Modelos generativos y difusión

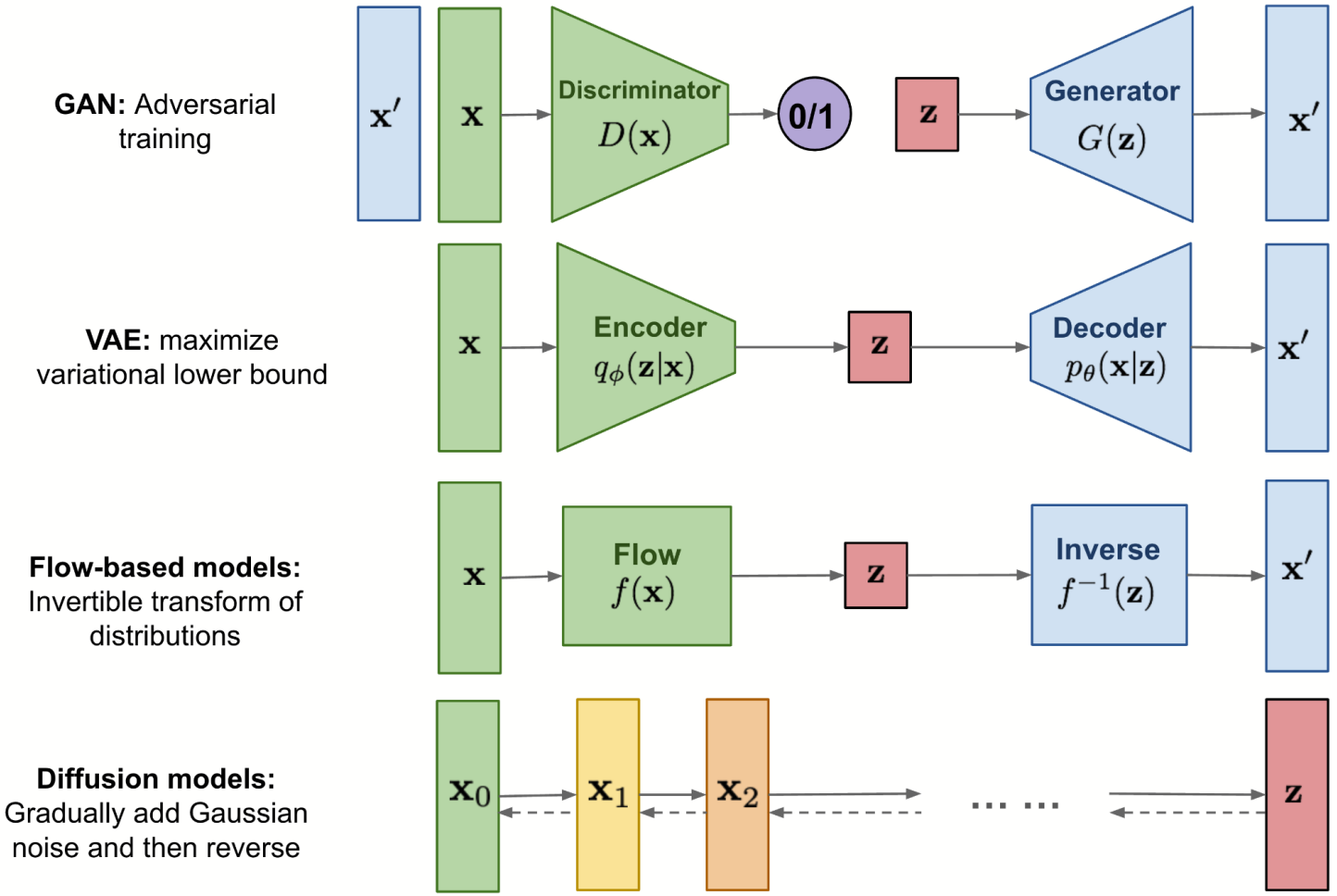
- **Autoencoders**
- **VAE**
- **GAN**
- **DDPM/DDIM**
- **Difusión condicionada**
- **DiT**

Temario

- Autoencoders (AE) y VAE
- GAN y entrenamiento adversario
- CLIP: alineamiento texto-imagen
- Difusión: idea central, DDPM y DDIM
- Denoiser multi-escala: U-Net y atención
- Difusión condicionada (texto -> imagen) y DiT

Panorama: familias de modelos generativos

Cuatro ideas recurrentes: latentes, critic, invertibilidad, denoising



Autoencoders (AE)

Encoder comprime · Decoder reconstruye

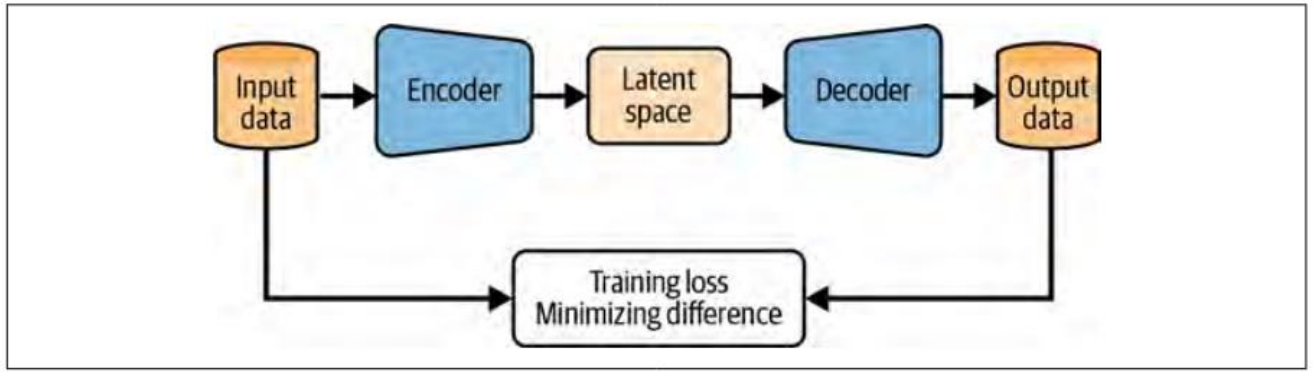
Idea

- Aprende una representación latente z útil para compresión o features.
- Optimiza pérdida de reconstrucción (MSE/BCE).
- Útil para anomalías (alto error de reconstrucción).

Limitación típica

- Si se usa como generador, puede producir reconstrucciones "promedio".
- No define explícitamente una distribución p(x) para muestrear bien.

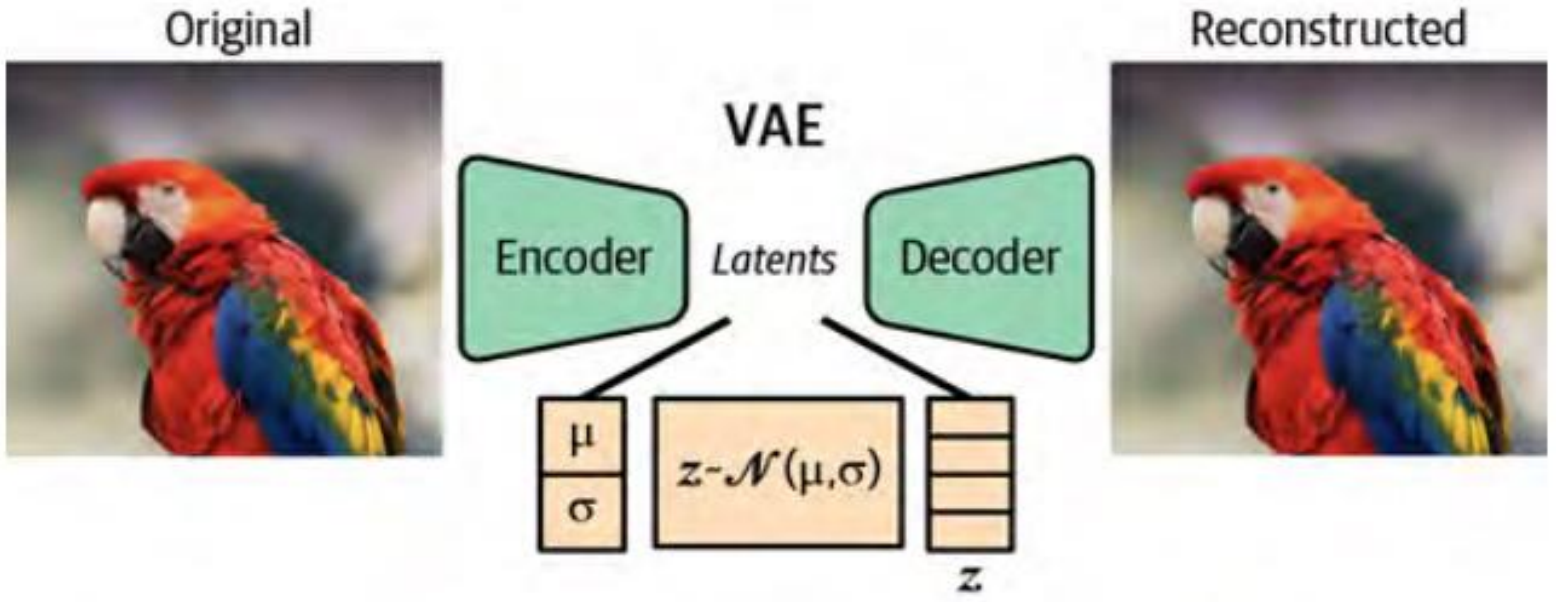
Esquema



x -> encoder ->z -> decoder -> $\hat{x}$

VAE: autoencoder probabilístico

Intuición visual



Pérdida (forma típica)

$$\mathcal{L} = \mathcal{L}_{rec} + \beta\, D_{KL}(q(z \mid x) \,\|\, p(z))$$

El latente z es muestreable -> generación coherente si p(z) ≈ N(0,I) y el decoder generaliza.

GAN: entrenamiento adversario

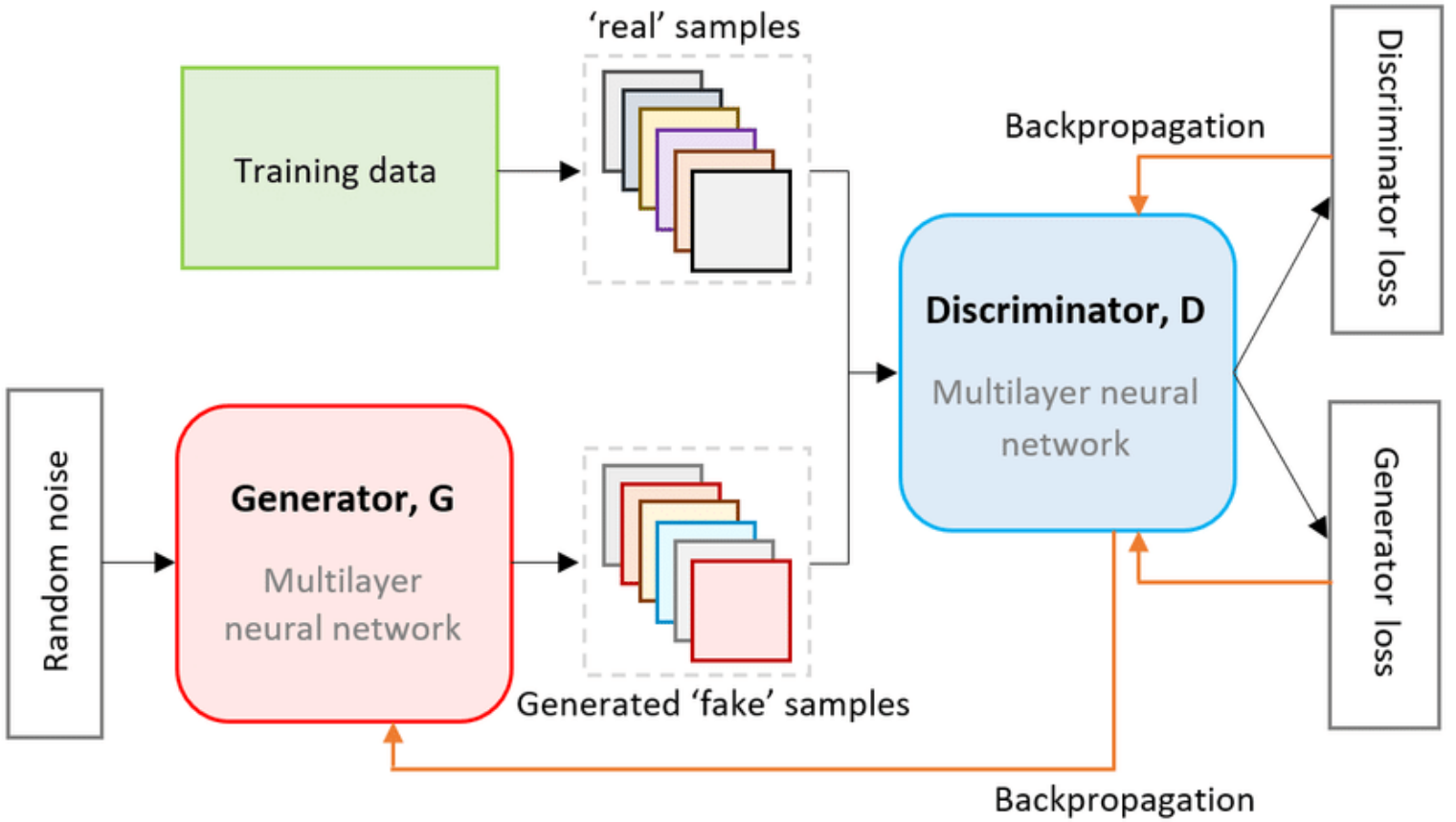
G intenta engañar · D intenta distinguir

Mecánica

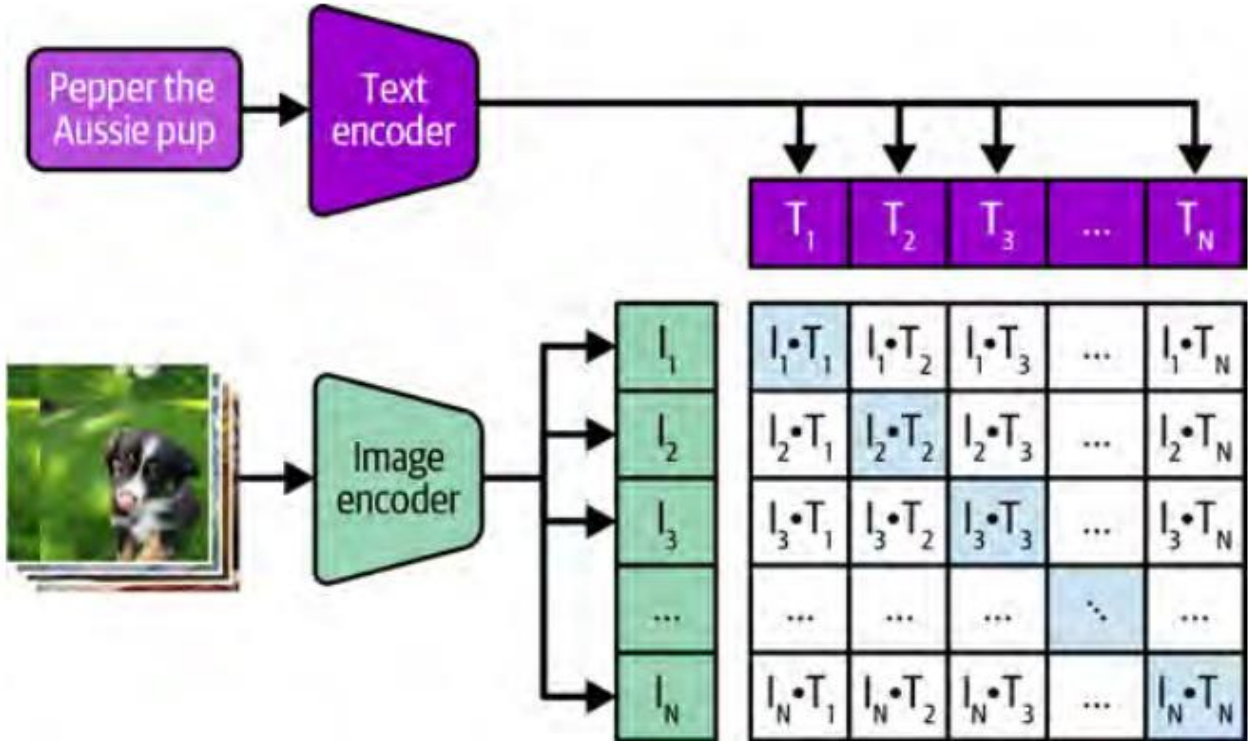
- Dos redes: Generador G(z) y Discriminador D(x).
- Juego minimax: G maximiza el error de D.
- Suele dar imágenes nítidas (alta fidelidad visual).

Problemas típicos

- Inestabilidad (balance G/D, colapso de modo).
- Entrenamiento sensible a la arquitectura e hiperparámetros.
- Difícil estimar likelihood de forma directa.



CLIP (Contrastive Language-Image Pre-training)

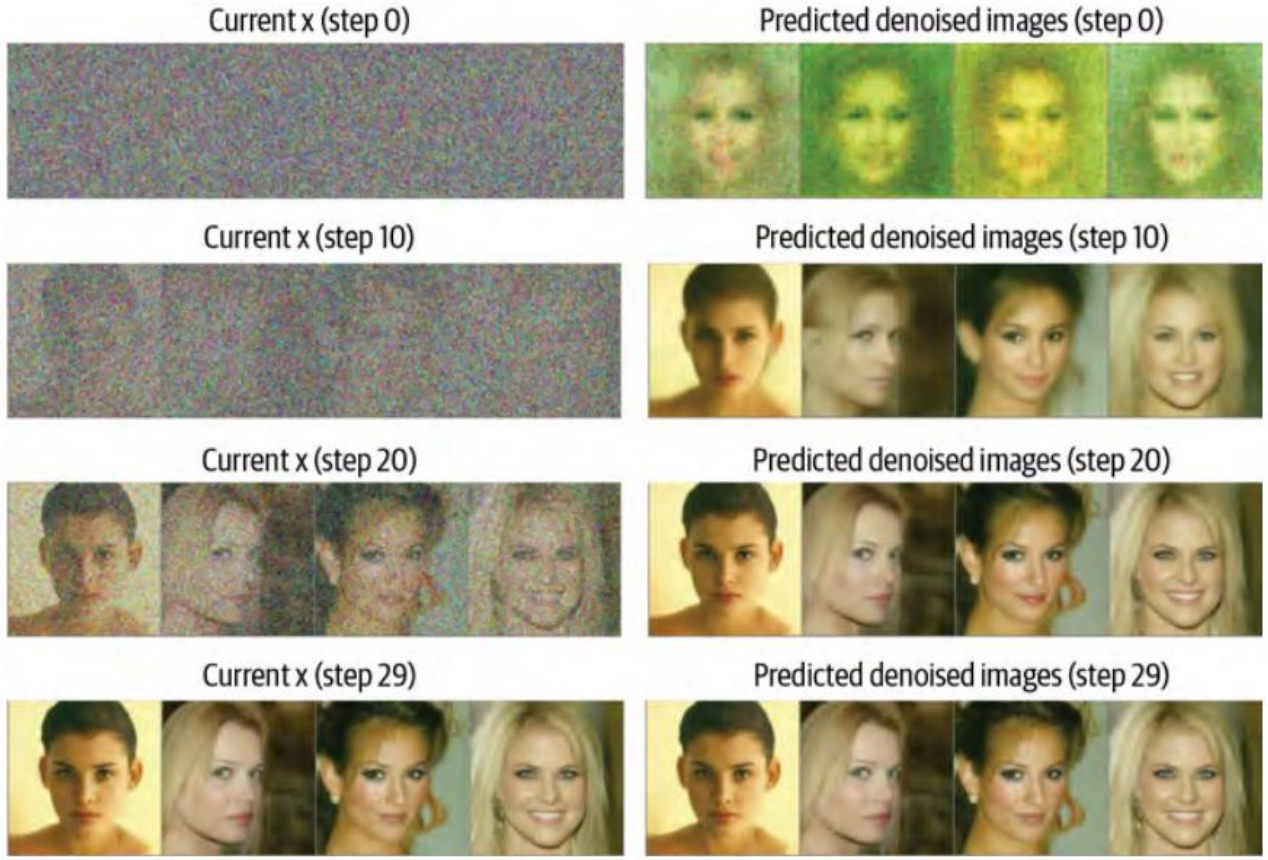


Caso

- Conjunto de datos: imágenes y subtítulos de texto que describen dichas imágenes.
- El objetivo de CLIP es crear un modelo que mida la precisión con la que un texto describe el contenido de una imagen para un par de imágenes de texto arbitrario que proporcionemos.

CLIP utiliza una función de pérdida llamada pérdida contrastiva

Difusión: la idea central



Difusión

se parte de una imagen casi puro ruido x_T y se aplica un proceso iterativo x_t -> x_{t-1}) para **reconstruir** una imagen plausible.

- (Izquierda) Se muestra el estado actual x_t. Al inicio es ruido (step 0) y, con los pasos (10, 20, 29), **aparecen estructuras** y luego detalles finos.
- (Derecha) es lo que la red (típicamente un U-Net) **predice como imagen limpia** en cada paso (≈ (x_0) o el "denoise"). Esa predicción guía la actualización del siguiente x_{t-1}, afinando progresivamente el resultado.

La **difusión** es un proceso generativo que **parte de ruido** y, paso a paso, usa un modelo para **predecir y quitar el ruido** en cada iteración hasta obtener una imagen nítida.

DDPM (Denoising Diffusion Probabilistic Models)

Entrenamiento (DDPM)

Un modelo generativo que produce imágenes empezando desde ruido.
Pertenece a la familia de "difusión": generar = ir quitando ruido en muchos pasos.

Muestreo (reverse)

¿Cómo genera (reverse / muestreo)?
Empieza con una imagen que es solo ruido.
Repite: la red predice el ruido del estado actual y lo resta.
Con cada paso aparecen primero formas globales y luego detalles finos

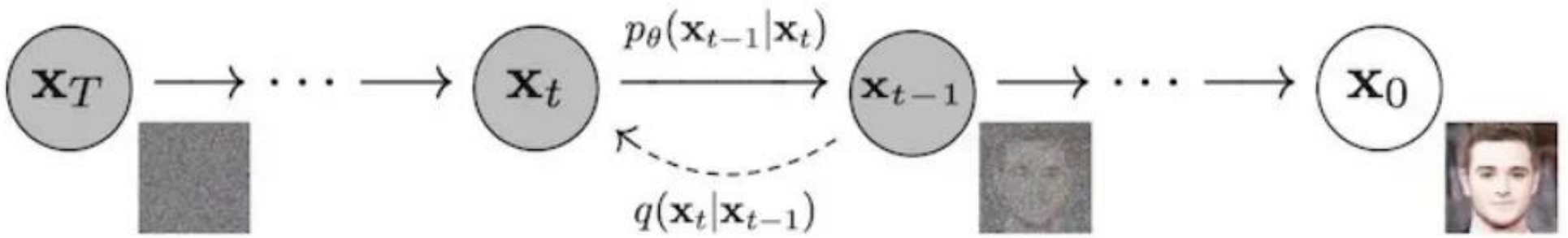
¿Cómo se entrena?
Toma una imagen real y elige al azar cuánta "suciedad" (ruido) agregar.
Entrena la red para reconocer el ruido agregado y aprender cómo retirarlo.
Esto hace que el modelo entienda casos fáciles (poco ruido) y difíciles (mucho ruido).

Claves prácticas
Más pasos => mejor detalle, pero más lento.
Menos pasos => más rápido (aquí entra DDIM como sampler acelerado).
La aleatoriedad controla diversidad: más variedad vs más consistencia.

DDPM: cómo se "revela" una imagen paso a paso

- En los primeros pasos solo hay ruido: el modelo todavía no puede "ver" estructura.
- A medida que se quita ruido, aparecen primero formas globales y luego detalles finos.
- Esto ilustra por qué DDPM usa muchos pasos: cada paso es una corrección pequeña y estable.

DDPM: proceso forward (ruido) y reverse (denoise)



Forward: agregar ruido (solo para enseñar al modelo)
Se toma una imagen real y se le añade ruido de manera gradual.
Así se generan ejemplos con distintos niveles de dificultad (poco vs mucho ruido).
En entrenamiento se elige un nivel al azar: no hace falta simular toda la cadena.

Reverse: quitar ruido (generación real)
El modelo predice qué parte del estado actual es "ruido" y la resta.
Con muchos pasos aparecen primero formas grandes, luego detalles finos.
La aleatoriedad del muestreo controla diversidad vs consistencia.

DDPM: entrenamiento y muestreo (vista operativa)

Algoritmo: Entrenamiento (Training)

Algorithm 1 Training

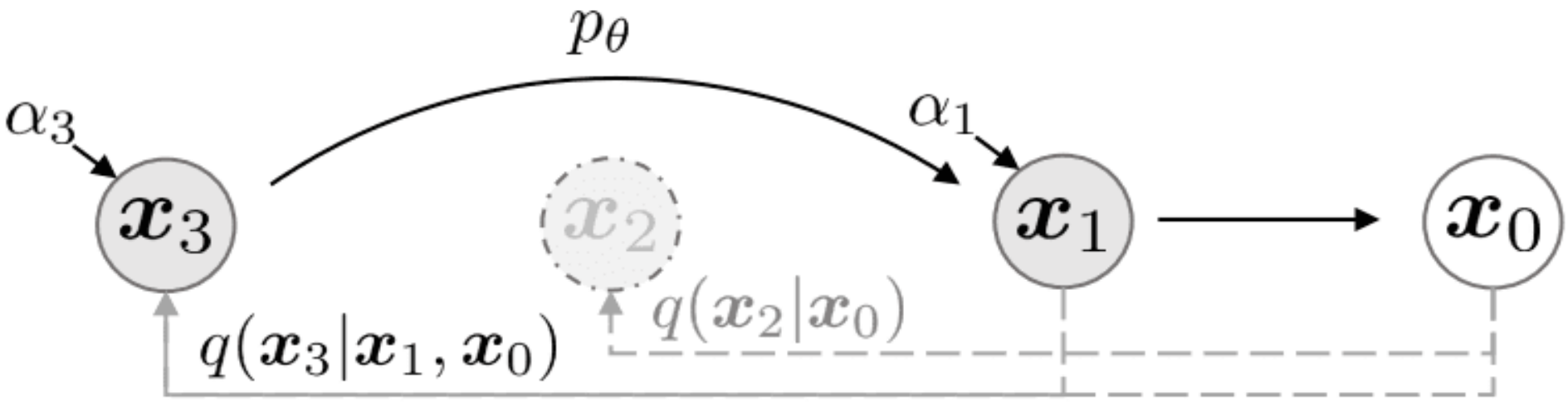
1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algoritmo: Muestreo (Sampling)

Algorithm 2 Sampling

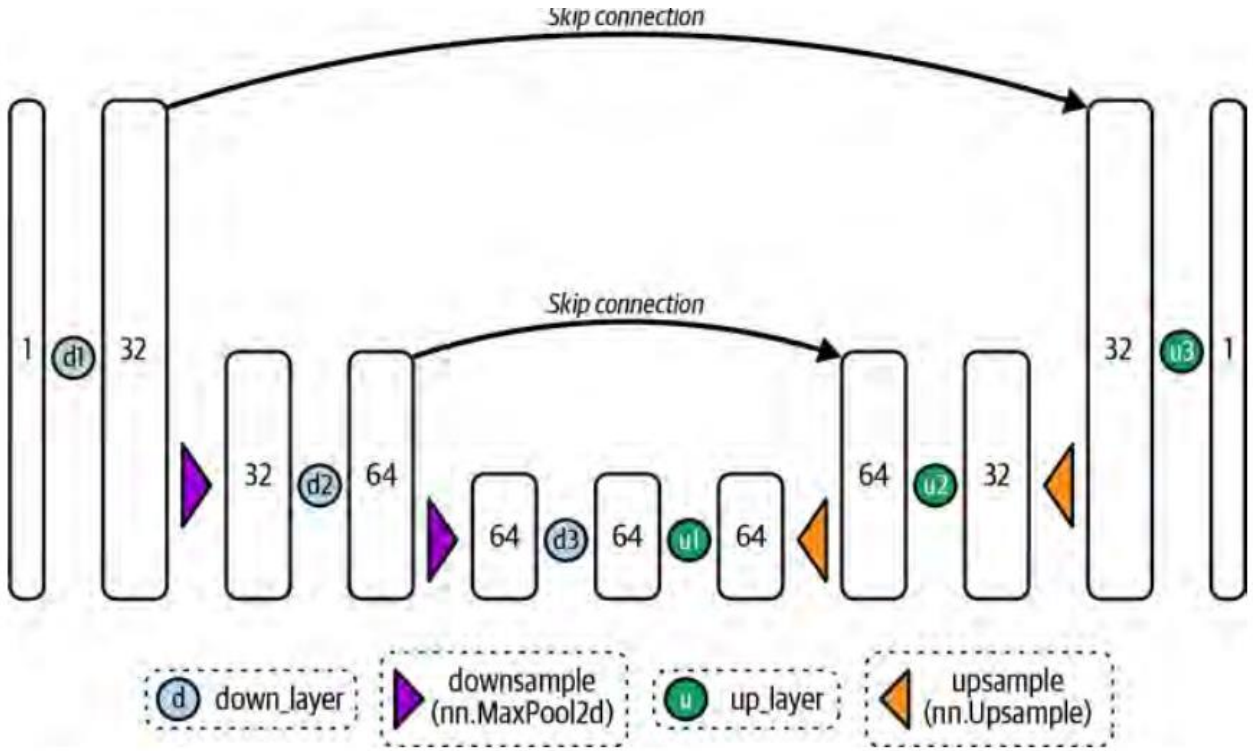
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

DDIM (Denoising Diffusion Implicit Models)



- **DDIM** reutiliza el **mismo modelo** de DDPM, pero **genera con menos pasos** de "desruido".
- En vez de muchos pasos pequeños, usa **saltos entre pasos seleccionados** (**20-50 por ejemplo**).
- Puede ser **casi determinista** (más consistente) o con **aleatoriedad controlada** (más diversidad).
- **No cambia el denoiser ni la programación del ruido**; cambia el **sampler** (qué pasos se recorren y cuánta aleatoriedad se deja).
- **Menos pasos = más rápido** (pero puede perder detalle); **más pasos = más calidad** (se acerca a DDPM).
- En texto-> imagen: ajustar **pasos + guidance** y usar un **scheduler** estable.

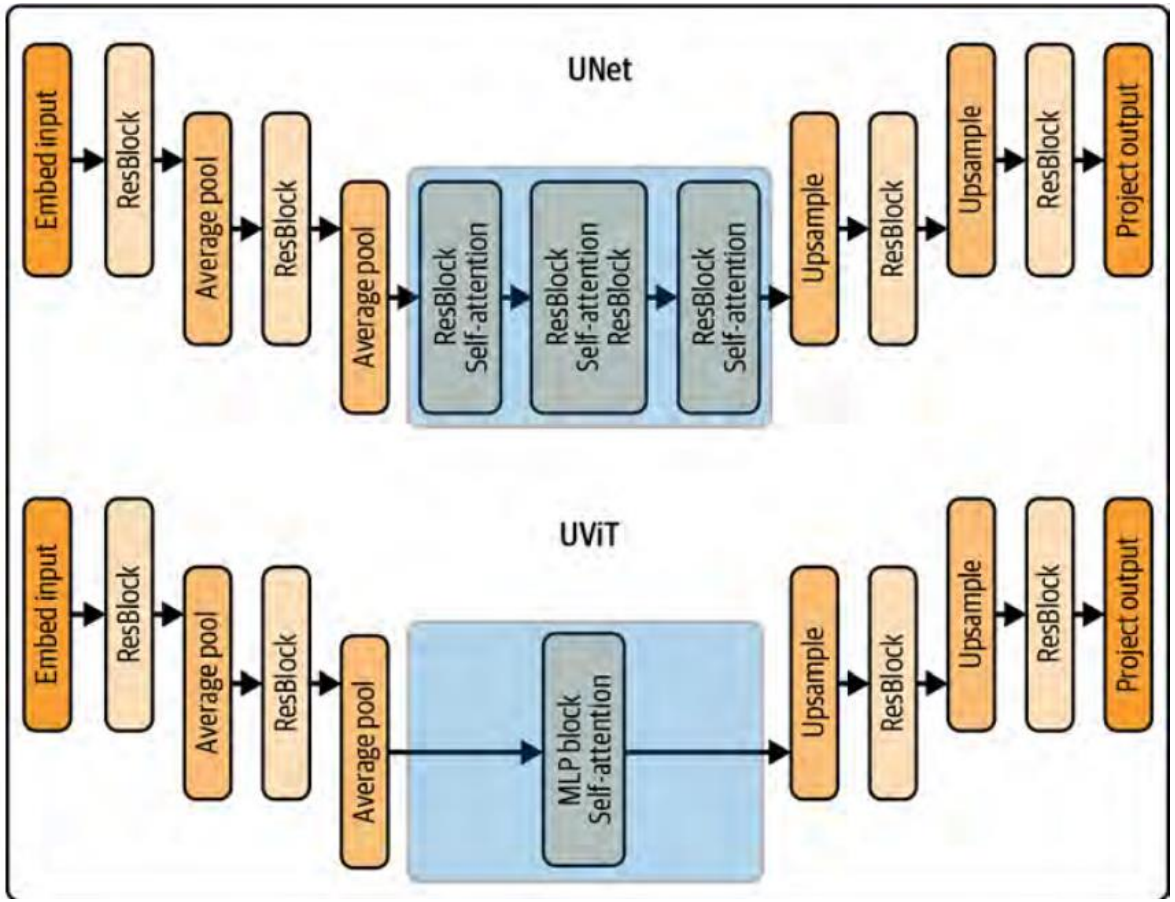
Arquitectura típica: U-Net como denoiser



- **Rol en DDPM (entrenamiento):** se toma una imagen real, se le agrega ruido a un nivel al azar y la U-Net aprende a **predecir el ruido agregado**.
- **Rol en DDPM (muestreo):** empezando desde ruido, se aplica la U-Net muchas veces para **quitar ruido paso a paso** y "revelar" la imagen.
- **Qué cambia en DDIM: la U-Net es la misma,** lo que cambia es el **sampler**: DDIM usa **menos pasos/saltos** para llegar al resultado más rápido, con opción de ser más determinista o más diverso.

- Estructura encoder-decoder (downsample/upsample).
- Skip connections: recuperan y preservan el detalle espacial en la subida.
- Atención suele ponerse en bajas resoluciones (más contexto).

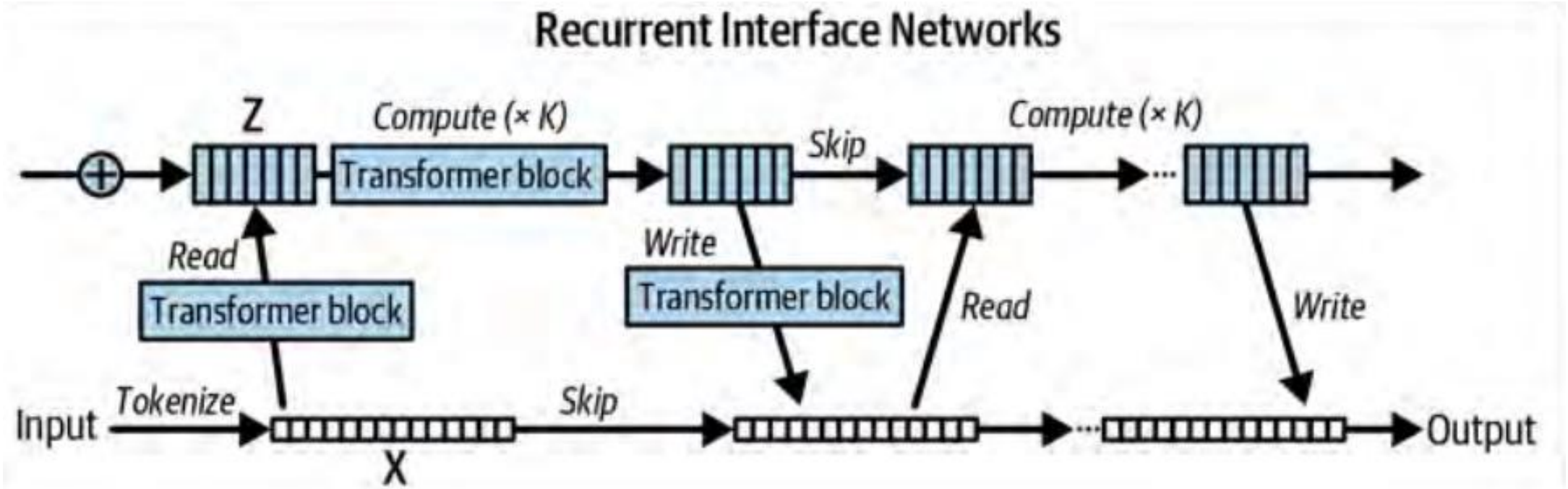
Otras arquitecturas



- UViT/DiT: "patchify" del latente + bloques Transformer.
- Pros: contexto global (atención) y buena escalabilidad.
- Costo $O(N^2)$ en #patches -> se usa latente/patches grandes.
- En práctica: híbridos (Conv + Attention) y atención por etapas.

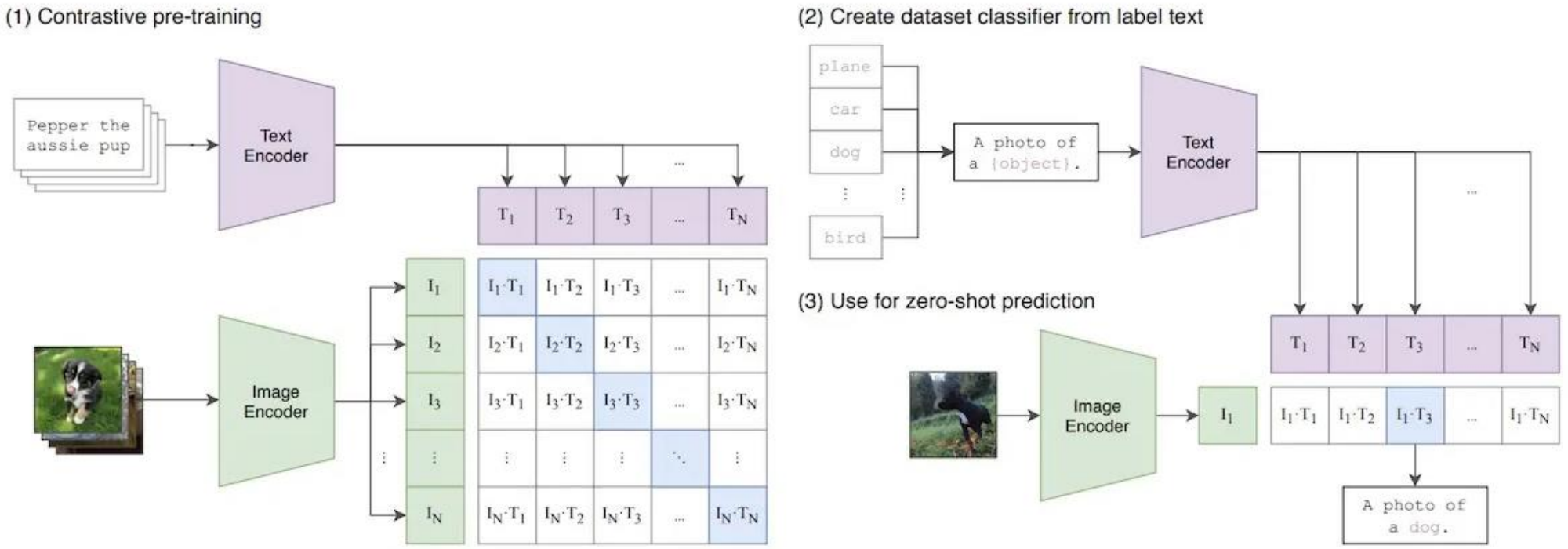
Comparación conceptual: U-Net vs UViT.

Recurrent Interface Networks (RIN): lectura/escritura iterativa



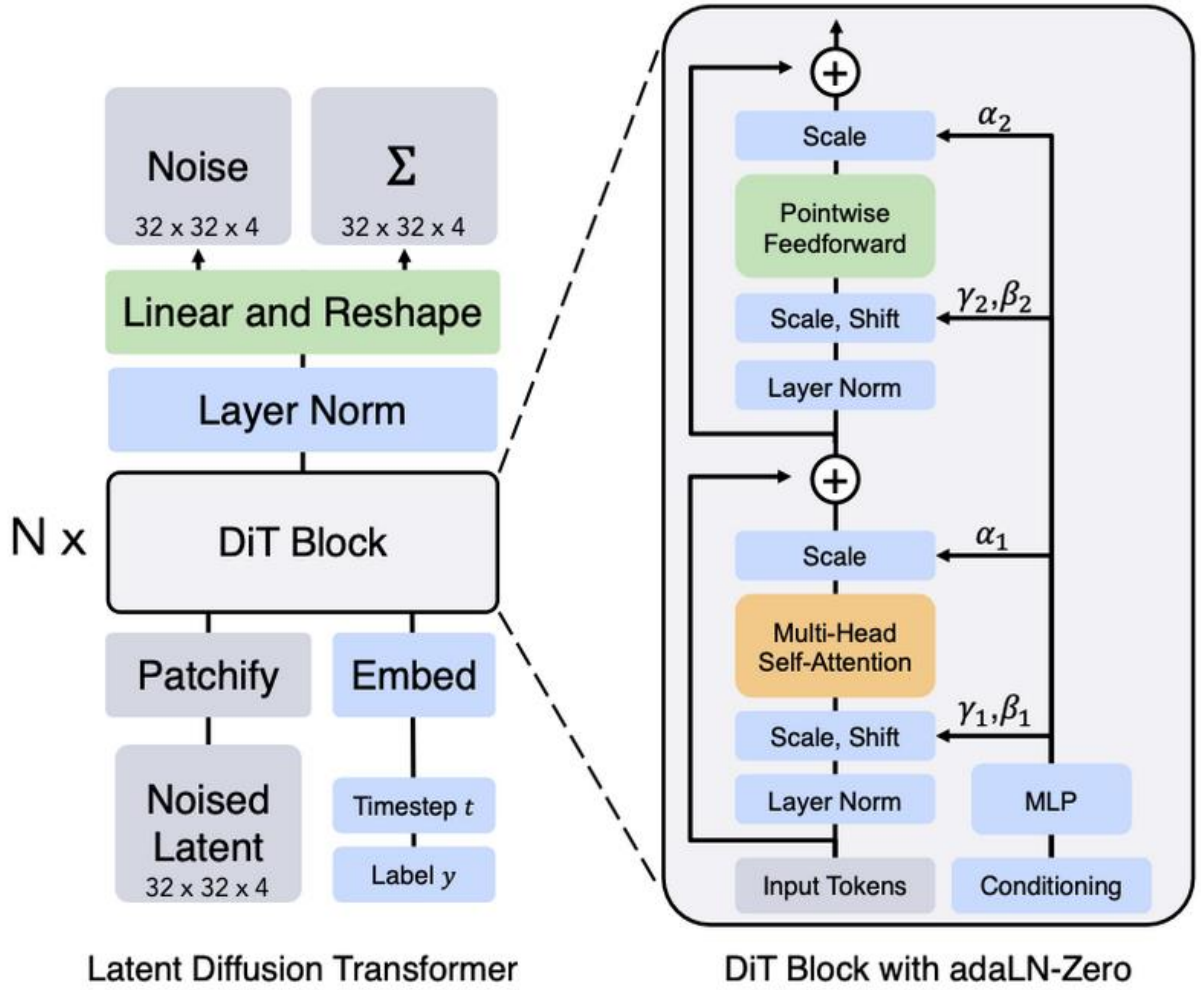
- **Dos estados: X** (tokens externos: texto/imagen) y **Z** (memoria de trabajo latente).
- **Read: Z "lee" X** mediante *cross-attention* para incorporar condición y contexto.
- **Compute (×K):** un Transformer refina **Z** iterativamente (más K => más refinamiento).
- **Write + Skip: Z actualiza X** (write) mientras **X mantiene un camino directo** (skip) hacia la salida.

Texto -> imagen: difusión condicionada



- Cada uno convierte su entrada en un **embedding.**
- Se entrena para que **texto-imagen correctos** tengan alta similitud y los incorrectos baja
- Se crean *prompts* tipo "a photo of a {clase}", se embeben con el **Text Encoder** y se comparan con el embedding de la imagen para predecir la clase.
- El **embedding de texto** que produce CLIP puede usarse como **condición** para guiar un modelo generativo.

DiT: Diffusion + Transformers



Importante

- El modelo recibe un *latente con ruido* y aprende a **predecir/retirar el ruido** para recuperar una versión más limpia.
- El latente se **divide en "parches" (patchify)** y se procesa con **N bloques Transformer**
- Se agregan *embeddings* del **timestep t** y la **condición y** (clase o texto).
- En cada bloque, la condición ajusta la **LayerNorm** con *scale/shift* (γ, β) y controla cuánto se suma en los *residuals* (α), para guiar la generación de forma estable.