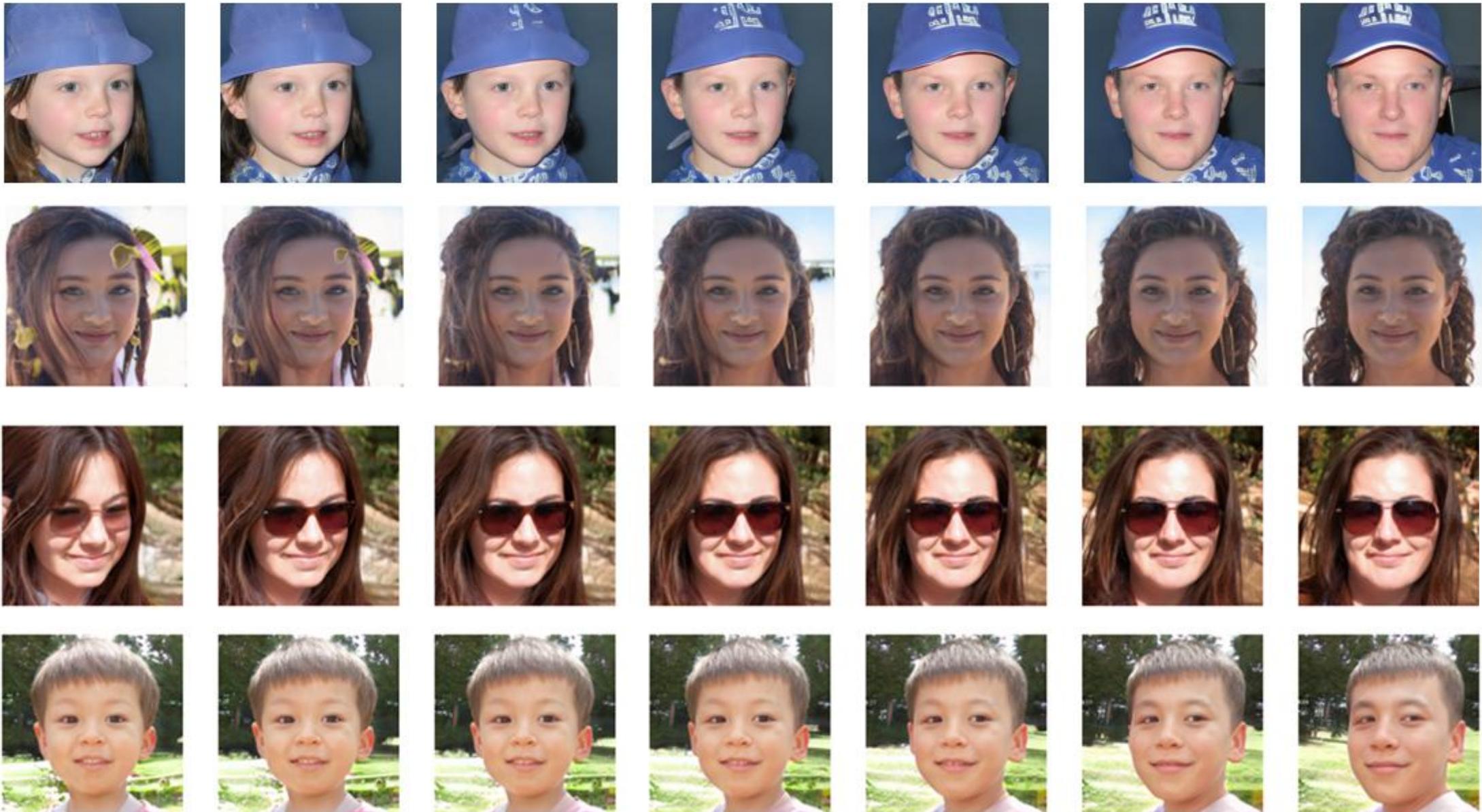
Ejecuta varias veces el experimento utilizando la **componente principal 1** ($ic = 0$) y observa los cambios en las imágenes generadas.

¿Qué tipo de **características visuales o semánticas** crees que están siendo modificadas por esta dirección en el espacio W ?



¿Corresponden a rasgos globales (como edad, pose) o locales (como textura, color, iluminación)?

Control latente disentangled: variación a lo largo de la componente PCA 1
edad ↔ madurez facial



3. Visualizar el efecto de una componente PCA (dirección semántica fija)

```
# Seleccionar componente (0 a 4)
ic = 0
```

```
# Generar nuevos vectores base z → w (sin semilla fija)
n_images = 5
z = torch.randn(n_images, G.z_dim, device=device)
with torch.no_grad():
    w_base = G.mapping(z, None)

# Definir valores de desplazamiento
alpha_values = np.linspace(-4, 4, 7)
direction = torch.from_numpy(components[ic]).to(device).reshape(1, 1, 512)
```

```
imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w_base.clone()
        w_mod += a * direction # aplicar desplazamiento
        img = G.synthesis(w_mod, noise_mode='const')
        imgs.append((img.clamp(-1, 1) + 1) / 2)
```

```
# 4. Mostrar resultados
plt.figure(figsize=(14, 3))
for i, img in enumerate(imgs):
    plt.subplot(1, len(imgs), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.axis("off")
```

```
plt.suptitle(f"Control latente disentangled: variación a lo largo de la componente PCA {ic+1}", fontsize=14)
plt.show()
```

Ejecuta varias veces el experimento utilizando la **componente principal 2** ($ic = 0$) y observa los cambios en las imágenes generadas.

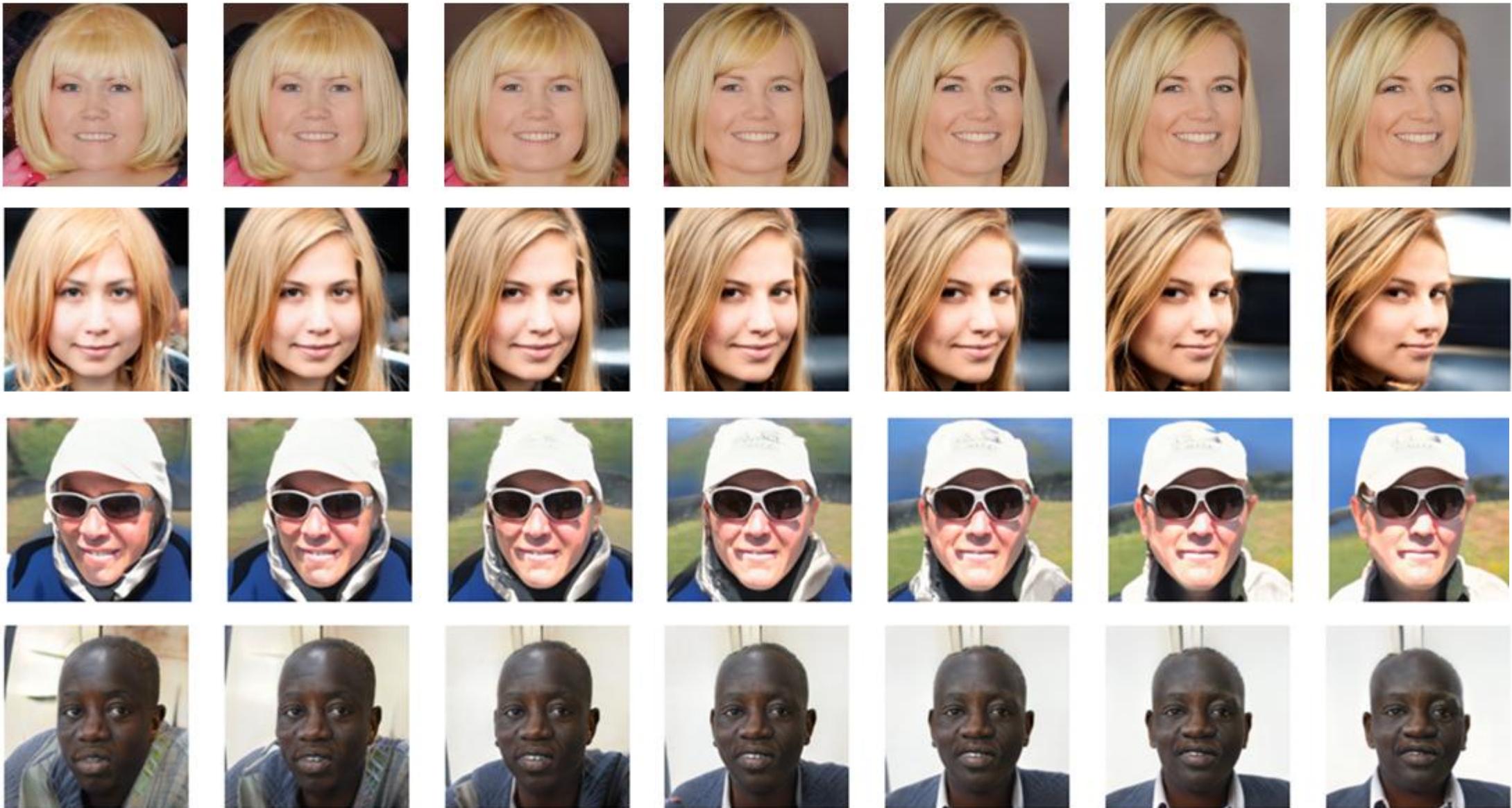
¿Qué tipo de **características visuales o semánticas** crees que están siendo modificadas por esta dirección en el espacio W ?



¿Corresponden a rasgos globales (como edad, pose) o locales (como textura, color, iluminación)?

Control latente disentangled: variación a lo largo de la componente PCA 2

?





https://youtu.be/jdTICDa_eAI

Otras formas de obtener las direcciones semánticas

Linear Classifier Boundaries (InterFaceGAN)

Introducido por Shen et al. (CVPR 2020):

- Se entrena n **clásificadores lineales** (SVM o regresión logística) sobre vectores w anotados con atributos (ej. sonrisa, edad, género).
- Cada clasificador define una **dirección semántica** en el espacio W .
- Luego puedes desplazarte a lo largo de esa dirección para modificar el atributo.
- *InterFaceGAN* fue el primer trabajo que mostró manipulación controlada y separable de edad, expresión, género, etc.

Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación / mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 6

4.2.2 Interpolación en una dirección semántica conocida (**edad**) en el espacio W

```
# =====
# Descarga de direcciones semánticas publicadas (InterFaceGAN)
# =====

!mkdir -p boundaries
!wget -q -O boundaries/age_boundary.npy https://github.com/genforce/interfacegan/raw/master/boundaries/stylegan_ffhq_age_boundary.npy
!wget -q -O boundaries/smile_boundary.npy https://github.com/genforce/interfacegan/raw/master/boundaries/stylegan_ffhq_smile_boundary.npy

print("Direcciones descargadas correctamente.")
```

estrategias_control_stylegan.ipynb



```

# =====
# Control latente: "Edad"
# =====

import torch
import numpy as np
import matplotlib.pyplot as plt

# 1. Dirección "edad" (normalizada)
age_dir = np.load("boundaries/age_boundary.npy")
age_dir = torch.from_numpy(age_dir).to(device)
age_dir = age_dir / torch.norm(age_dir)

# 2. Identidad base
z = torch.randn(1, G.z_dim, device=device)
with torch.no_grad():
    w = G.mapping(z, None)

# 3. Rango moderado y continuo
#   ±6 suele ser el máximo seguro para mantener coherencia
alpha_values = np.linspace(-6, 6, 9)

```

- Carga el vector que representa la dirección semántica "edad" previamente cargada
- Lo convierte en un tensor de PyTorch y lo pasa al dispositivo (CPU o GPU).
- Lo normaliza para que tenga longitud unitaria, lo cual permite aplicar desplazamientos proporcionales y coherentes al movernos en esa dirección del espacio latente.

```

# =====
# Control latente: "Edad"
# =====

import torch
import numpy as np
import matplotlib.pyplot as plt

# 1. Dirección "edad" (normalizada)
age_dir = np.load("boundaries/age_boundary.npy")
age_dir = torch.from_numpy(age_dir).to(device)
age_dir = age_dir / torch.norm(age_dir)

# 2. Identidad base
z = torch.randn(1, G.z_dim, device=device)
with torch.no_grad():
    w = G.mapping(z, None)

# 3. Rango moderado y continuo
#   ±6 suele ser el máximo seguro para mantener coherencia
alpha_values = np.linspace(-6, 6, 9)

```

- Se genera un **vector aleatorio en el espacio Z** y se proyecta al **espacio intermedio W** usando la red de *mapping* de StyleGAN.
- Este vector **w** es el punto base a partir del cual se aplicarán los desplazamientos a lo largo de la dirección "edad".
- `torch.no_grad()` desactiva el cálculo de gradientes, ya que aquí solo se generan muestras, no se entrena nada.

```
# =====
# Control latente: "Edad"
# =====

import torch
import numpy as np
import matplotlib.pyplot as plt

# 1. Dirección "edad" (normalizada)
age_dir = np.load("boundaries/age_boundary.npy")
age_dir = torch.from_numpy(age_dir).to(device)
age_dir = age_dir / torch.norm(age_dir)

# 2. Identidad base
z = torch.randn(1, G.z_dim, device=device)
with torch.no_grad():
    w = G.mapping(z, None)

# 3. Rango moderado y continuo
#   ±6 suele ser el máximo seguro para mantener coherencia
alpha_values = np.linspace(-6, 6, 9)
```

} Define los valores de desplazamiento α para moverse en la dirección de "edad".

```
# 4. Aplicar dirección con peso decreciente en capas altas
imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w.clone()
        # estructura (capas 0-6): efecto fuerte
        w_mod[:, :7, :] += a * 1.2 * age_dir
        # expresión/media (capas 7-11): efecto medio
        w_mod[:, 7:12, :] += a * 0.7 * age_dir
        # textura fina (capas 12-17): efecto leve
        w_mod[:, 12:, :] += a * 0.25 * age_dir

        img = (G.synthesis(w_mod, noise_mode='const').clamp(-1, 1) + 1) / 2
        imgs.append(img)

# 5. Visualización
plt.figure(figsize=(20, 4))
for i, (a, img) in enumerate(zip(alpha_values, imgs)):
    plt.subplot(1, len(alpha_values), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.title(f" $\alpha = {a:.1f}$ ", fontsize=10)
    plt.axis("off")

plt.suptitle("Control latente: de niño a viejo (dirección 'edad')", fontsize=16, y=0.95)
plt.tight_layout(rect=[0, 0, 1, 0.92])
plt.show()
```



Aplica la dirección “edad” dentro del espacio latente W , pero de forma **controlada por capas**, para modificar diferentes niveles de detalle de la imagen.

```

# 4. Aplicar dirección con peso decreciente en capas altas
imgs = []
with torch.no_grad():
    for a in alpha_values:
        w_mod = w.clone()
        # estructura (capas 0-6): efecto fuerte
        w_mod[:, :7, :] += a * 1.2 * age_dir
        # expresión/media (capas 7-11): efecto medio
        w_mod[:, 7:12, :] += a * 0.7 * age_dir
        # textura fina (capas 12-17): efecto leve
        w_mod[:, 12:, :] += a * 0.25 * age_dir

        img = (G.synthesis(w_mod, noise_mode='const').clamp(-1, 1) + 1) / 2
        imgs.append(img)

# 5. Visualización
plt.figure(figsize=(20, 4))
for i, (a, img) in enumerate(zip(alpha_values, imgs)):
    plt.subplot(1, len(alpha_values), i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.title(f" $\alpha = {a:.1f}$ ", fontsize=10)
    plt.axis("off")

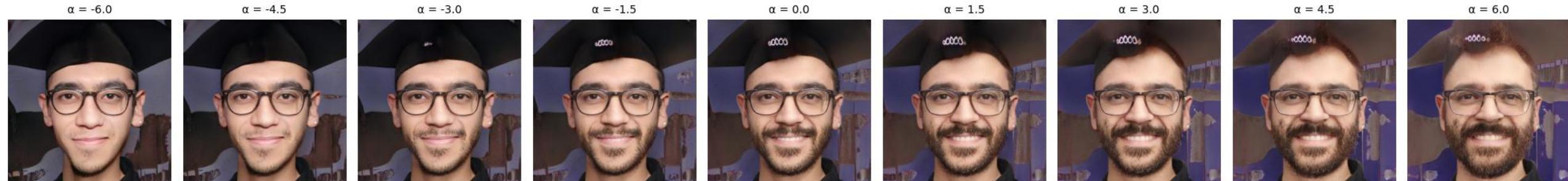
plt.suptitle("Control latente: de niño a viejo (dirección 'edad')", fontsize=16, y=0.95)
plt.tight_layout(rect=[0, 0, 1, 0.92])
plt.show()

```

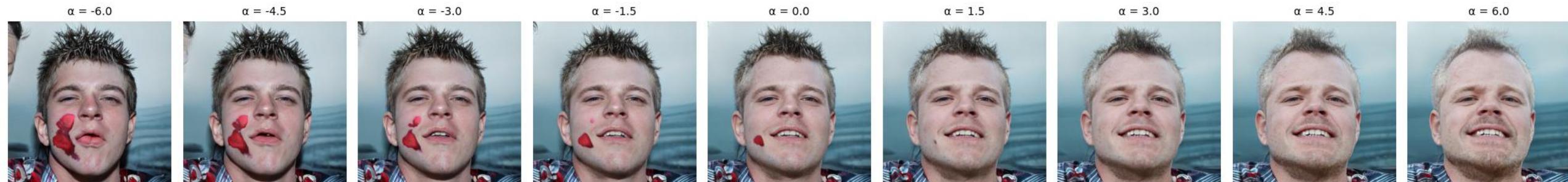


Construye una **galería ordenada de imágenes** que muestra cómo el rostro cambia gradualmente al desplazarse a lo largo de la dirección "edad", con cada columna etiquetada por su valor de α .

Control latente: de niño a viejo (dirección 'edad')



Control latente: de niño a viejo (dirección 'edad')



Control latente: de niño a viejo (dirección 'edad')





Hacer lo mismo, pero ahora en la dirección 'sonrisa'

Control latente: de serio a sonriente (dirección 'sonrisa')

$\alpha = -6.0$



$\alpha = -4.5$



$\alpha = -3.0$



$\alpha = -1.5$



$\alpha = 0.0$



$\alpha = 1.5$



$\alpha = 3.0$



$\alpha = 4.5$



$\alpha = 6.0$



Control latente: de serio a sonriente (dirección 'sonrisa')

$\alpha = -6.0$



$\alpha = -4.5$



$\alpha = -3.0$



$\alpha = -1.5$



$\alpha = 0.0$



$\alpha = 1.5$



$\alpha = 3.0$



$\alpha = 4.5$



$\alpha = 6.0$



Control latente: de serio a sonriente (dirección 'sonrisa')

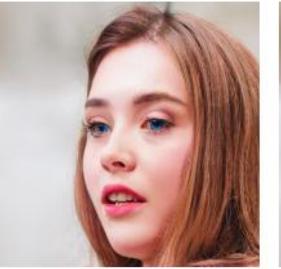
$\alpha = -6.0$



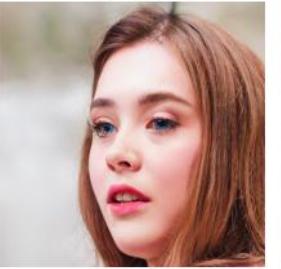
$\alpha = -4.5$



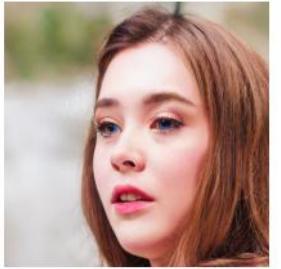
$\alpha = -3.0$



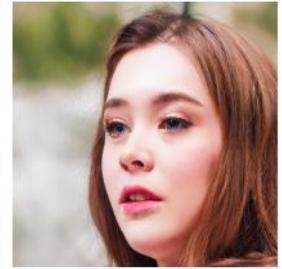
$\alpha = -1.5$



$\alpha = 0.0$



$\alpha = 1.5$



$\alpha = 3.0$



$\alpha = 4.5$



$\alpha = 6.0$



Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación / mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 8

4.2.4 Combinación Lineal de Direcciones Semánticas en el Espacio W (**edad** y **sonrisa**)

Combinar dos atributos independientes (por ejemplo, edad y expresión) usando direcciones publicadas

Este experimento muestra que las direcciones latentes pueden combinarse linealmente, generando nuevas identidades o transiciones semánticas compuestas —una demostración empírica de que el espacio W de StyleGAN2 posee una estructura localmente lineal y disentangled.

- Se toma un vector w base (una identidad aleatoria o conocida).
- Se aplican dos desplazamientos latentes independientes:
 - uno en la dirección “edad”
 - otro en la dirección “sonrisa”
- Se combinan gradualmente para observar cómo el modelo sintetiza **jóvenes serios, adultos sonrientes, adultos serios**, etc.

El resultado puede representarse como una **rejilla 3x3**, donde:

- las **filas** representan variaciones en la sonrisa, y
- las **columnas** representan variaciones en la edad.

estrategias_control_stylegan.ipynb



```
# =====
# Experimento: Composición de direcciones latentes (Edad + Sonrisa)
# =====

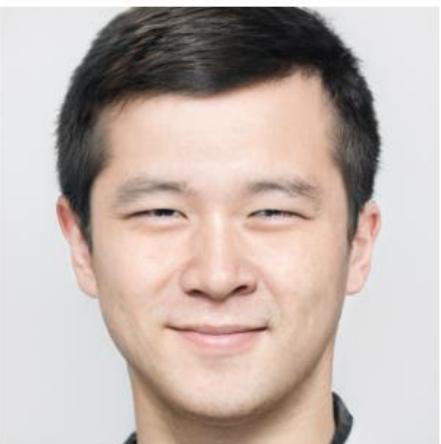
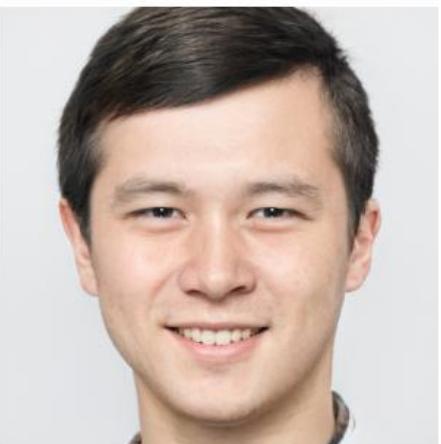
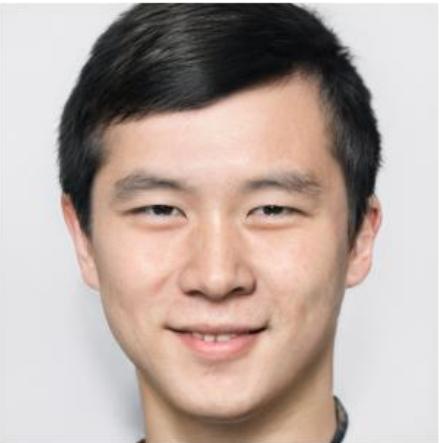
# 1. Vector latente base (identidad inicial)
z = torch.randn(1, G.z_dim, device=device)
with torch.no_grad():
    w = G.mapping(z, None)

# 2. Parámetros de desplazamiento
age_range = np.linspace(-3, 3, 3)      # columnas = edad
smile_range = np.linspace(-3, 3, 3)     # filas = sonrisa

# 3. Generar combinaciones
plt.figure(figsize=(10, 10))
for i, a in enumerate(age_range):
    for j, s in enumerate(smile_range):
        w_new = w + a * age_dir + s * smile_dir
        with torch.no_grad():
            img = (G.synthesis(w_new, noise_mode='const').clamp(-1, 1) + 1) / 2
            plt.subplot(len(age_range), len(smile_range), i * len(smile_range) + j + 1)
            plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
            plt.axis("off")

plt.suptitle("Composición de atributos: Edad × Sonrisa", fontsize=16)
plt.tight_layout()
plt.show()
```

Composición de atributos: Edad x Sonrisa



Estrategia	Principio de control	Ejemplo representativo
Espacios disentangled	Se estructura el espacio latente para que cada dimensión o dirección represente un atributo interpretable y manipulable.	StyleGAN
Interpolación y mezcla	Se interpolan o combinan vectores latentes para generar transiciones o fusiones controladas entre atributos o estilos.	StyleGAN

Celda 9

4.3 Mezcla de estilos (Style Mixing)

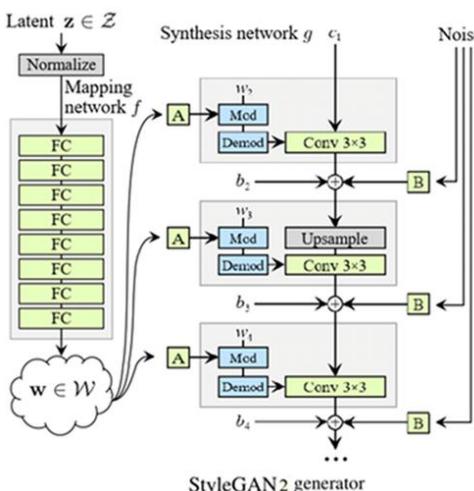
¿Qué observarás?

El rostro “mezclado” combina la estructura facial del primero (capas bajas) con el color de piel, cabello o textura del segundo (capas altas).

Si cambias crossover, puedes decidir a partir de qué nivel de detalle se mezclan:

crossover=4 → cambia forma global.

crossover=8 → cambia color y textura.



estrategias_control_stylegan.ipynb

Google
colab

```

# Control latente por mezcla de estilos (Style Mixing)

z_src = torch.randn(1, G.z_dim, device=device) # rostro fuente (forma)
z_dst = torch.randn(1, G.z_dim, device=device) # rostro destino (textura)

with torch.no_grad():
    w_src = G.mapping(z_src, None)
    w_dst = G.mapping(z_dst, None)

crossover = 6 # capa donde se combinan los estilos
w_mix = w_src.clone()
w_mix[:, crossover:, :] = w_dst[:, crossover:, :]

with torch.no_grad():
    img_src = (G.synthesis(w_src, noise_mode='const').clamp(-1, 1) + 1) / 2
    img_dst = (G.synthesis(w_dst, noise_mode='const').clamp(-1, 1) + 1) / 2
    img_mix = (G.synthesis(w_mix, noise_mode='const').clamp(-1, 1) + 1) / 2

# Mostrar resultados
titles = ["Rostro base", "Rostro donante", "Rostro mezclado"]
plt.figure(figsize=(10, 3))
for i, (img, t) in enumerate(zip([img_src, img_dst, img_mix], titles)):
    plt.subplot(1, 3, i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.title(t, fontsize=12)
    plt.axis("off")

plt.suptitle(f"Style Mixing - capas de crossover a partir de {crossover}", fontsize=14, y=0.95)
plt.tight_layout(rect=[0, 0, 1, 0.92])
plt.show()

```

- Se generan dos vectores latentes z en el espacio Z , y se proyectan al espacio intermedio W .
- w_{src} : controla el **rostro base** (estructura).
- w_{dst} : controla el **rostro donante** (textura).

```

# Control latente por mezcla de estilos (Style Mixing)

z_src = torch.randn(1, G.z_dim, device=device) # rostro fuente (forma)
z_dst = torch.randn(1, G.z_dim, device=device) # rostro destino (textura)

with torch.no_grad():
    w_src = G.mapping(z_src, None)
    w_dst = G.mapping(z_dst, None)

crossover = 6 # capa donde se combinan los estilos
w_mix = w_src.clone()
w_mix[:, crossover:, :] = w_dst[:, crossover:, :]

with torch.no_grad():
    img_src = (G.synthesis(w_src, noise_mode='const').clamp(-1, 1) + 1) / 2
    img_dst = (G.synthesis(w_dst, noise_mode='const').clamp(-1, 1) + 1) / 2
    img_mix = (G.synthesis(w_mix, noise_mode='const').clamp(-1, 1) + 1) / 2

# Mostrar resultados
titles = ["Rostro base", "Rostro donante", "Rostro mezclado"]
plt.figure(figsize=(10, 3))
for i, (img, t) in enumerate(zip([img_src, img_dst, img_mix], titles)):
    plt.subplot(1, 3, i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.title(t, fontsize=12)
    plt.axis("off")

plt.suptitle(f"Style Mixing - capas de crossover a partir de {crossover}", fontsize=14, y=0.95)
plt.tight_layout(rect=[0, 0, 1, 0.92])
plt.show()

```

- crossover = 6 define la capa a partir de la cual se "mezclan" los estilos.
- Se clona w_src (la base) y luego se reemplazan las **capas posteriores a la 6** por las de w_dst.
- Las **primeras 6 capas** (resoluciones bajas, que determinan forma global y pose) vienen del rostro fuente.
- Las **capas altas** (resoluciones finas, que determinan textura, color, cabello, detalles de piel) vienen del rostro donante.

```

# Control latente por mezcla de estilos (Style Mixing)

z_src = torch.randn(1, G.z_dim, device=device) # rostro fuente (forma)
z_dst = torch.randn(1, G.z_dim, device=device) # rostro destino (textura)

with torch.no_grad():
    w_src = G.mapping(z_src, None)
    w_dst = G.mapping(z_dst, None)

crossover = 6 # capa donde se combinan los estilos
w_mix = w_src.clone()
w_mix[:, crossover:, :] = w_dst[:, crossover:, :]

with torch.no_grad():
    img_src = (G.synthesis(w_src, noise_mode='const').clamp(-1, 1) + 1) / 2
    img_dst = (G.synthesis(w_dst, noise_mode='const').clamp(-1, 1) + 1) / 2
    img_mix = (G.synthesis(w_mix, noise_mode='const').clamp(-1, 1) + 1) / 2

# Mostrar resultados
titles = ["Rostro base", "Rostro donante", "Rostro mezclado"]
plt.figure(figsize=(10, 3))
for i, (img, t) in enumerate(zip([img_src, img_dst, img_mix], titles)):
    plt.subplot(1, 3, i + 1)
    plt.imshow(np.transpose(img[0].cpu().numpy(), (1, 2, 0)))
    plt.title(t, fontsize=12)
    plt.axis("off")

plt.suptitle(f"Style Mixing - capas de crossover a partir de {crossover}", fontsize=14, y=0.95)
plt.tight_layout(rect=[0, 0, 1, 0.92])
plt.show()

```

- Se generan tres imágenes:
 - img_src: rostro original del vector fuente.
 - img_dst: rostro original del vector donante.
 - img_mix: rostro híbrido que mezcla ambos estilos.
- clamp(-1, 1) ajusta los valores de salida a un rango válido y luego se escalan a [0,1].

Control latente por mezcla de estilos (Style Mixing)

Se combinan los estilos de dos códigos latentes —uno que define la estructura global (“forma”) y otro que aporta los detalles locales (“textura”)— a partir de una capa de *crossover* específica.

El rostro conserva la forma general del base, pero adopta la textura, color o estilo del donante.

StyleGAN separa las características visuales por **nivel jerárquico de detalle**:

- Las **capas tempranas** controlan la **geometría global** (forma del rostro, pose, proporciones).
- Las **capas intermedias** controlan rasgos como peinado, facciones y expresión.
- Las **capas tardías** modulan **texturas finas** (piel, color, iluminación, fondo).
- Cuando haces un **crossover** a partir de la **capa 6**, las primeras capas provienen del **rostro base (z_1)** y las últimas del **rostro donante (z_2)**, produciendo una **hibridación estructurada**:

Rostro base



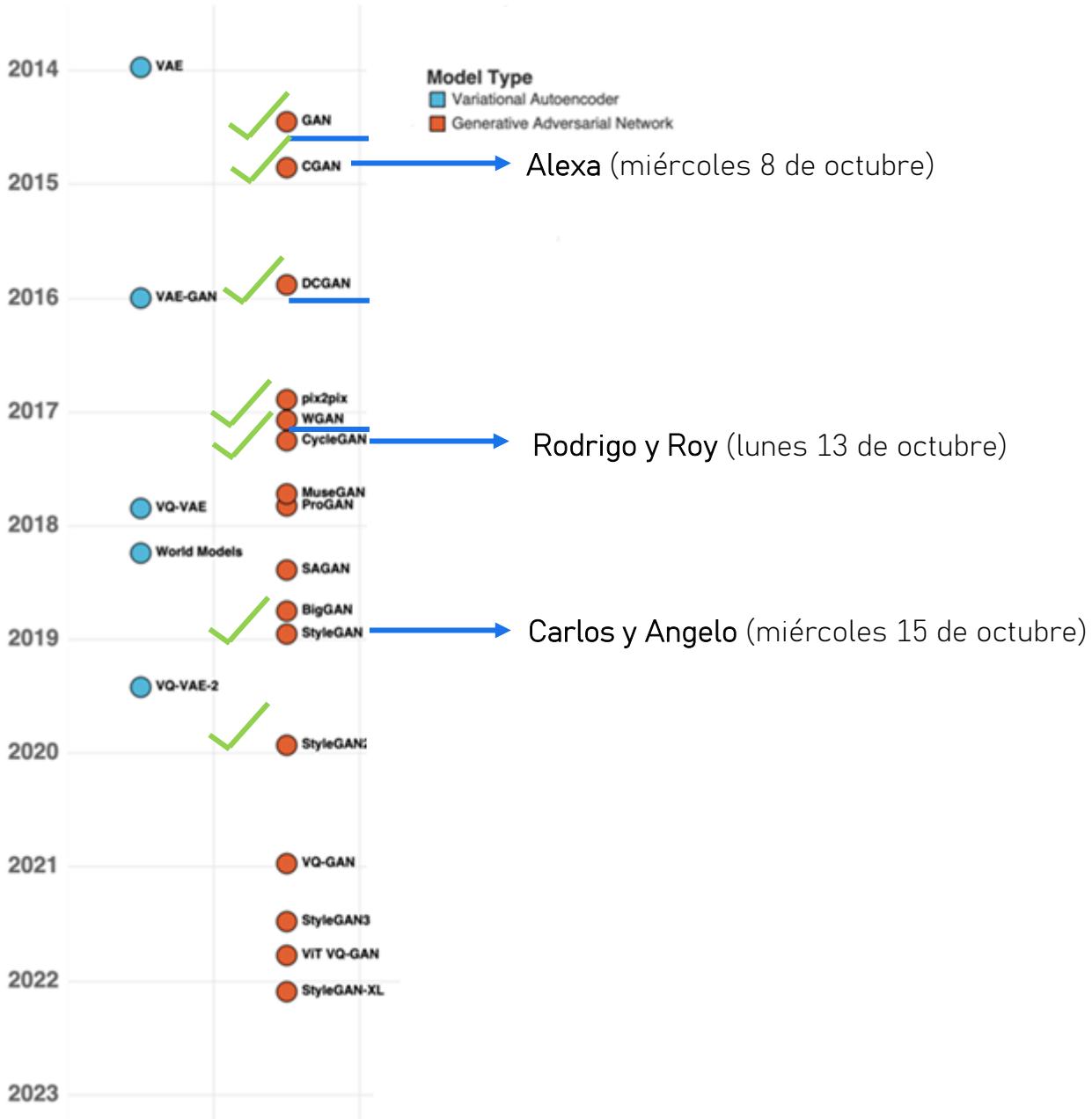
Rostro donante



Rostro mezclado



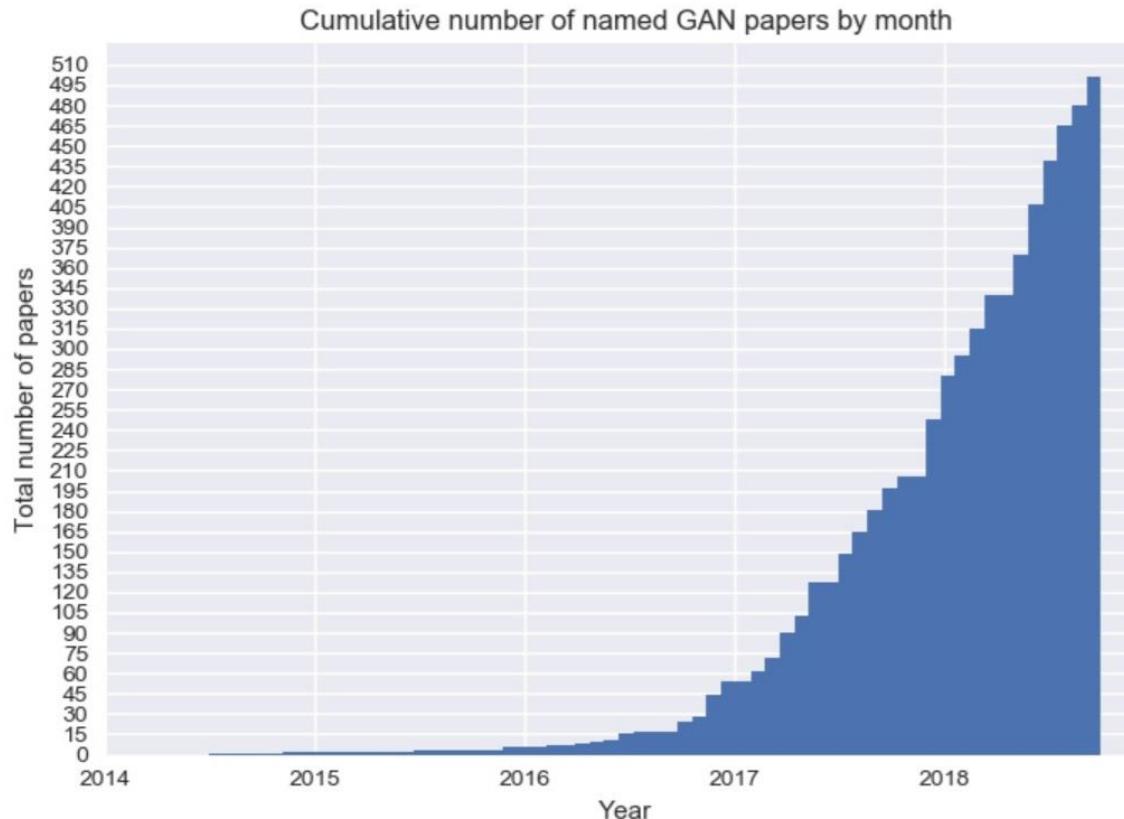
Generative AI Timeline



The GAN Zoo

<https://github.com/hindupuravinash/the-gan-zoo>

Every week, new GAN papers are coming out and it's hard to keep track of them all, not to mention the incredibly creative ways in which researchers are naming these GANs! So, here's a list of what started as a fun activity compiling all named GANs!



Modelos Generativos Profundos

Modelos basados en energía

Clase 16

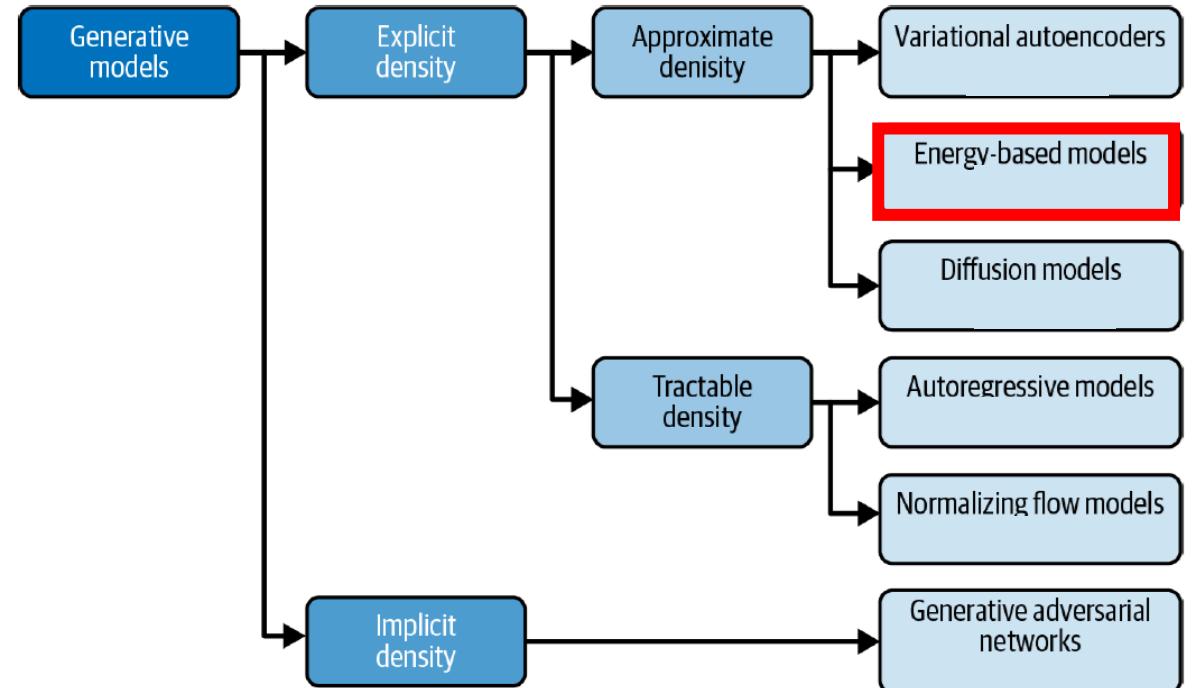
Dra. Wendy Aguilar

UN ENFOQUE DESDE LA
CREATIVIDAD
COMPUTACIONAL



¿Qué son los Modelos Basados en Energía (EBMs)?

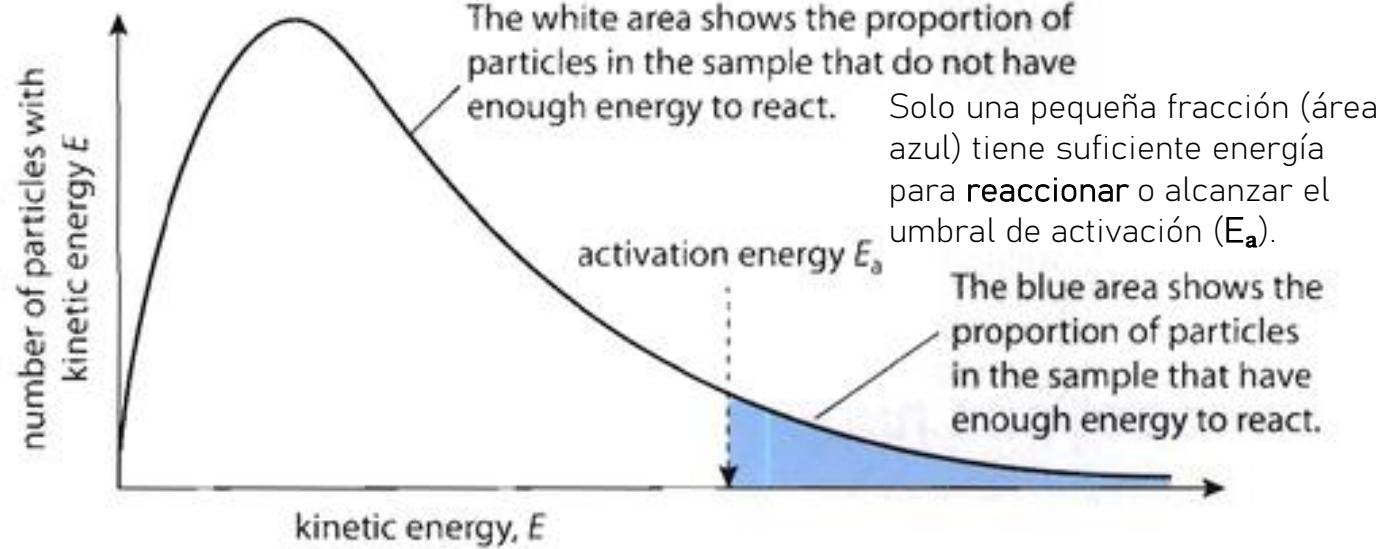
- Son una amplia familia de **modelos generativos**.
- Se inspiran en una idea clave de la **física estadística**:
 - la probabilidad de un evento puede expresarse en función de su **energía**.
- Esta probabilidad se describe mediante la **distribución de Boltzmann**, que transforma una energía real en un valor entre 0 y 1.



Origen del concepto

- La distribución de Boltzmann fue formulada en 1868 por Ludwig Boltzmann.
- Boltzmann la utilizó para describir cómo se distribuyen las partículas de un gas cuando el sistema está en **equilibrio térmico**.
- En los EBMs, se toma esa misma idea:
 - los estados más probables del sistema son aquellos con **menor energía**.

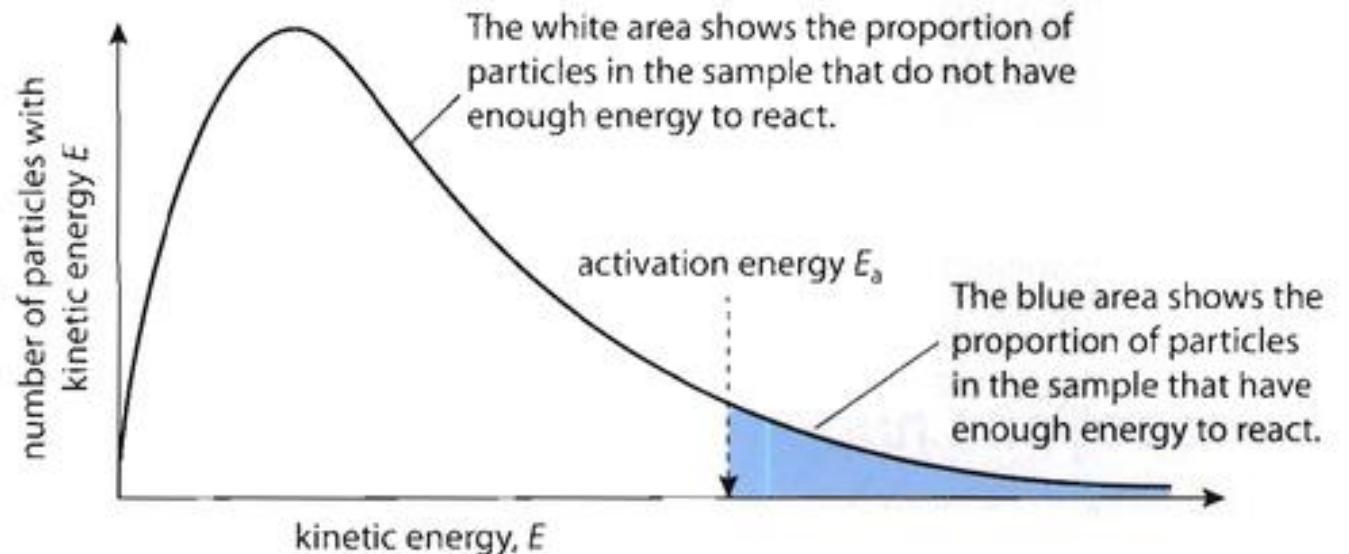
El eje y muestra el número de partículas que poseen esa energía.



El eje x representa la **energía cinética (E)** de las partículas.

Aplicación en aprendizaje profundo

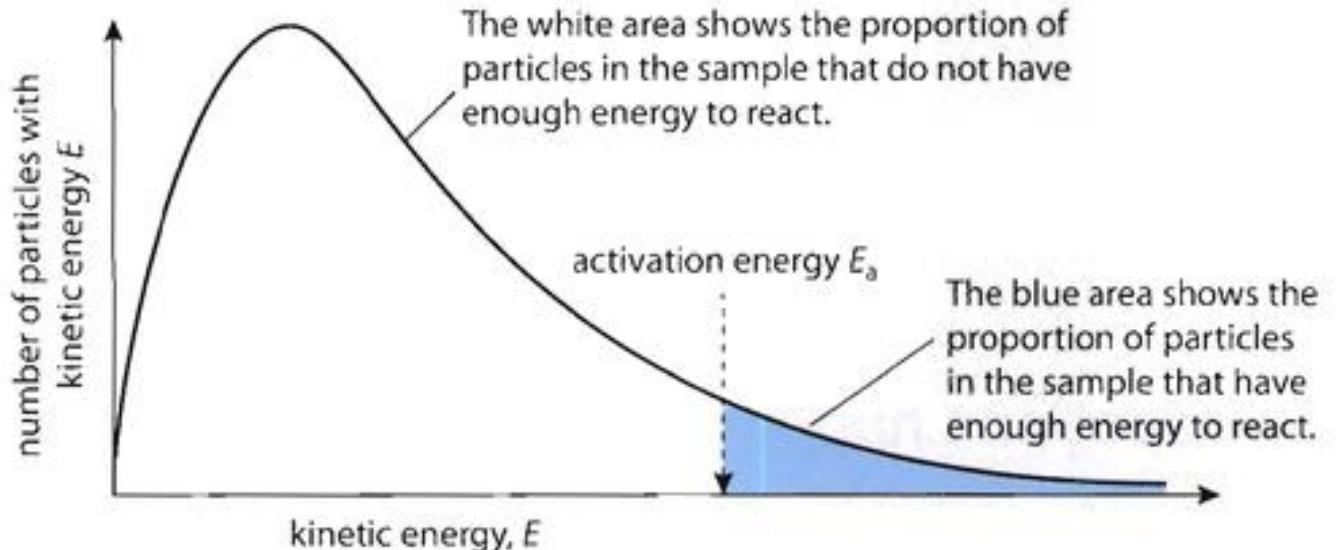
- En aprendizaje profundo, esta idea se usa para construir modelos que **aprenden una función de energía** sobre los datos.
- Esa función asigna **baja energía** a ejemplos reales y **alta energía** a ejemplos falsos o improbables.
- De este modo, el modelo **aprende la estructura del conjunto de datos** sin necesidad de que se especifique explícitamente una probabilidad.



- En lugar de partículas, tenemos **estados posibles de los datos**.
- Cada estado tiene una **energía** asignada por el modelo.
- Los estados con **baja energía** son los más **probables o realistas** (análogos a partículas que pueden "reaccionar").
- Los estados con **alta energía** son poco probables o "irreales".

Aplicación en aprendizaje profundo

- En aprendizaje profundo, esta idea se usa para construir modelos que **aprenden una función de energía** sobre los datos.
- Esa función asigna **baja energía** a ejemplos reales y **alta energía** a ejemplos falsos o improbables.
- De este modo, el modelo **aprende la estructura del conjunto de datos** sin necesidad de que se especifique explícitamente una probabilidad.



- En lugar de partículas, tenemos **estados posibles** de los datos.
- Cada estado tiene una **energía** asignada por el modelo.
- Los estados con **baja energía** son los más **probables** o **realistas** (análogos a partículas que pueden "reaccionar").
- Los estados con **alta energía** son poco probables o "irreales".

Así, el modelo **aprende una superficie de energía** sobre el espacio de los datos, donde las regiones más bajas corresponden a ejemplos coherentes con el conjunto de entrenamiento.

Modelos Basados en Energía

- Los Modelos Basados en Energía (EBMs) buscan modelar la distribución real que genera los datos utilizando una distribución de Boltzmann.
- En esta distribución, $E(x)$ representa la función de energía (también llamada *score*) de una observación x .

Distribución de Boltzmann

$$p(x) = \frac{e^{-E(x)}}{\sum_{\hat{x} \in X} e^{-E(\hat{x})}}$$

- Cuanto **menor** sea la energía $E(x)$, **mayor** será la probabilidad $p(x)$.
- En la práctica, entrenamos una **red neuronal** $E(x)$ para que:
 - Asigne **baja energía** (o **bajo score**) a las observaciones **reales** $\rightarrow p(x) \approx 1$.
 - Asigne **alta energía** a las observaciones **falsas** $\rightarrow p(x) \approx 0$.

Desafíos principales



$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int_{\widehat{\mathbf{x}} \in \mathbf{X}} e^{-E(\widehat{\mathbf{x}})}}$$

1. ¿Cómo generamos nuevas observaciones?

- El modelo puede calcular una energía dada una observación, pero **no sabe cómo crear una observación con baja energía** (es decir, un ejemplo plausible).

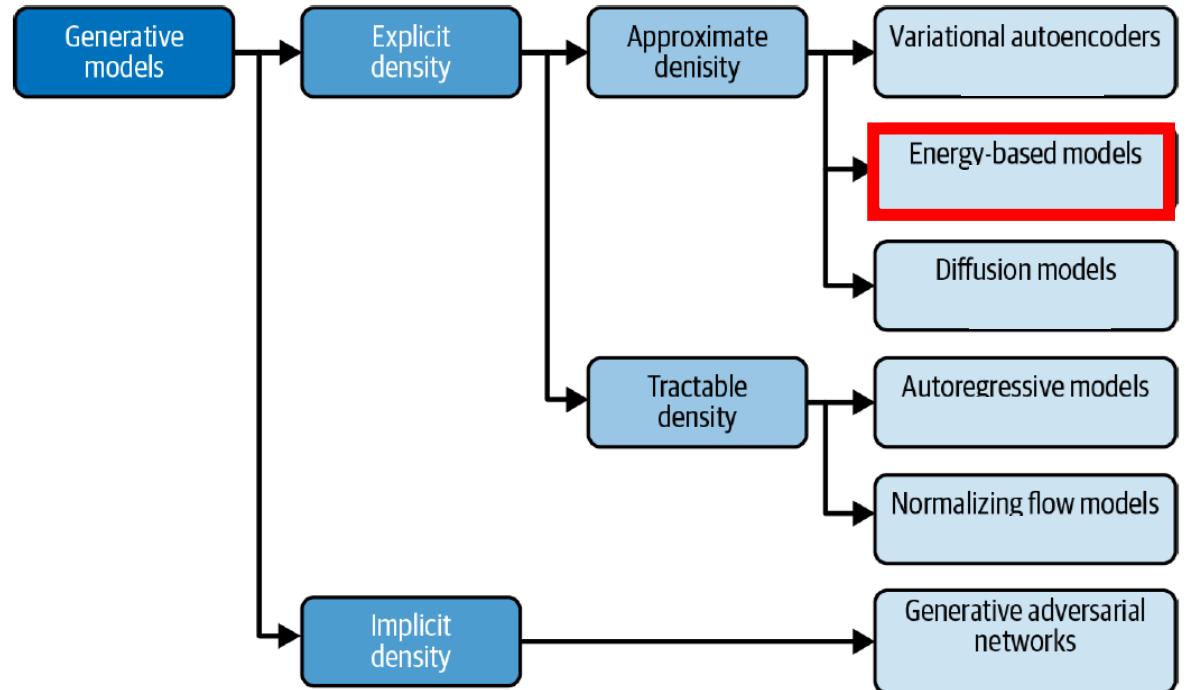
2. El denominador intratable (la integral de normalización)

- La integral $\int e^{-E(x')} dx'$ es imposible de calcular.
No podemos calcularla porque implicaría integrar sobre todos los posibles estados x' .
- Sin ese valor, **no podemos aplicar estimación por máxima verosimilitud**, porque no tenemos una distribución de probabilidad válida.

La idea clave

Los EBMs evitan calcular la integral directamente usando técnicas de aproximación.

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int_{\hat{\mathbf{x}} \in \mathbf{X}} e^{-E(\hat{\mathbf{x}})}}$$



Soluciones prácticas

Para resolver los dos problemas anteriores, se utilizan dos técnicas complementarias:

- Divergencia Contrastiva (Contrastive Divergence) → para **entrenar** el modelo sin calcular la integral.
- Dinámica de Langevin (Langevin Dynamics) → para **muestrar** observaciones con baja energía (simular ejemplos plausibles).

Estas ideas fueron introducidas:

2019

Implicit Generation and Modeling with Energy-Based Models

Yilun Du *
MIT CSAIL

Igor Mordatch
Google Brain

Abstract

Energy based models (EBMs) are appealing due to their generality and simplicity in likelihood modeling, but have been traditionally difficult to train. We present techniques to scale MCMC based EBM training on continuous neural networks, and we show its success on the high-dimensional data domains of ImageNet32x32, ImageNet128x128, CIFAR-10, and robotic hand trajectories, achieving better samples than other likelihood models and nearing the performance of contemporary GAN approaches, while covering all modes of the data. We highlight some unique capabilities of implicit generation such as compositionality and corrupt image reconstruction and inpainting. Finally, we show that EBMs are useful models across a wide variety of tasks, achieving state-of-the-art out-of-distribution classification, adversarially robust classification, state-of-the-art continual online class learning, and coherent long term predicted trajectory rollouts.

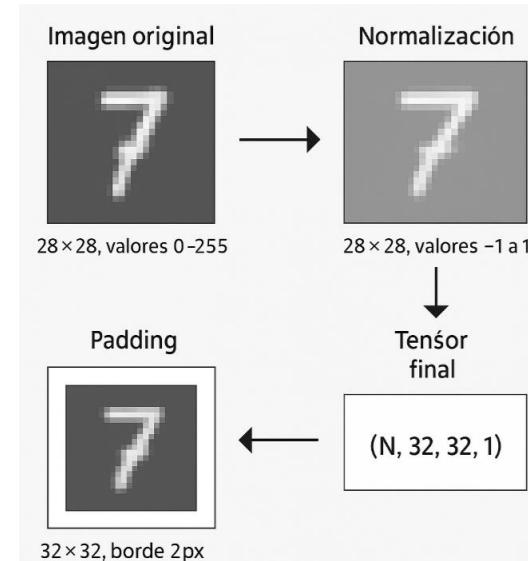
Manos a la obra

- Prepararemos un **dataset** y diseñaremos una **red neuronal simple** que representará nuestra función de energía real-valuada $E(x)$.



MNIST

```
from tensorflow.keras import datasets  
(x_train, _), (x_test, _) = datasets.mnist.load_data()  
  
def preprocess(imgs):  
    imgs = (imgs.astype("float32") - 127.5) / 127.5  
    imgs = np.pad(imgs , ((0,0), (2,2), (2,2)), constant_values= -1.0)  
    imgs = np.expand_dims(imgs, -1)  
    return imgs  
  
x_train = preprocess(x_train)  
x_test = preprocess(x_test)  
x_train = tf.data.Dataset.from_tensor_slices(x_train).batch(128)  
x_test = tf.data.Dataset.from_tensor_slices(x_test).batch(128)
```



Función de energía

- La función de energía $E_\theta(x)$ es una **red neuronal** con parámetros θ que transforma una imagen de entrada x en un valor escalar.
- Ese valor escalar representa **la energía asociada a la imagen**:
 - Baja energía → imagen probable (realista)
 - Alta energía → imagen improbable (no realista)

Arquitectura de la red $E_{\theta}(x)$

- Está compuesta por una **serie de capas convolucionales (Conv2D)** apiladas.
- Cada capa **reduce el tamaño espacial** de la imagen pero **incrementa el número de canales**.
- La **última capa** es una unidad totalmente conectada (Dense) con **activación lineal**, lo que permite producir **valores en todo el rango real ($-\infty, \infty$)**.

```
ebm_input = layers.Input(shape=(32, 32, 1))
x = layers.Conv2D(
    16, kernel_size=5, strides=2, padding="same", activation = activations.swish
)(ebm_input) ①
x = layers.Conv2D(
    32, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Flatten()(x)
x = layers.Dense(64, activation = activations.swish)(x)
ebm_output = layers.Dense(1)(x) ②
model = models.Model(ebm_input, ebm_output)| ③
```

Arquitectura de la red $E_{\theta}(x)$

- Está compuesta por una **serie de capas convolucionales** (Conv2D) apiladas.
- Cada capa **reduce el tamaño espacial** de la imagen pero **incrementa el número de canales**.
- La **última capa** es una unidad totalmente conectada (Dense) con **activación lineal**, lo que permite producir **valores en todo el rango real** ($-\infty, \infty$).

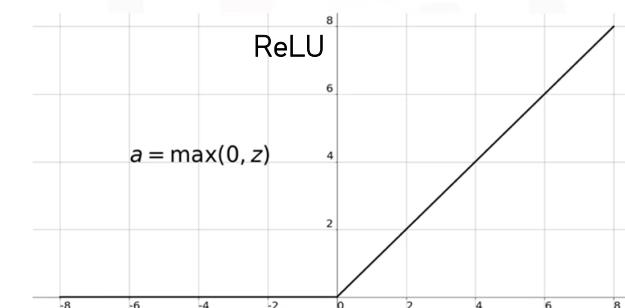
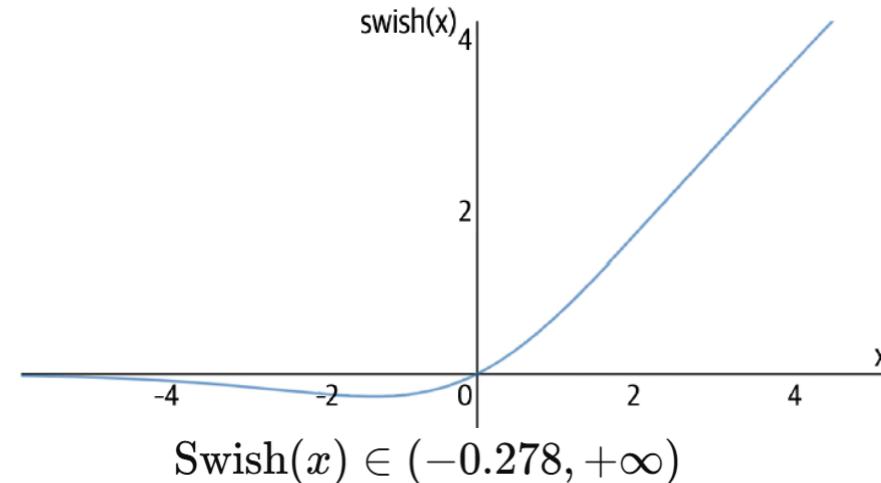
```
ebm_input = layers.Input(shape=(32, 32, 1))
x = layers.Conv2D(
    16, kernel_size=5, strides=2, padding="same", activation = activations.swish
)(ebm_input) ①
x = layers.Conv2D(
    32, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Flatten()(x)
x = layers.Dense(64, activation = activations.swish)(x) ②
ebm_output = layers.Dense(1)(x) ③
model = models.Model(ebm_input, ebm_output)| ③
```

Función de activación Swish

Introducida por Google en 2017, la función *Swish* se define como:

$$\text{swish}(x) = x \cdot \text{sigmoid}(x) = \frac{x}{1 + e^{-x}}$$

- Es una alternativa a ReLU, con la diferencia de que:
 - Es **suave y continua** → evita discontinuidades.
 - Atenúa el problema del gradiente que desaparece, crucial en modelos basados en energía.
- Su forma es parecida a ReLU, pero con una **transición gradual** entre valores negativos y positivos.

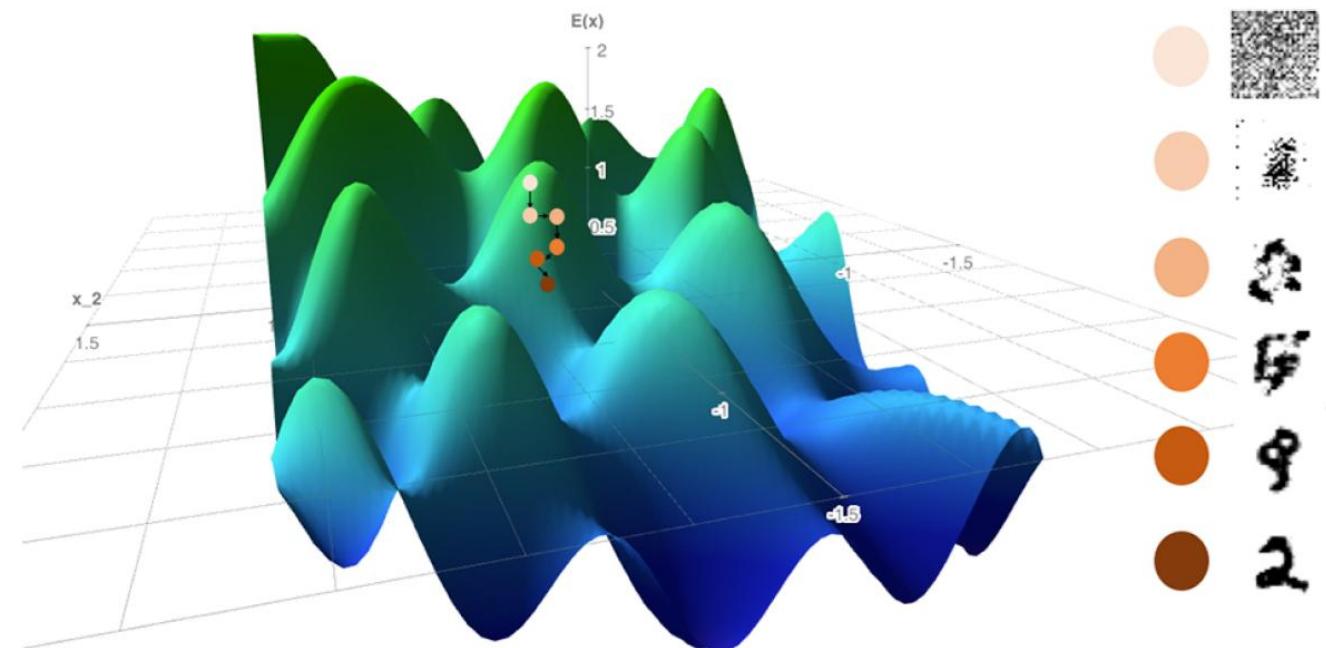


Muestreo usando Dinámica de Langevin

La función de energía solo devuelve un valor escalar (una puntuación) para una imagen dada.

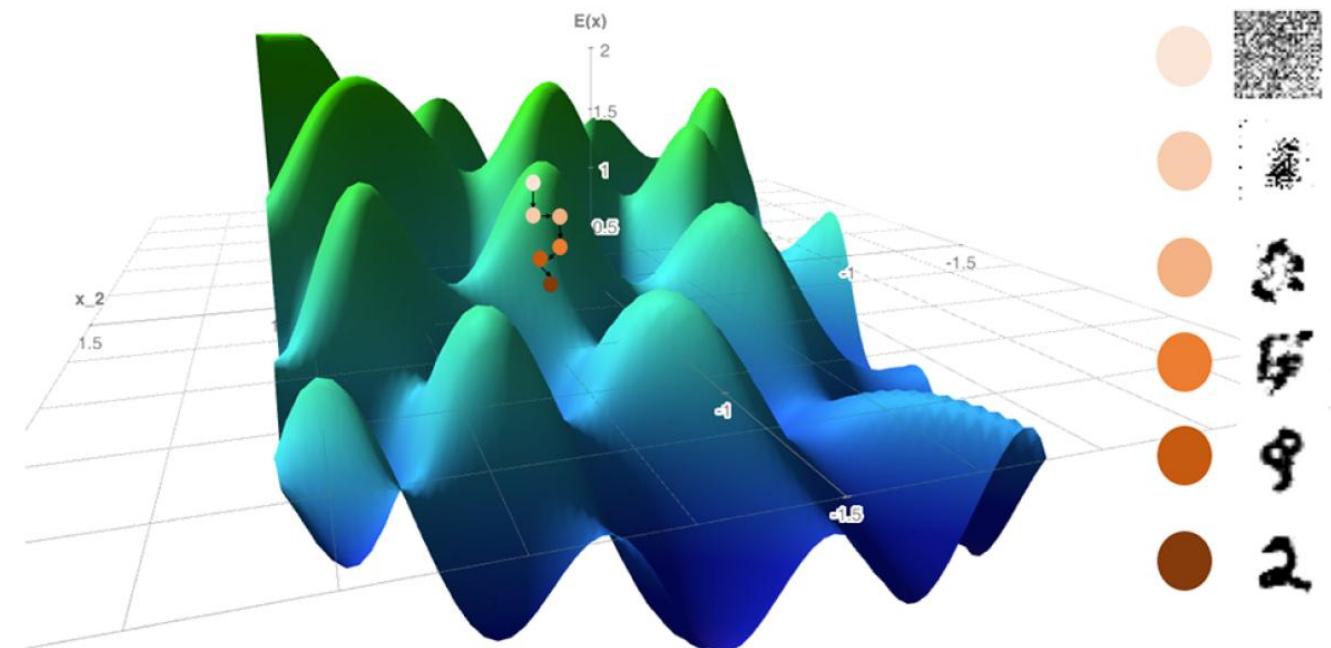
Entonces, ¿cómo podemos usar esta función para **generar nuevas muestras** que tengan una **baja energía**?

- Usaremos una técnica llamada **Dinámica de Langevin**, que aprovecha el hecho de que podemos **calcular el gradiente de la función de energía con respecto a su entrada**.
- Si partimos de un punto aleatorio en el espacio de muestras y damos pequeños pasos en la **dirección opuesta al gradiente**, reduciremos gradualmente la energía.
- Si la red neuronal está bien entrenada, el ruido inicial se transformará **progresivamente en una imagen realista**, similar a las del conjunto de entrenamiento.



Dinámica Estocástica de Gradiente de Langevin

- Es importante añadir una **pequeña cantidad de ruido aleatorio** en cada paso; de lo contrario, podríamos **caer en un mínimo local**.
- Por eso, esta técnica se llama **Dinámica Estocástica de Gradiente de Langevin (SGLD)**.
- El proceso se asemeja a un **descenso ruidoso** por un paisaje tridimensional donde la altura representa la energía $E(x)$.
- En el caso de MNIST, el “paisaje” tiene **1024 dimensiones** (una por cada píxel), pero la idea es la misma: moverse hacia regiones de baja energía.



Diferencia con el gradiente descendente clásico

Diferencia con el gradiente descendente clásico

Aspecto	Entrenamiento de red neuronal	Dinámica de Langevin
Qué se actualiza	Los pesos (θ) de la red	La entrada (x) de la red
Objetivo	Minimizar la pérdida (loss)	Minimizar la energía ($E(x)$)
Gradiente de	$\nabla_{\theta} L(\theta)$	$\nabla_x E_{\theta}(x)$
Resultado	Red entrenada	Imagen generada

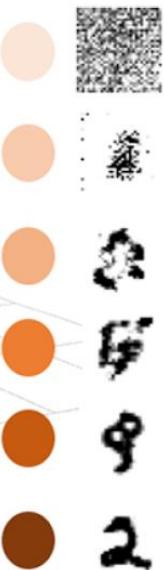
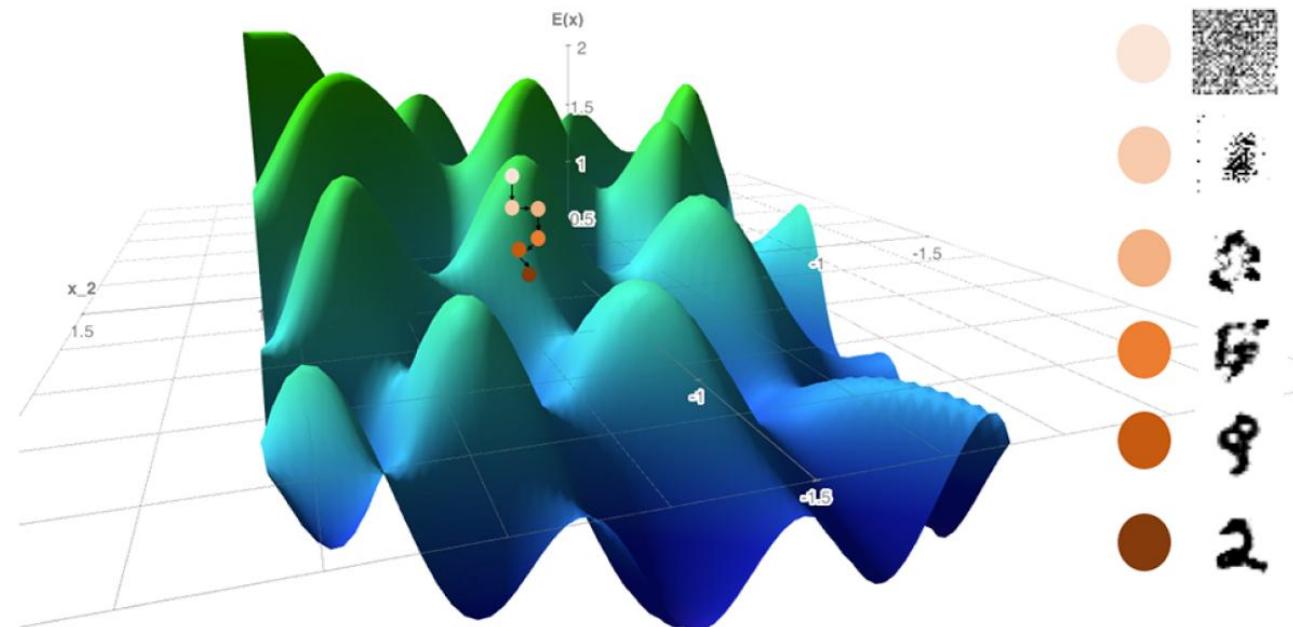
Ecuación formal de Langevin Dynamics

$$x_k = x_{k-1} - \eta \nabla_x E_\theta(x_{k-1}) + \omega$$

donde:

- $\eta \rightarrow$ tamaño del paso (*step size*, hiperparámetro)
- $\omega \sim \mathcal{N}(0, \sigma^2) \rightarrow$ ruido gaussiano añadido en cada paso
- $x_0 \sim \mathcal{U}(-1, 1) \rightarrow$ punto inicial, muestra aleatoria uniforme

Si η es demasiado grande \rightarrow saltamos los mínimos.
Si η es demasiado pequeño \rightarrow convergencia muy lenta.

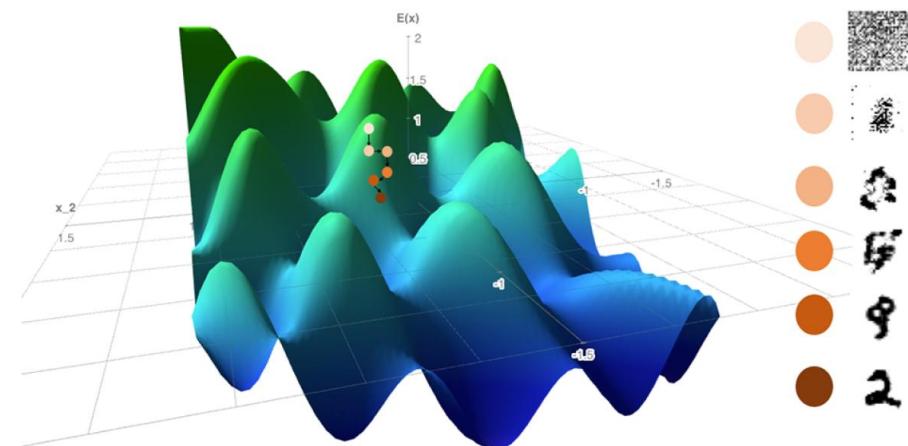


Interpretación conceptual

- Cada paso reduce ligeramente la energía del ejemplo actual.
- El ruido mantiene la exploración del espacio, evitando estancarse en mínimos locales.
- Tras suficientes iteraciones, el ruido inicial se transforma gradualmente en una imagen coherente del dominio aprendido (por ejemplo, un dígito de MNIST).



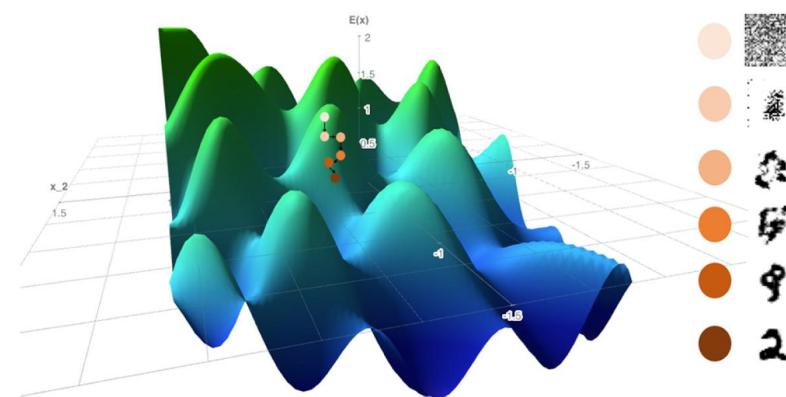
En los **EBCs**, el modelo no “genera” directamente las imágenes,
sino que **las esculpe gradualmente** a partir de ruido,
moviéndose
sobre el paisaje de energía que aprendió durante el
entrenamiento.



```

def generate_samples(model, inp_imgs, steps, step_size, noise):
    imgs_per_step = []
    for _ in range(steps): ❶ → Iterar durante el número de pasos indicado.
        inp_imgs += tf.random.normal(inp_imgs.shape, mean = 0, stddev = noise) ❷ → Añadir una pequeña cantidad de ruido a la imagen.
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
        with tf.GradientTape() as tape:
            tape.watch(inp_imgs)
            out_score = -model(inp_imgs) ❸ → Pasar la imagen por el modelo para obtener el puntaje de energía.
            grads = tape.gradient(out_score, inp_imgs) ❹ → Calcular el gradiente de la salida con respecto a la entrada.
            grads = tf.clip_by_value(grads, -0.03, 0.03)
            inp_imgs += -step_size * grads ❺ → Agregar una pequeña cantidad del gradiente a la imagen de entrada.
            inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
    return inp_imgs

```



Soluciones prácticas

Para resolver los dos problemas anteriores, se utilizan dos técnicas complementarias:

- Divergencia Contrastiva (Contrastive Divergence) → para **entrenar** el modelo sin calcular la integral.
- Dinámica de Langevin (Langevin Dynamics) → para **muestrear** observaciones con baja energía (simular ejemplos plausibles).

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int_{\hat{\mathbf{x}} \in \mathcal{X}} e^{-E(\hat{\mathbf{x}})}}$$

Estas ideas fueron introducidas:

2019

Implicit Generation and Modeling with Energy-Based Models

Yilun Du *
MIT CSAIL

Igor Mordatch
Google Brain

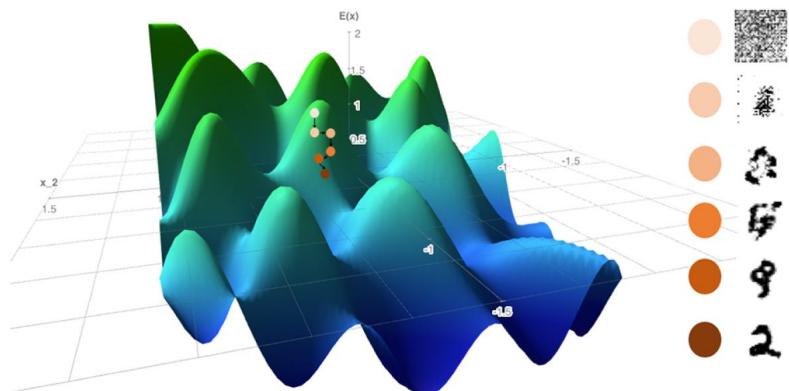
Abstract

Energy based models (EBMs) are appealing due to their generality and simplicity in likelihood modeling, but have been traditionally difficult to train. We present techniques to scale MCMC based EBM training on continuous neural networks, and we show its success on the high-dimensional data domains of ImageNet32x32, ImageNet128x128, CIFAR-10, and robotic hand trajectories, achieving better samples than other likelihood models and nearing the performance of contemporary GAN approaches, while covering all modes of the data. We highlight some unique capabilities of implicit generation such as compositionality and corrupt image reconstruction and inpainting. Finally, we show that EBMs are useful models across a wide variety of tasks, achieving state-of-the-art out-of-distribution classification, adversarially robust classification, state-of-the-art continual online class learning, and coherent long term predicted trajectory rollouts.

Divergencia Contrastiva en Modelos Basados en Energía

¿Qué queremos lograr?

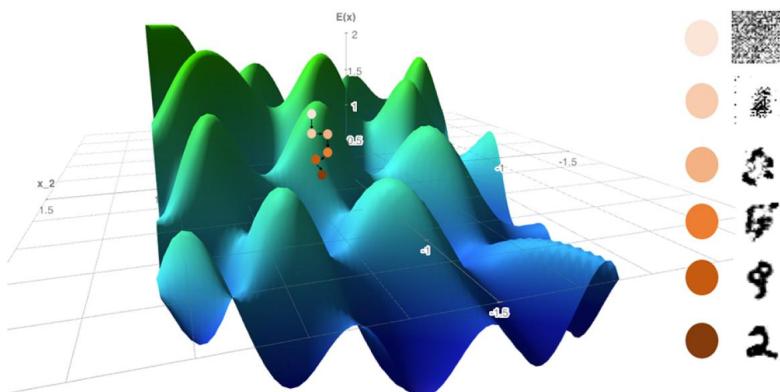
- Enseñar al modelo a **asignar baja energía** a los datos **reales** y **alta energía** a los datos **generados o falsos**.
- En otras palabras:
- “Que los ejemplos **reales** vivan en los **valles** del paisaje de energía y los **falsos** en las **colinas**.”



Divergencia Contrastiva en Modelos Basados en Energía

¿Qué queremos lograr?

- Enseñar al modelo a **asignar baja energía a los datos reales y alta energía a los datos generados o falsos.**
- En otras palabras:
- “Que los ejemplos **reales vivan en los valles del paisaje de energía y los falsos en las colinas.**”



Problema de fondo

- En teoría, deberíamos **maximizar la probabilidad de los datos reales:**

$$\mathcal{L} = -\mathbb{E}_{x \sim \text{data}} [\log p_\theta(x)]$$

- Pero $p_\theta(x)$ tiene una **integral de normalización intratable:**

$$p_\theta(x) = \frac{e^{-E_\theta(x)}}{\int e^{-E_\theta(x')} dx'}$$

- No podemos calcular esa integral → no podemos usar máxima verosimilitud directa.

Divergencia Contrastiva en Modelos Basados en Energía



Solución

- Propuesta por Geoffrey Hinton (2002).
- Permite entrenar **modelos no normalizados** como los Modelos Basados en Energía (EBMs).
- Aproxima el gradiente del aprendizaje mediante dos expectativas:

Expectativa (promedio) del gradiente de la energía para las **muestras reales** del conjunto de datos
Empuja a disminuir la energía de los datos reales.

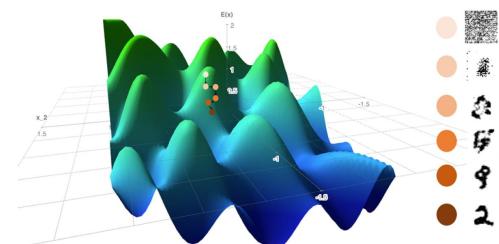
$$\nabla_{\theta} \mathcal{L} = \underbrace{\mathbb{E}_{x \sim \text{data}} [\nabla_{\theta} E_{\theta}(x)]}_{\text{Gradiente de la función de pérdida con respecto a los parámetros } \theta \text{ del modelo}} - \underbrace{\mathbb{E}_{x \sim \text{model}} [\nabla_{\theta} E_{\theta}(x)]}_{\text{Expectativa (promedio) del gradiente de energía para las muestras generadas por el modelo (las falsas).}}$$

Indica cómo deben ajustarse los pesos de la red para mejorar el modelo.

Estas muestras suelen obtenerse mediante **Dinámica de Langevin**.
Este término **empuja a aumentar la energía** de los ejemplos falsos o improbables.

- El modelo aprende por **contraste** → de ahí el nombre “divergencia contrastiva”.

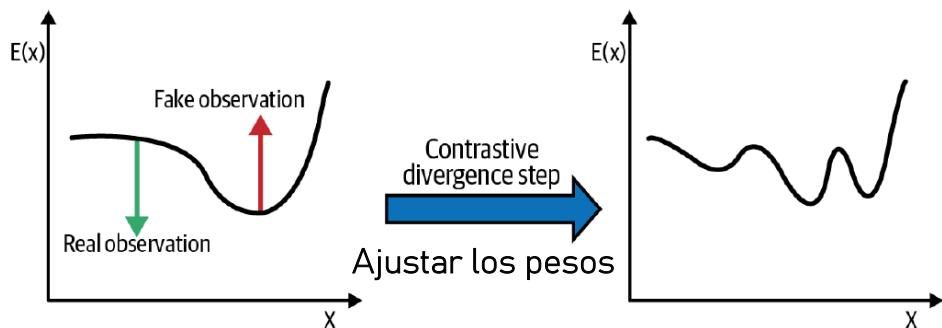
- Datos reales → energía baja
- Datos falsos → energía alta



Entrenamiento con Divergencia Contrastiva

Procedimiento general

1. Tomamos ejemplos reales del conjunto de entrenamiento.
2. Generamos ejemplos falsos (o "negativos") desplazando ruido mediante muestreo de Langevin.
3. Comparamos las energías $E_\theta(x)$ para ambos conjuntos.
4. Actualizamos los pesos θ para aumentar ese contraste:
 - Disminuir $E_\theta(x)$ de los reales.
 - Aumentar $E_\theta(x)$ de los falsos.

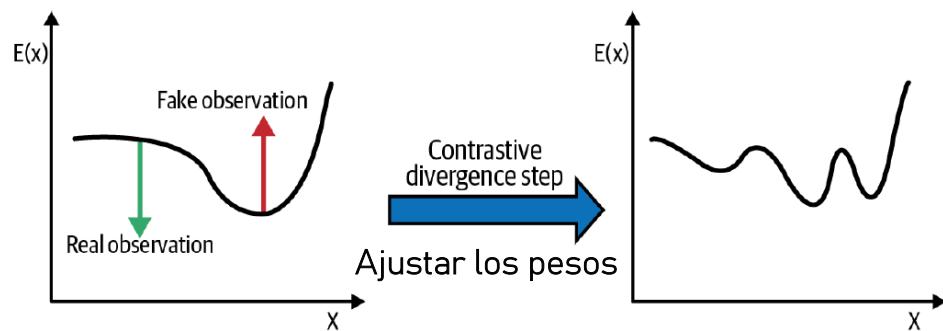


Ahora el paisaje tiene más estructura: aparecen valles y colinas más definidos.

Entrenamiento con Divergencia Contrastiva

Procedimiento general

1. Tomamos **ejemplos reales** del conjunto de entrenamiento.
2. Generamos **ejemplos falsos** (o "negativos") desplazando ruido mediante muestreo de Langevin.
3. Comparamos las energías $E_\theta(x)$ para ambos conjuntos.
4. Actualizamos los pesos θ para aumentar ese contraste:
 - Disminuir $E_\theta(x)$ de los reales.
 - Aumentar $E_\theta(x)$ de los falsos.



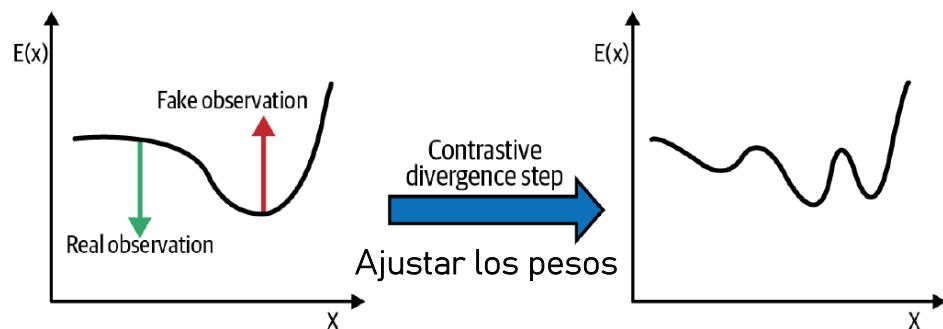
Buffer de muestreo

- Se mantiene un **búfer de observaciones previas** para:
 - Evitar empezar cada iteración desde ruido puro, usando ejemplos ya cercanos a regiones de baja energía.
 - Acelerar la **convergencia** del muestreo.
 - Aumenta la **estabilidad** del entrenamiento.
 - Actúa como **memoria** del modelo, guardando conocimiento implícito acerca de las regiones del espacio de datos que ya fueron exploradas.
 - Combina exploración y explotación:
 - ~5% de muestras → ruido nuevo (exploración).
 - ~95% de muestras → del buffer previo (explotación).
 - Este equilibrio permite aprender sin atascarse en mínimos locales.
- Se mantiene un límite fijo (por ejemplo, 8,192 muestras) para evitar que crezca indefinidamente y mantener diversidad.

Entrenamiento con Divergencia Contrastiva

Procedimiento general

1. Tomamos **ejemplos reales** del conjunto de entrenamiento.
2. Generamos **ejemplos falsos** (o “negativos”) desplazando ruido mediante muestreo de Langevin.
3. Comparamos las energías $E_\theta(x)$ para ambos conjuntos.
4. Actualizamos los pesos θ para aumentar ese contraste:
 - Disminuir $E_\theta(x)$ de los reales.
 - Aumentar $E_\theta(x)$ de los falsos.



Buffer de muestreo

1. Inicialización:
 - Al inicio, se llena con **ruido aleatorio** (por ejemplo, valores uniformes entre -1 y 1).
2. Actualización de las muestras:
 - Durante el entrenamiento
En cada iteración:
 - Se selecciona una **submuestra** del buffer (por ejemplo, un minibatch de 128 imágenes).
 - Se **ajustan** con *Langevin Dynamics* (para que se acerquen a regiones de baja energía).
 - Luego se **devuelven al buffer** con sus versiones actualizadas.

Así, el buffer **evoluciona con el entrenamiento**, conteniendo ejemplos falsos cada vez más realistas.

```
class Buffer:
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.examples = [
            tf.random.uniform(shape = (1, 32, 32, 1)) * 2 - 1
            for _ in range(128)
        ] ❶ → Crea 128 imágenes iniciales (32x32x1) de ruido uniforme entre -1 y 1.
        Estas representan las primeras "muestras falsas" del modelo.

    def sample_new_exmps(self, steps, step_size, noise):
        n_new = np.random.binomial(128, 0.05) ❷ → Calcula cuántas de las 128 muestras del nuevo lote serán ruido
        rand_imgs = (
            tf.random.uniform((n_new, 32, 32, 1)) * 2 - 1
        )
        old_imgs = tf.concat(
            random.choices(self.examples, k=128-n_new), axis=0
        ) ❸ → El 95% restante se tomarán del buffer existente.

        inp_imgs = tf.concat([rand_imgs, old_imgs], axis=0)
        inp_imgs = generate_samples(
            self.model, inp_imgs, steps=steps, step_size=step_size, noise = noise
        ) ❹ → Las observaciones se concatenan y se procesan
        mediante el muestreador de Langevin.

        self.examples = tf.split(inp_imgs, 128, axis = 0) + self.examples ❺ → Este proceso ajusta las imágenes moviéndolas hacia
        self.examples = self.examples[:8192] → regiones de baja energía,
        es decir, las hace más parecidas a ejemplos reales.

    return inp_imgs
```

Divide las nuevas imágenes refinadas y las añade al buffer existente.
Mantiene un **máximo de 8,192 muestras** (descarta las más antiguas si se supera el límite).

```

class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        ) ❶
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        ) ❷
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)

```

Encapsula todo el proceso de entrenamiento de un modelo basado en energía, incluyendo:

- su modelo interno,
- buffer de muestreo,
- métricas y
- paso de entrenamiento (train_step)

```
class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1 → Es un factor de regularización, que evita que los valores de energía crezcan
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        ) ❶
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        ) ❷
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
```

```

class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")
    }

@property
def metrics(self):
    return [
        self.loss_metric,
        self.reg_loss_metric,
        self.cdiv_loss_metric,
        self.real_out_metric,
        self.fake_out_metric
    ]

def train_step(self, real_imgs):
    real_imgs += tf.random.normal(
        shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
    ) ❶
    real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
    fake_imgs = self.buffer.sample_new_exmps(
        steps=60, step_size=10, noise = 0.005
    ) ❷
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)

```

Define métricas para monitorear el entrenamiento:

- `loss_metric`: pérdida total promedio.
- `reg_loss_metric`: término de regularización (estabilidad numérica).
- `cdiv_loss_metric`: pérdida por divergencia contrastiva (principal).
- `real_out_metric`: energía promedio de las imágenes reales.
- `fake_out_metric`: energía promedio de las imágenes falsas.

```
class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        ) ❶
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        ) ❷
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
```

Informa a Keras qué métricas debe reiniciar y registrar automáticamente durante el entrenamiento.

```

class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

@property
def metrics(self):
    return [
        self.loss_metric,
        self.reg_loss_metric,
        self.cdiv_loss_metric,
        self.real_out_metric,
        self.fake_out_metric
    ]

```

→ Lo que el modelo hace en cada iteración de entrenamiento

```

def train_step(self, real_imgs): → Recibe un lote de imágenes reales del dataset.
    real_imgs += tf.random.normal(
        shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
    ) ❶ → Les añade ruido gaussiano leve (desviación 0.005) para:
        • Aumentar la robustez del modelo.
        • Evitar sobreajuste.
        • Facilitar que el modelo aprenda una superficie de energía más suave.

    real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0) → Recorta los valores de los píxeles al rango [-1,1].
    fake_imgs = self.buffer.sample_new_exmps(
        steps=60, step_size=10, noise = 0.005
    ) ❷ Para mantener la coherencia con el preprocesamiento aplicado a las
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)

```

```

class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        ) ❶
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        ) ❷
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)

```

Uso el **buffer** para generar un nuevo lote de **imágenes falsas**:

- Toma parte del buffer anterior y parte de ruido nuevo (~5%).
- Ejecuta 60 pasos de **Dinámica de Langevin** con tamaño de paso 10.
- Devuelve imágenes refinadas que el modelo considera plausibles, pero que siguen siendo "falsas".

* Estas serán las muestras negativas usadas en la Divergencia Contrastiva.

```

class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        ) ❶
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        ) ❷
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)

```

- Junta ambos conjuntos (reales y falsas) en un solo tensor.
- Así puede calcular simultáneamente las energías $E_\theta(x)$ de ambos tipos de imágenes.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ❸ →
        cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
            real_out, axis = 0
        ) ❹
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ❺
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ❻
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

def test_step(self, real_imgs): ❼
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```

- Pasa el lote combinado de imágenes reales y falsas (`inp_imgs`) por el modelo de energía.
- El modelo devuelve **energías** para todas las imágenes.
- Luego divide esas salidas en dos mitades:
 - `real_out`: energías de las imágenes reales.
 - `fake_out`: energías de las imágenes falsas.
- *Cada imagen recibe un valor escalar $E_\theta(x)$.*

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ③
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ④ →
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ⑤
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ⑥
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

def test_step(self, real_imgs): ⑦
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```

Calcula la diferencia promedio entre las energías falsas y reales:

$$\mathcal{L}_{\text{CD}} = \mathbb{E}[E_{\text{fake}}] - \mathbb{E}[E_{\text{real}}]$$

Este es el núcleo de la Divergencia Contrastiva.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ③
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ④
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ⑤ →
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ⑥
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

```

```

def test_step(self, real_imgs): ⑦
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```

- Añade un **término de regularización** para evitar que las energías crezcan sin límite.
- Penaliza energías muy grandes (positivas o negativas).
- self.alpha (0.1) controla cuánto pesa este término.
- Esto suaviza el paisaje de energía y estabiliza el entrenamiento.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ❸
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ❹
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ❺
    loss = reg_loss + cdiv_loss → La pérdida total combina ambos componentes:  $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{CD}} + \alpha \cdot \mathcal{L}_{\text{reg}}$ 
grads = training_tape.gradient(loss, self.model.trainable_variables) ❻
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

def test_step(self, real_imgs): ❼
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ③
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ④
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ⑤
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ⑥
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

def test_step(self, real_imgs): ⑦
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```

- Calcula cómo cambia la pérdida con respecto a los pesos del modelo.
- Luego **actualiza los parámetros** usando el optimizador configurado (por ejemplo, Adam).
- Es el paso en que el modelo "aprende": ajusta θ para reducir la energía de los datos reales.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ③
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ④
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ⑤
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ⑥
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

def test_step(self, real_imgs): ⑦
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```



Actualización de métricas

- Guarda los valores promedio de cada pérdida y de las energías reales/falsas.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ❸
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ❹
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ❺
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ❻
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics} ❼

```

```

def test_step(self, real_imgs): ❽
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]} ❼

```

→ Devuelve un diccionario con los valores de todas las métricas acumuladas.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ③
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ④
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ⑤
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ⑥
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

```

Cómo se evalúa el modelo en validación (sin actualizar pesos).

```

def test_step(self, real_imgs): ⑦
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )
    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
    return {m.name: m.result() for m in self.metrics[2:]}

```

- Genera imágenes falsas desde ruido aleatorio (no usa el buffer ni Langevin).
- Calcula las energías reales y falsas con el modelo.

```

with tf.GradientTape() as training_tape:
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ❸
    cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    ) ❹
    reg_loss = self.alpha * tf.reduce_mean(
        real_out ** 2 + fake_out ** 2, axis = 0
    ) ❺
    loss = reg_loss + cdiv_loss
grads = training_tape.gradient(loss, self.model.trainable_variables) ❻
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables)
)
self.loss_metric.update_state(loss)
self.reg_loss_metric.update_state(reg_loss)
self.cdiv_loss_metric.update_state(cdiv_loss)
self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics}

```

Cómo se evalúa el modelo en validación (sin actualizar pesos).

```

def test_step(self, real_imgs): ❷
    batch_size = real_imgs.shape[0]
    fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
    inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
    real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
    cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
        real_out, axis = 0
    )

```

Igual que en el entrenamiento, mide la diferencia de energías, pero **sin actualizar los pesos** (no hay gradientes).

```

    self.cdiv_loss_metric.update_state(cdiv)
    self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
    self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
return {m.name: m.result() for m in self.metrics[2:]} ❾

```

- Actualiza las métricas relevantes para la evaluación (a partir de la tercera: cdiv, real, fake).

❾ Un diccionario con el nombre y valor promedio de cada métrica hasta ese momento.

```
ebm = EBM()
ebm.compile(optimizer=optimizers.Adam(learning_rate=0.0001), run_eagerly=True)
ebm.fit(x_train, epochs=60, validation_data = x_test,)
```

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.