# Transformers

**Clase 19**

Dra. Wendy Aguilar

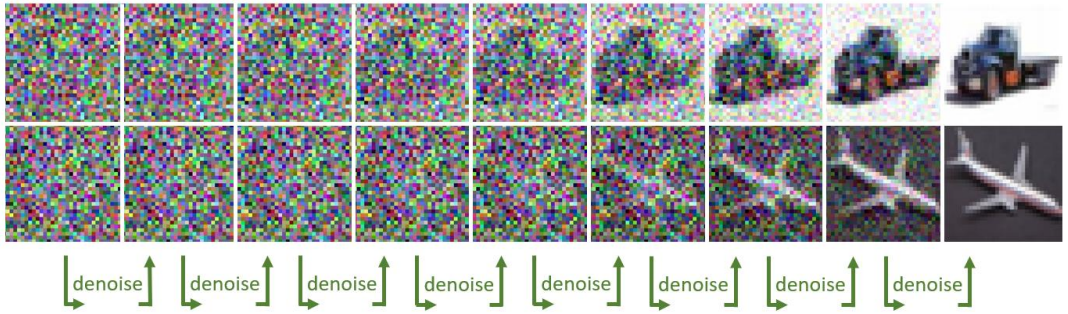
# Modelos Generativos Profundos

UN ENFOQUE DESDE LA CREATIVIDAD COMPUTACIONAL

¿Hay un espacio latente en el modelo DDPM?

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

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



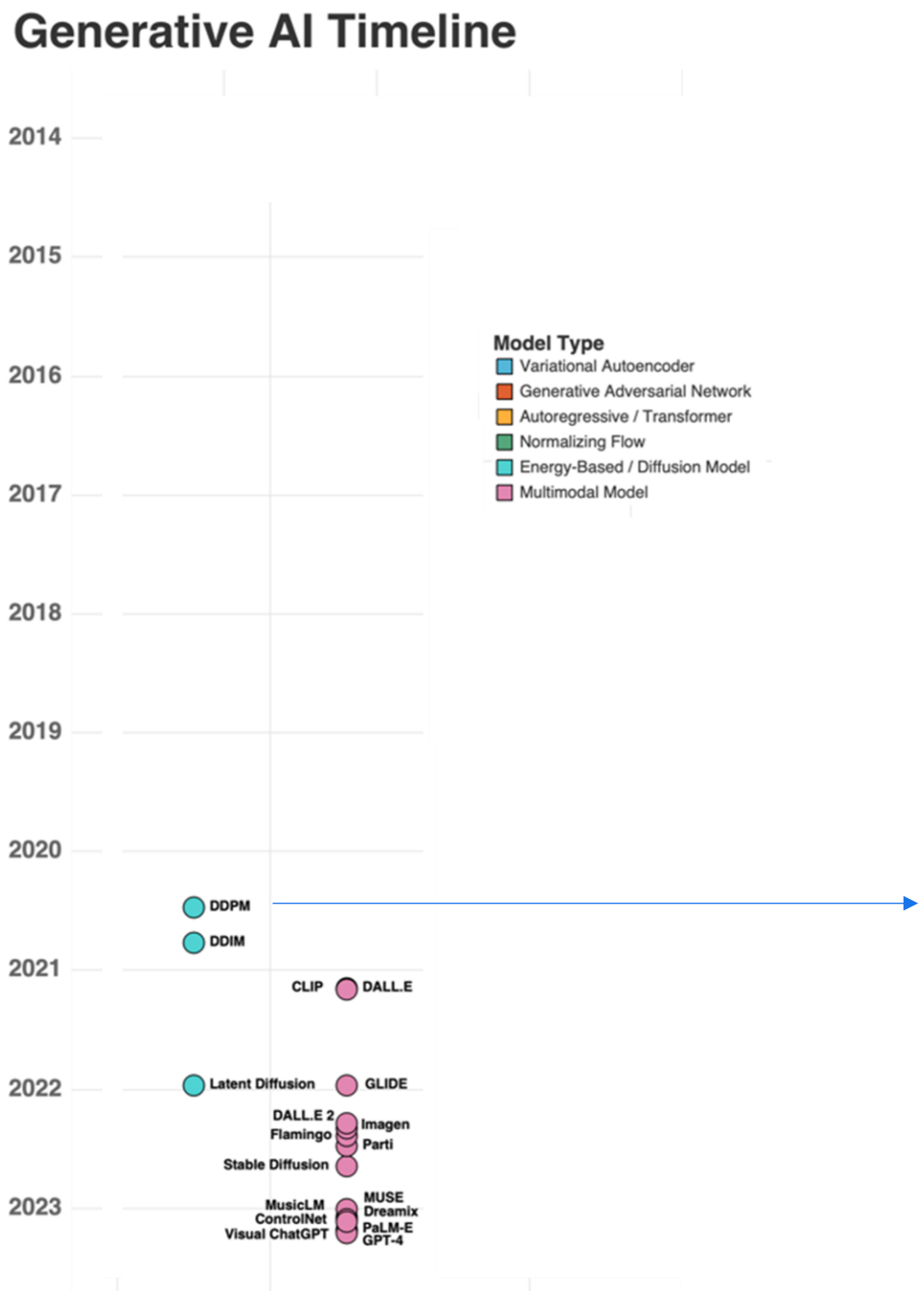

El modelo aprende una trayectoria probabilística $x_T \to x_{T-1} \to \cdots \to x_0$

en el **espacio de los datos mismos**, no en un espacio latente separado.

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

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

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




Ho, J., Jain, A. y Abbeel, P. (2020). *Denoising Diffusion Probabilistic Models*. En *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, pág. 6840-6851.

## Denoising Diffusion Probabilistic Models

**Jonathan Ho**
UC Berkeley
jonathanho@berkeley.edu

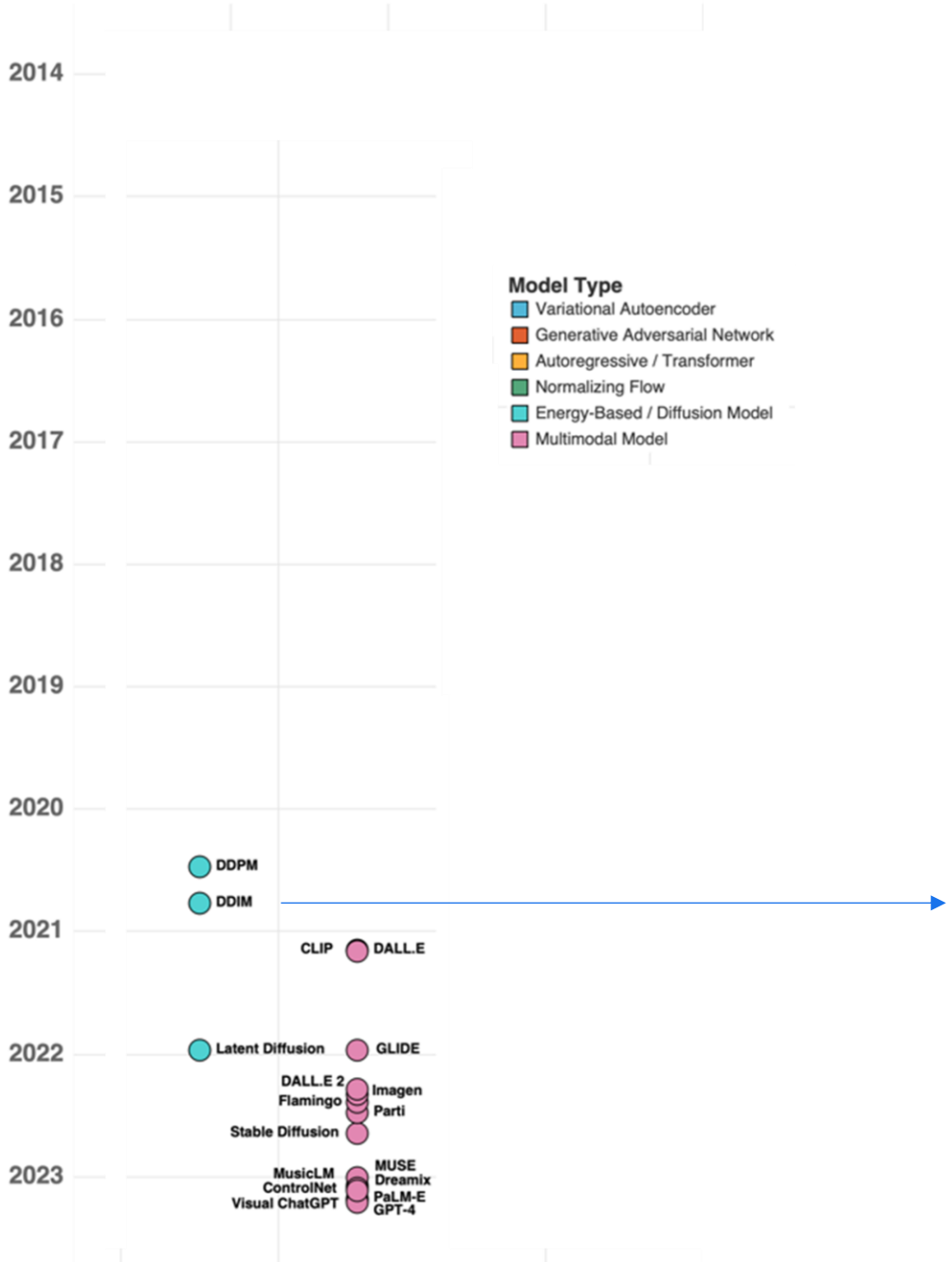
**Ajay Jain**
UC Berkeley
ajayj@berkeley.edu

**Pieter Abbeel**
UC Berkeley
pabbeel@cs.berkeley.edu

**Abstract**

We present high quality image synthesis results using diffusion probabilistic models, a class of latent variable models inspired by considerations from nonequilibrium thermodynamics. Our best results are obtained by training on a weighted variational bound designed according to a novel connection between diffusion probabilistic models and denoising score matching with Langevin dynamics, and our models naturally admit a progressive lossy decompression scheme that can be interpreted as a generalization of autoregressive decoding. On the unconditional CIFAR10 dataset, we obtain an Inception score of 9.46 and a state-of-the-art FID score of 3.17. On 256x256 LSUN, we obtain sample quality similar to ProgressiveGAN. Our implementation is available at `https://github.com/hojonathanho/diffusion`.

- En un **DDPM**, el proceso inverso es **aleatorio** (probabilístico).
- Cada paso de denoising introduce un nuevo ruido, por eso se llama *probabilistic model*.

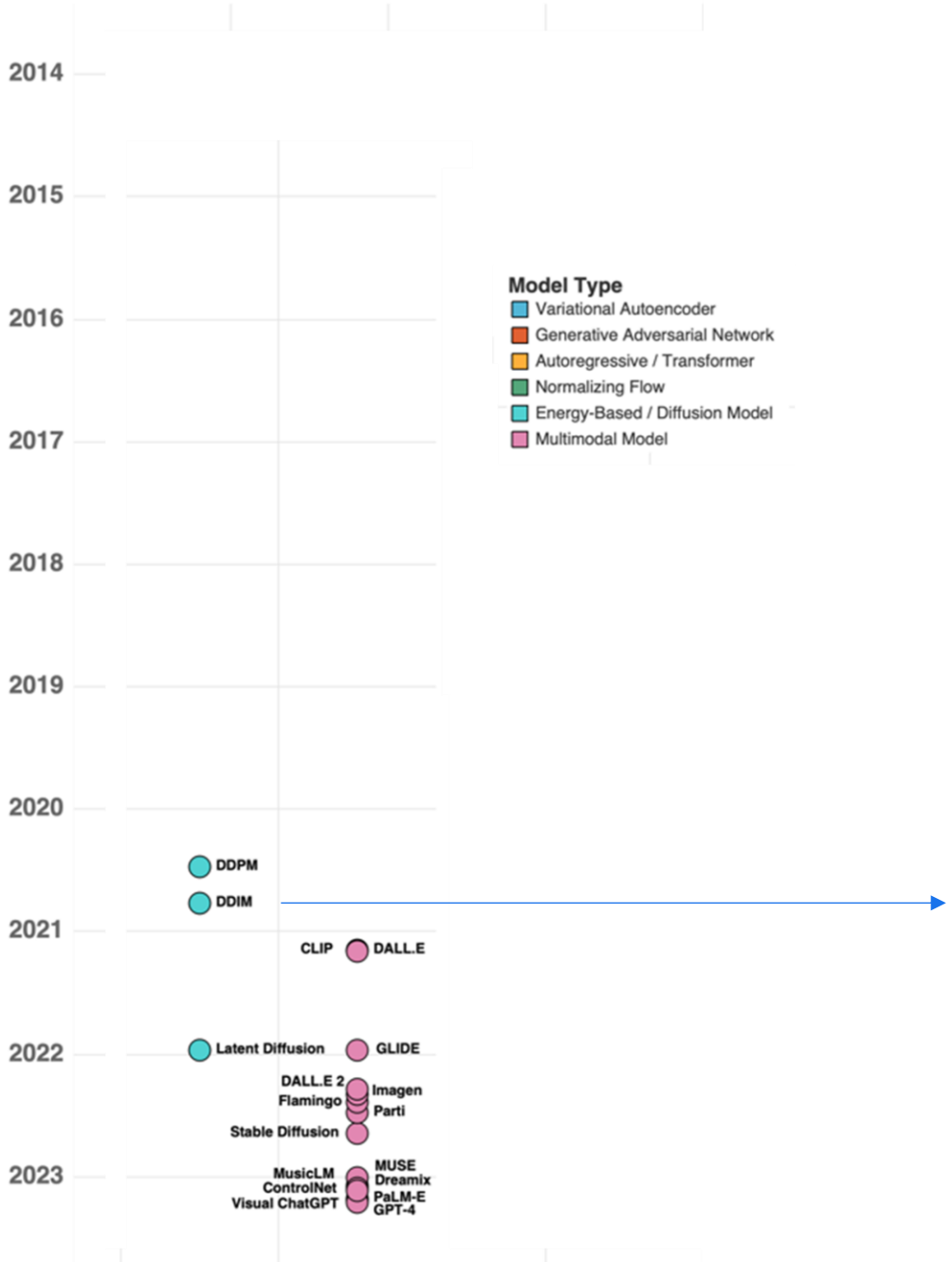



Published as a conference paper at ICLR 2021

# DENOISING DIFFUSION IMPLICIT MODELS

**Jiaming Song, Chenlin Meng & Stefano Ermon**
Stanford University
{tsong,chenlin,ermon}@cs.stanford.edu

- Introduce una variante **determinista** del proceso inverso, que **usa el mismo modelo entrenado de un DDPM**, pero **elimina el ruido estocástico** del muestreo.
- Por tanto, el muestreo es **determinista**, lo que significa:
  - Si fijas la misma semilla y el mismo número de pasos → obtienes **la misma imagen exacta**.
  - Puedes usar **menos pasos de difusión** (p. ej. 20 en lugar de 1000) y aún así obtener buena calidad.




Published as a conference paper at ICLR 2021

# DENOISING DIFFUSION IMPLICIT MODELS

**Jiaming Song, Chenlin Meng & Stefano Ermon**
Stanford University
{tsong,chenlin,ermon}@cs.stanford.edu

- Introduce una variante **determinista** del proceso inverso, que **usa el mismo modelo entrenado de un DDPM**, pero **elimina el ruido estocástico** del muestreo.
- Por tanto, el muestreo es **determinista**, lo que significa:
  - Si fijas la misma semilla y el mismo número de pasos → obtienes **la misma imagen exacta**.
  - Puedes usar **menos pasos de difusión** (p. ej. 20 en lugar de 1000) y aún así obtener buena calidad.

- La implementación que hicimos corresponde a un **DDIM (Denoising Diffusion Implicit Model)**, porque el proceso de generación es **determinista** y **no incluye ruido en los pasos inversos**.

- Esto explica por qué pudimos:
  - Usar diffusion_steps=20 (en lugar de 1000).
  - Obtener la misma secuencia de imágenes cada vez.
  - Visualizar interpolaciones suaves en el espacio de ruido.

# Limitaciones principales del DDIM

## Limitación 1. Operar directamente en el espacio de píxeles

- El proceso de denoising sigue ocurriendo en el **espacio de datos crudo** $x_t \in \mathbb{R}^{H \times W \times C}$.
- En imágenes grandes (ej. 256×256×3 o más), eso implica **dimensiones del orden de 200 mil a varios millones**.
- Cada paso del proceso (aunque sean solo 20) requiere:
  - Convoluciones 2D costosas,
  - Memoria significativa para tensores grandes,
  - Y cómputo intensivo durante el muestreo.

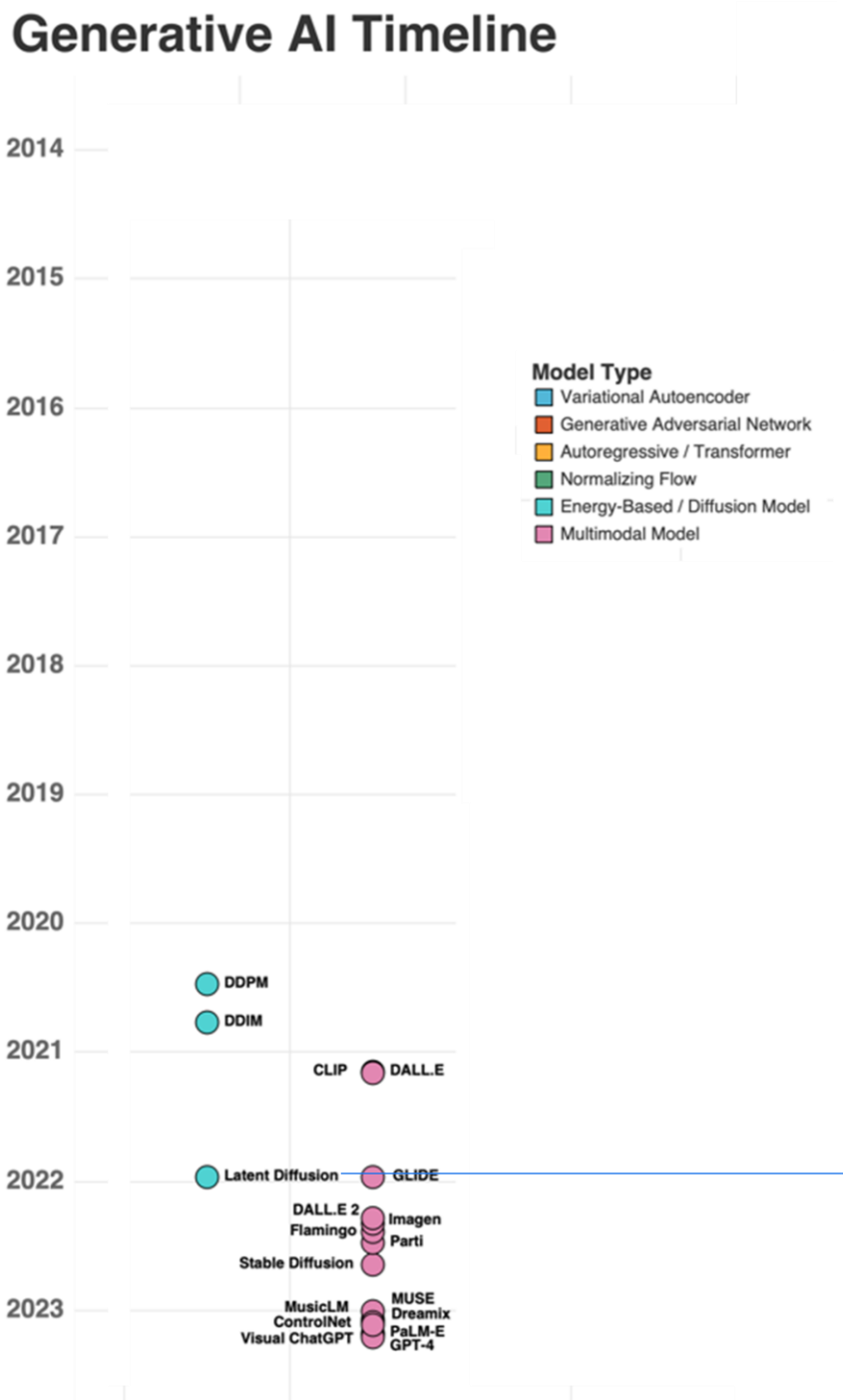
**Consecuencia:** El muestreo sigue siendo lento y costoso, incluso en su versión determinista.

## Limitación 2. Ineficiencia semántica del espacio de píxeles

- Aunque el DDIM es determinista, el modelo:
  - Aprende a eliminar el ruido en el **espacio visual crudo**, donde los cambios son **locales y de bajo nivel (textura, color, ruido)**.
  - No aprovecha una representación **semántica comprimida** (por ejemplo, "rostro", "árbol", "cielo").
- Esto implica que:
  - El modelo necesita aprender relaciones visuales complejas directamente en píxeles.
  - No hay una disociación entre contenido y detalle visual, lo que limita la generalización y el control semántico.

**Consecuencia:** El modelo es costoso de entrenar y requiere enormes datasets y potencia de cómputo para capturar relaciones semánticas de alto nivel.

No es práctico combinar DDIM con información semántica rica
(p.ej., para implementar la generación de imágenes guiadas por texto)




**High-Resolution Image Synthesis with Latent Diffusion Models**

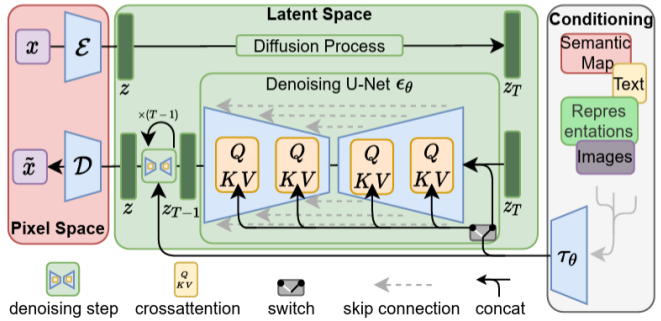
Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

2022

- Surgieron como una extensión de los DDPM/DDIM, buscando superar sus principales limitaciones.

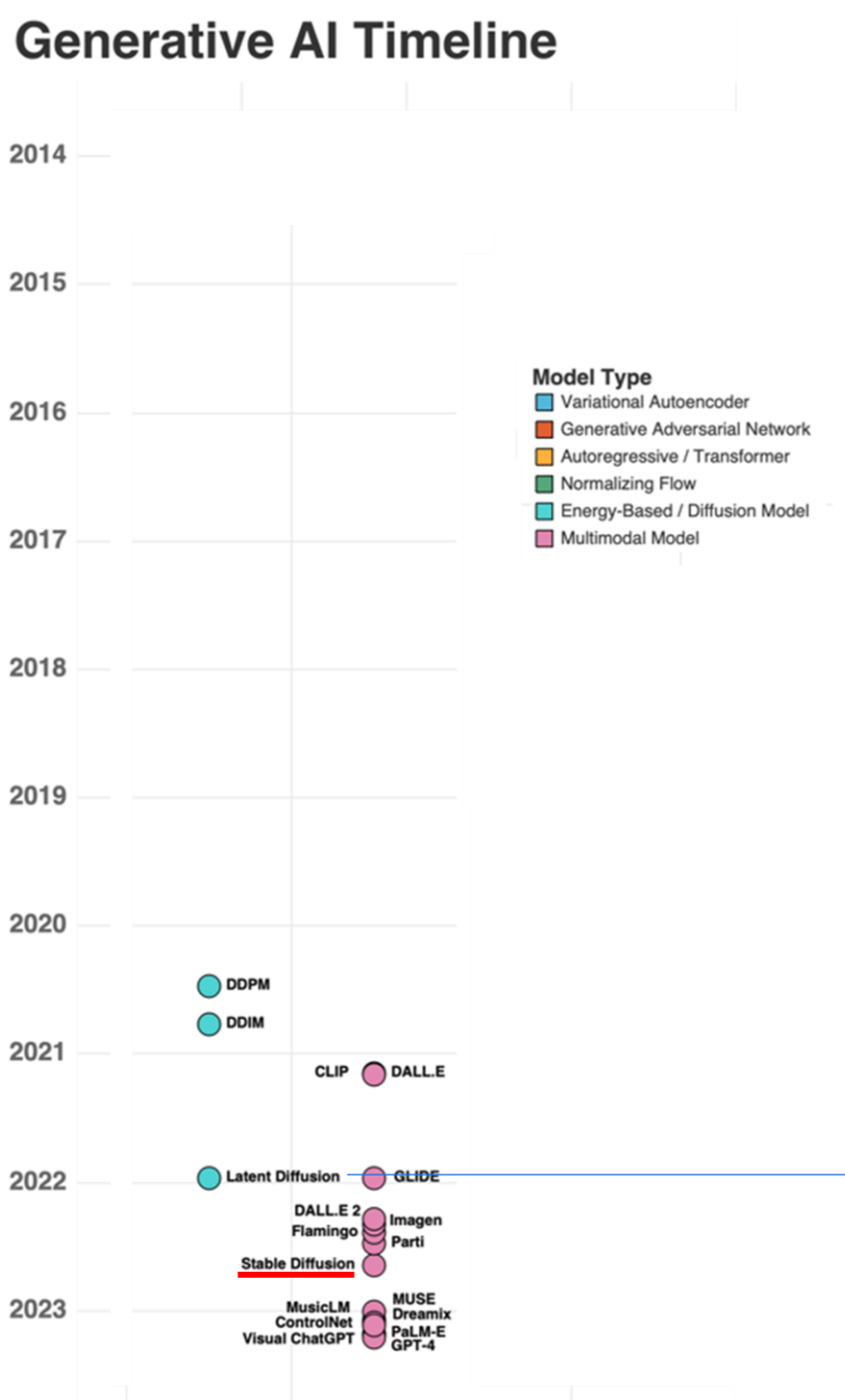
- La idea central fue **mover la difusión del espacio de píxeles al espacio latente** aprendido por un *autoencoder*.

  - En lugar de añadir ruido a cada píxel, se aplica sobre una **representación comprimida** de la imagen.

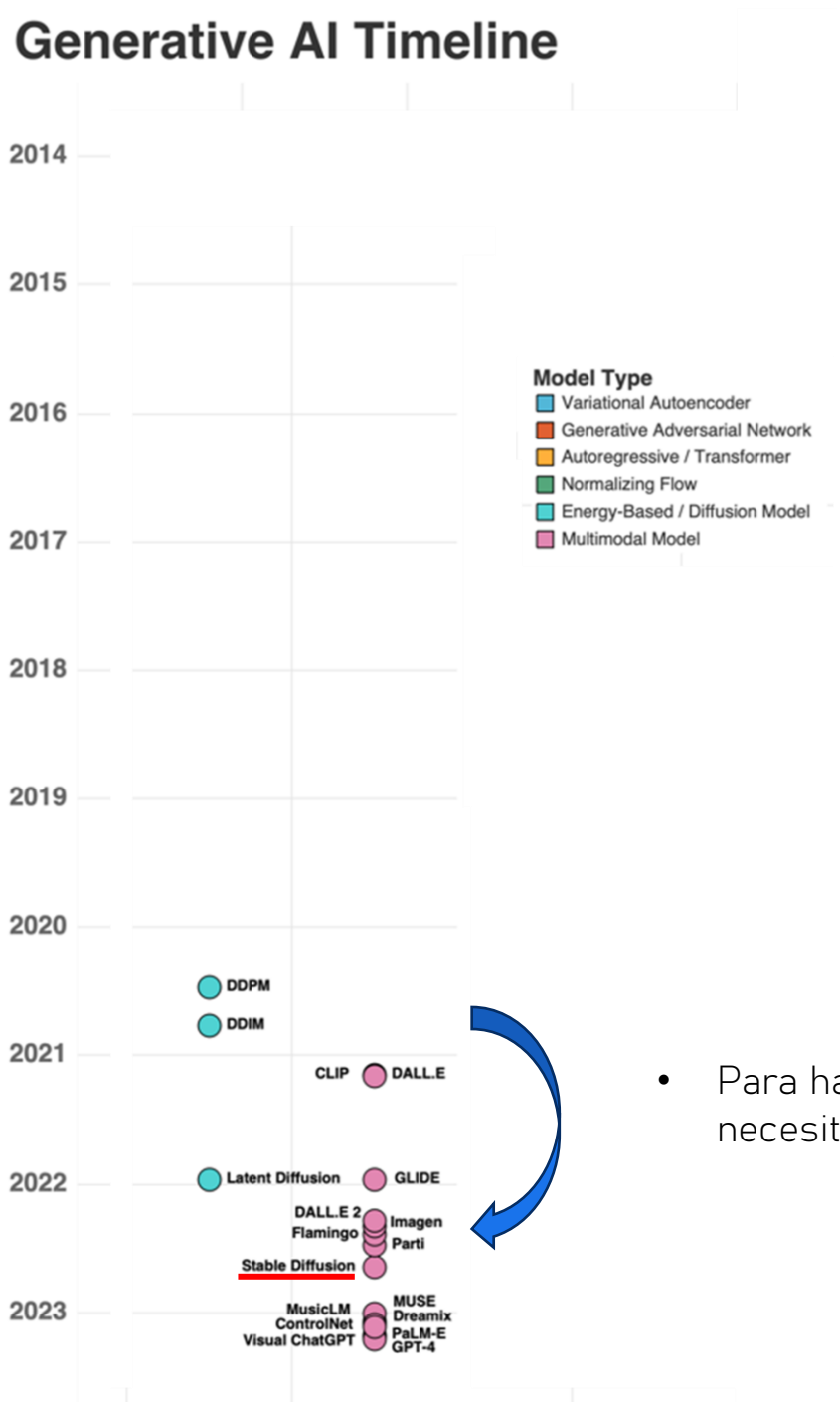



- Ese espacio latente **preserva la estructura semántica** (formas, colores, composición), pero reduce drásticamente la dimensionalidad → menos memoria y cómputo.

El LDM es un **DDIM operando en un espacio latente aprendido**, lo que conserva la calidad generativa, pero con **costo y complejidad reducidos en órdenes de magnitud.**

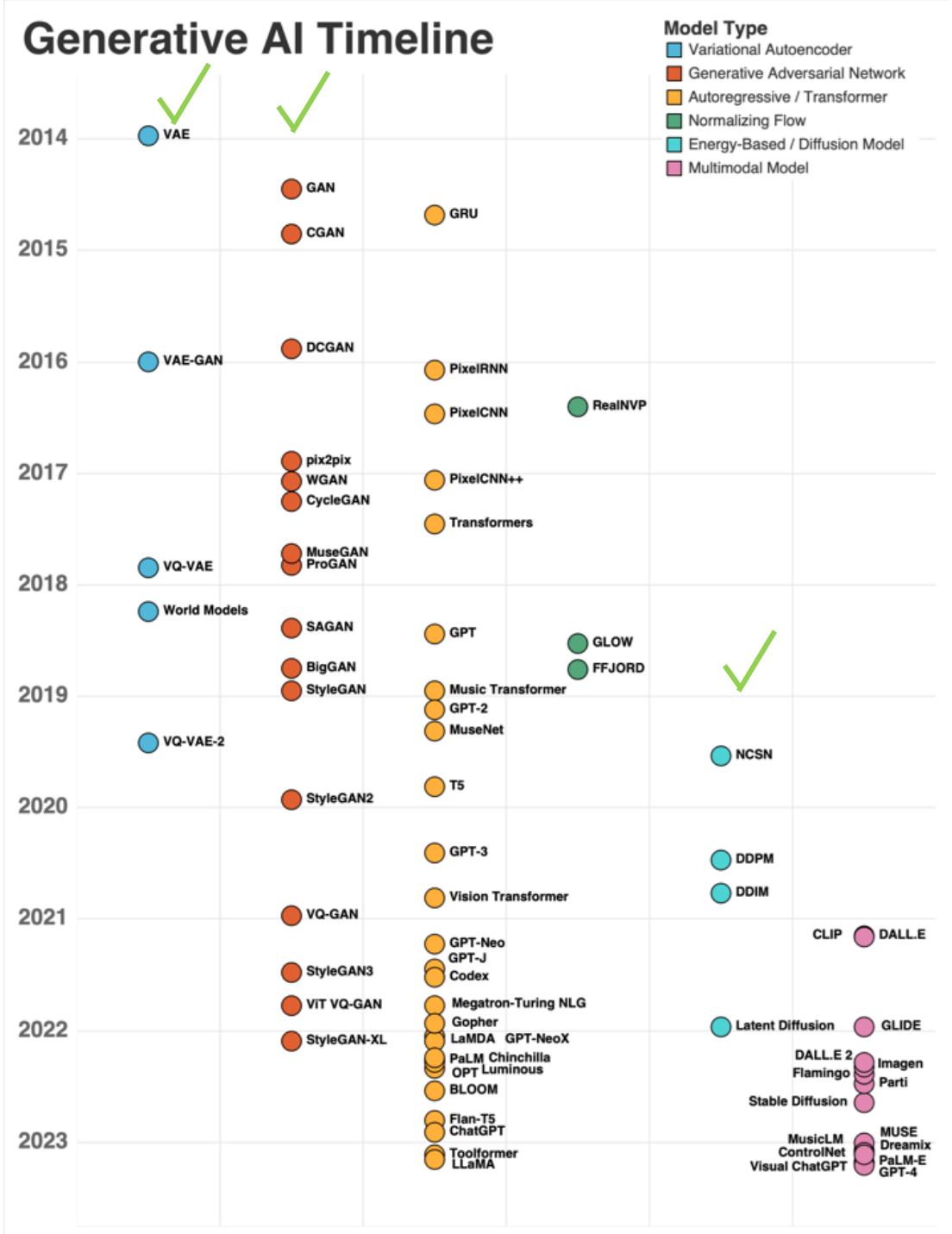


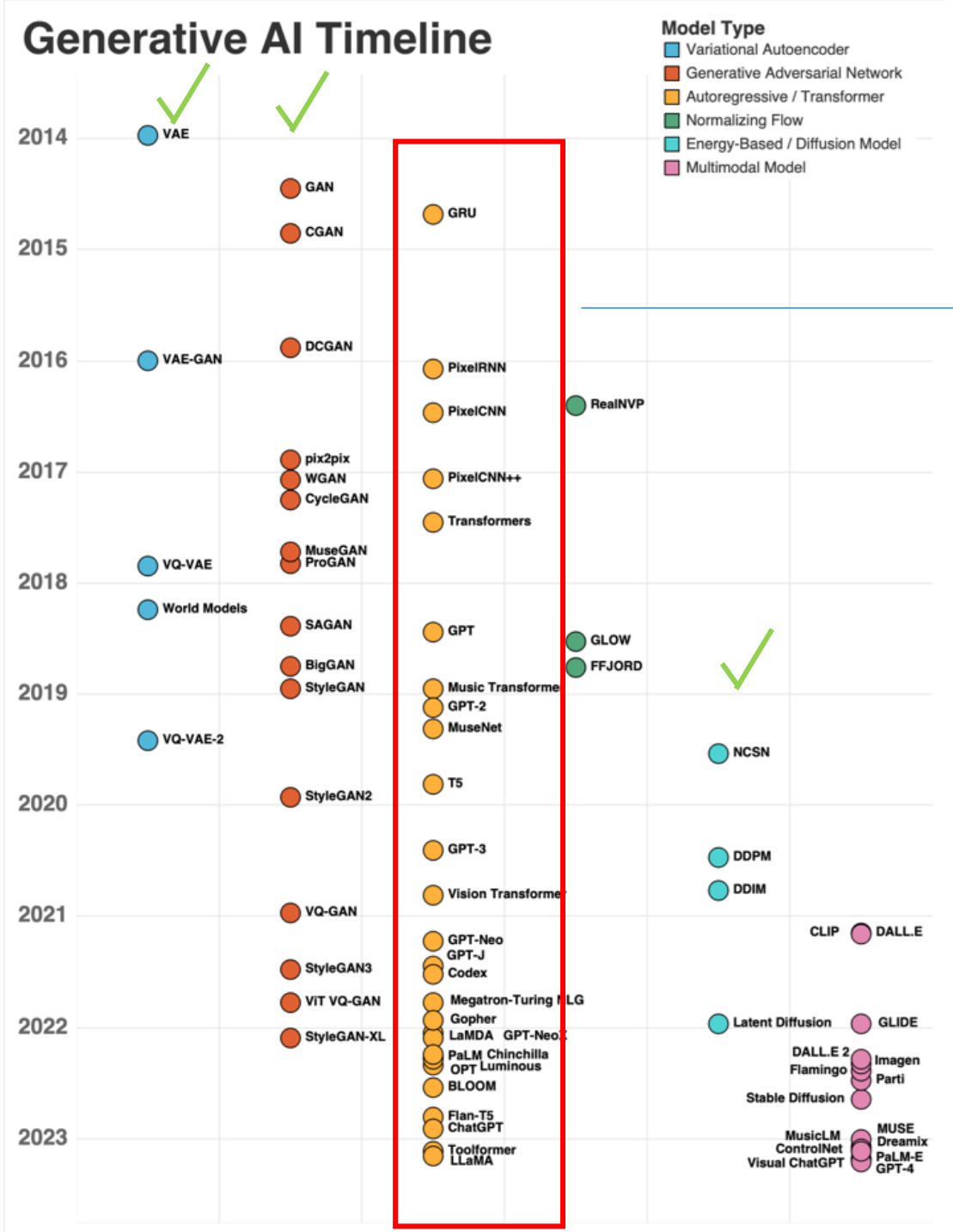

Este enfoque abrió el camino a los **modelos multimodales**, permitiendo incorporar información textual mediante *cross-attention*.
→ Esa combinación dio origen a **Stable Diffusion**, el modelo texto-a-imagen más conocido.




- Para hacer este salto a generar imágenes guiadas por texto, necesitamos introducir un segundo tipo de información:

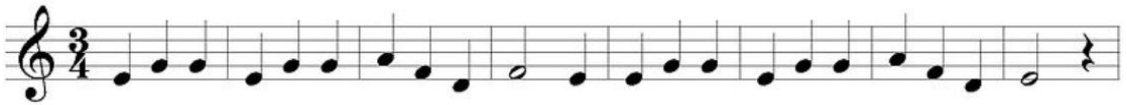
El lenguage

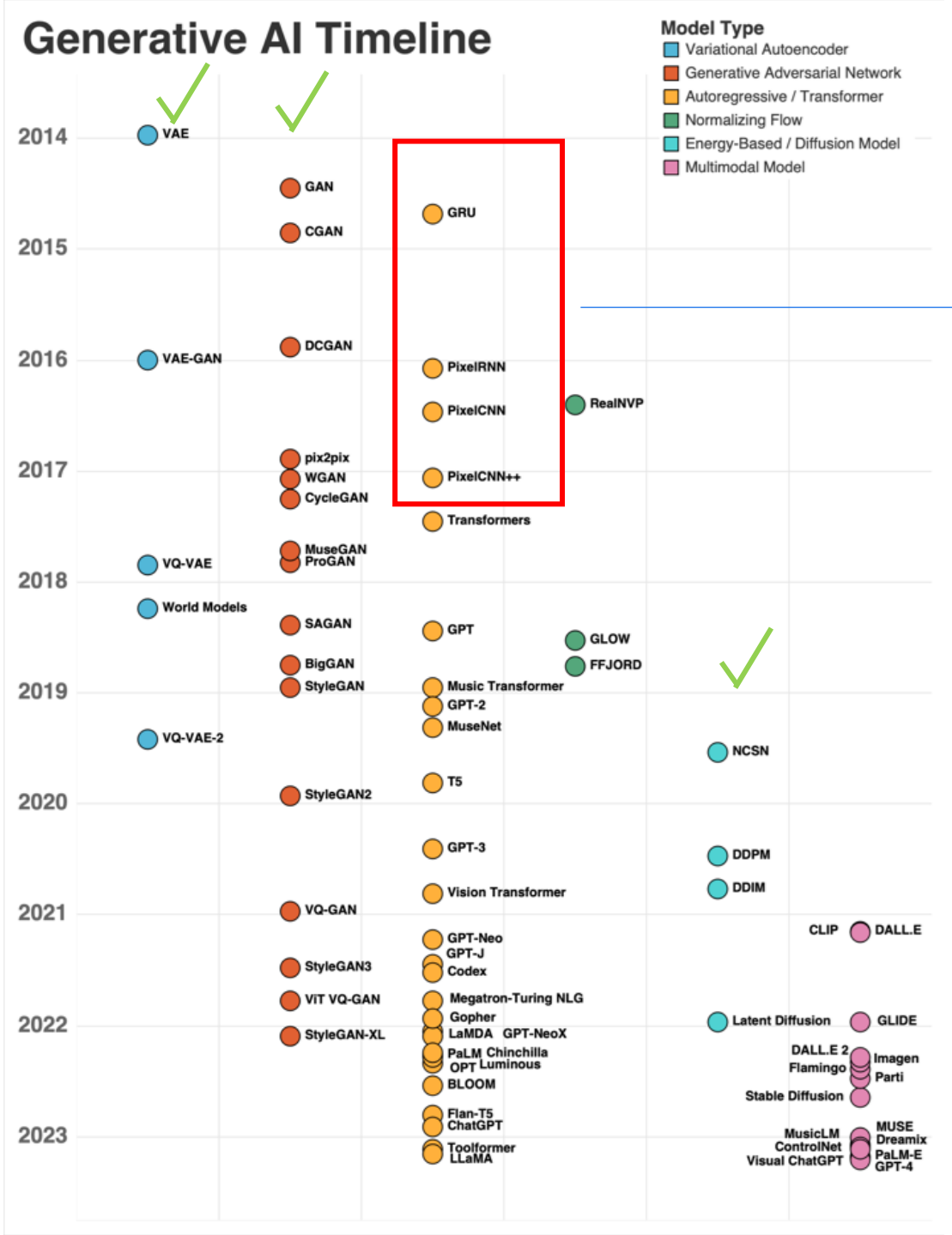

Generative AI Timeline
Model Type
Variational Autoencoder
Generative Adversarial Network
Autoregressive / Transformer
Normalizing Flow
Energy-Based / Diffusion Model
Multimodal Model
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
VAE
GAN
CGAN
GRU
DCGAN
VAE-GAN
PixelRNN
PixelCNN
RealNVP
pix2pix
WGAN
CycleGAN
PixelCNN++
Transformers
MuseGAN
ProGAN
VQ-VAE
World Models
SAGAN
GPT
GLOW
FFJORD
BigGAN
StyleGAN
Music Transformer
GPT-2
MuseNet
VQ-VAE-2
NCSN
T5
StyleGAN2
GPT-3
DDPM
DDIM
Vision Transformer
VQ-GAN
CLIP
DALL.E
GPT-Neo
GPT-J
Codex
StyleGAN3
ViT VQ-GAN
Megatron-Turing NLG
Gopher
LaMDA
GPT-NeoX
Latent Diffusion
GLIDE
StyleGAN-XL
PaLM
Chinchilla
OPT
Luminous
DALL.E 2
Flamingo
Imagen
Parti
BLOOM
Stable Diffusion
Flan-T5
ChatGPT
MusicLM
ControlNet
Visual ChatGPT
MUSE
Dreamix
PaLM-E
GPT-4
Toolformer
LLaMA




Arquitecturas diseñadas para procesar **secuencias ordenadas:**

- Texto
- Audio/Notas musicales
- Video
- Pixeles
- Secuencias genéticas
- Secuencias de acciones o estados en entornos de agentes
- Secuencias de movimientos



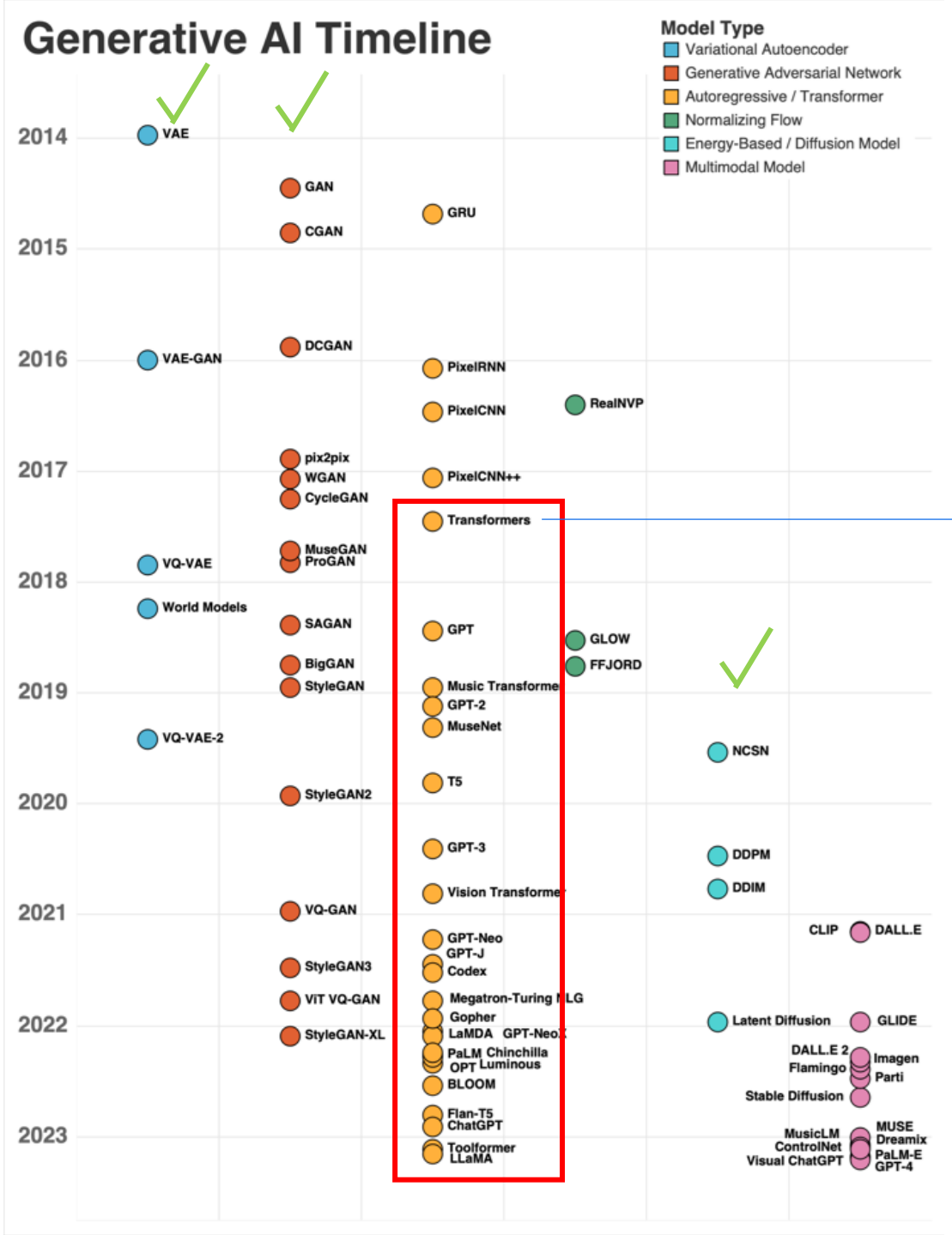



## Antes de 2017:

- Los **modelos recurrentes autoregresivos** dominaban el procesamiento secuencial:
  RNN, LSTM, GRU, PixelRNN, PixelCNN, WaveNet, etc.

- Su lógica era estrictamente temporal: procesar o generar **un elemento a la vez**.

- Tenían **limitaciones** serias:
  - Dificultad para capturar dependencias largas.
  - Entrenamiento secuencial lento.
  - Problemas de gradientes (explosión o desaparición).




**Attention Is All You Need**

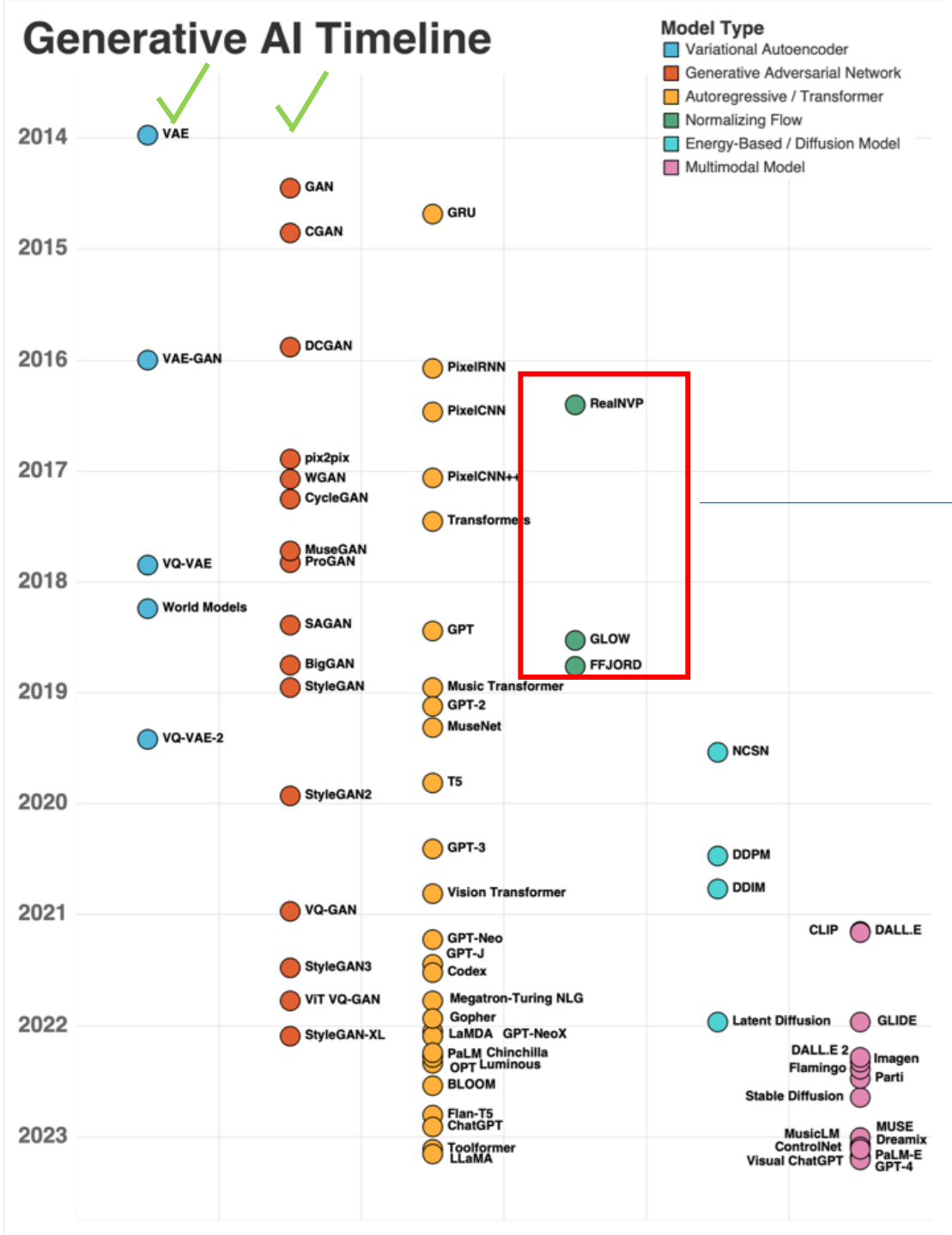

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com


## Desde de 2017: "Attention is all you need"

- El paper de Vaswani et al. (2017) **revolucionó completamente** este panorama:
  - Propuso **eliminar la recurrencia**.
  - Reemplazarla por **atención auto-paralela (self-attention)**.
  - Mantener el esquema **autoregresivo en la salida**, pero con procesamiento global.
- Los Transformers demostraron que podían:
  - Aprender dependencias de largo alcance.
  - Entrenar en paralelo.
  - Escalar a miles de millones de parámetros.

Lo que cambió no fue la lógica autoregresiva, sino la arquitectura que la implementa.

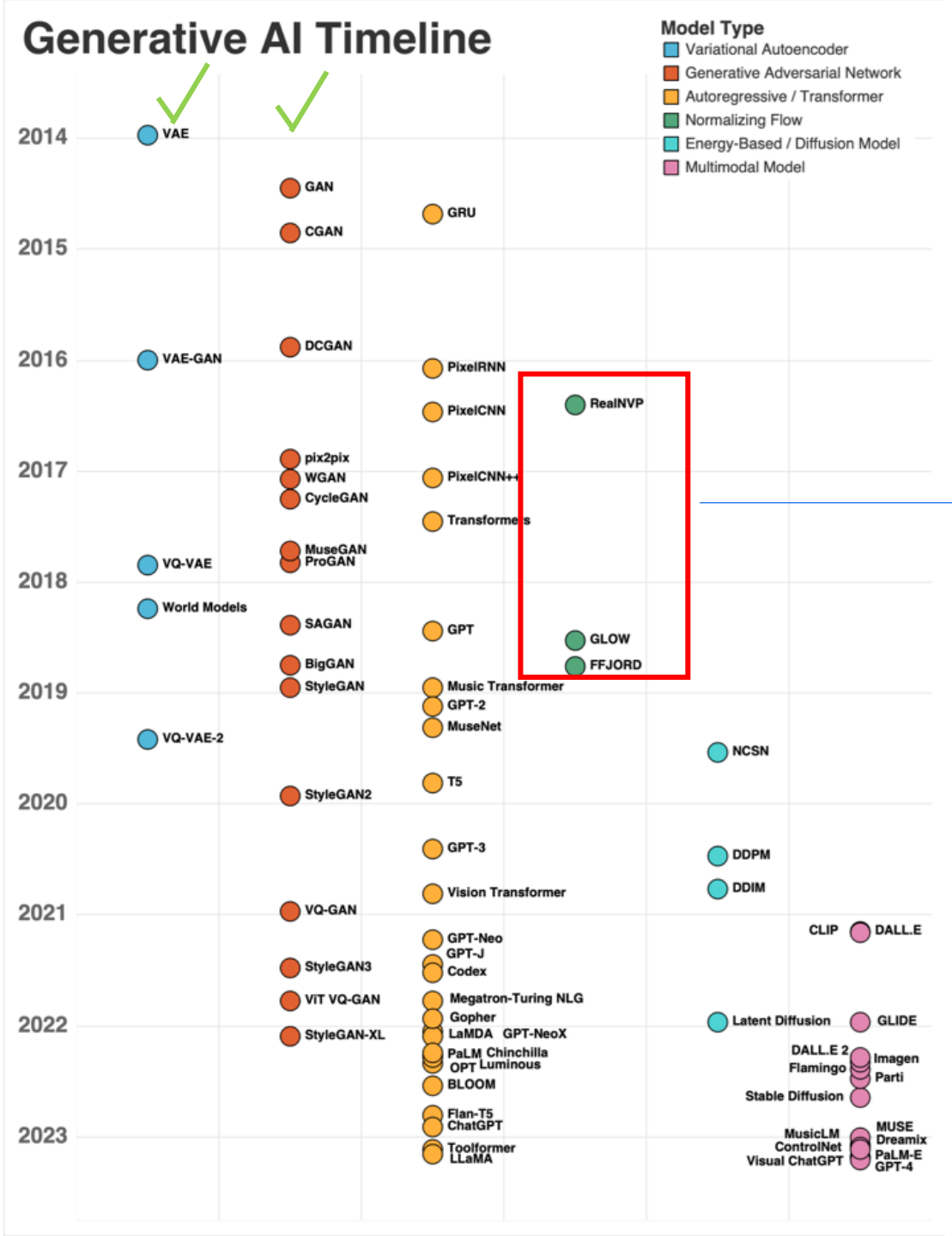
- Aunque las arquitecturas antiguas desaparecieron, el **paradigma autoregresivo** (predecir el siguiente elemento) **sigue siendo el corazón** de:
  - GPT, LLaMA, Claude, Gemini, etc.




## Modelos de normalización de flujo

### Introdujeron una idea fundamental

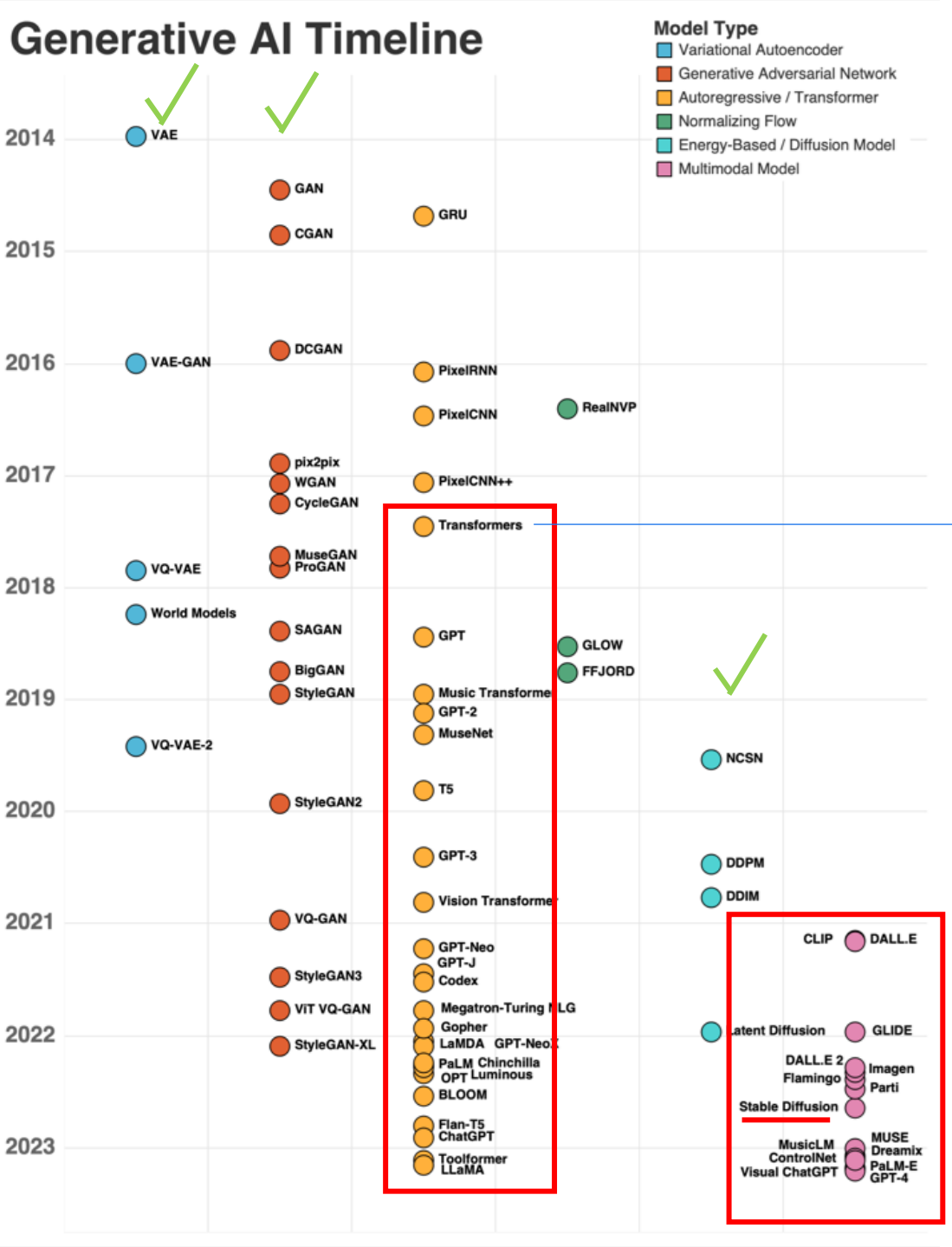
- Aprender una transformación invertible que mapee datos complejos a una distribución simple (como una gaussiana) de **forma exacta y diferenciable**.
- En lugar de aproximar o muestrear, calculan la probabilidad exacta de una imagen gracias a **transformaciones invertibles**.
- Eso los vuelve **elegantes desde el punto de vista teórico**, porque —a diferencia de VAEs, GANs o Diffusion Models— **pueden evaluar log-likelihood exactos.**




## Limitaciones que reducen su relevancia actual

- **Poco expresivos:** las funciones deben ser invertibles, lo que restringe su capacidad de modelar distribuciones muy complejas.
- **Computacionalmente costosos:** cada paso requiere calcular determinantes jacobianos.
- **Escalabilidad limitada:** generan imágenes de calidad inferior a GANs y Diffusion Models.
- **Difusión los superó directamente:**
  Los *Diffusion Models* (especialmente DDPM y DDIM) logran densidades implícitas de alta fidelidad **sin necesidad de invertibilidad ni determinantes.**

Los Normalizing Flows son matemáticamente elegantes, pero menos potentes en la práctica.




Estudiaremos

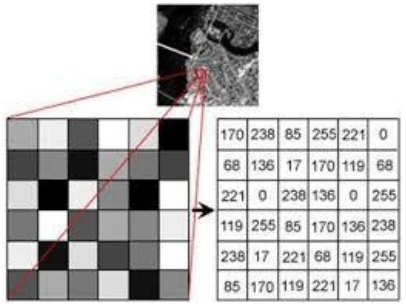
Inicio Presentacion Temario Clas

## Temario

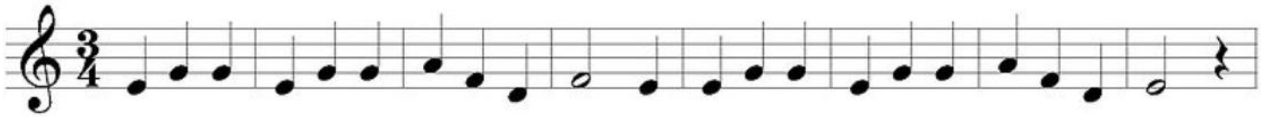
1. Introducción
2. Matemáticas esenciales para el aprendizaje profundo
3. Fundamentos de las redes neuronales profundas
4. Autocodificadores variacionales
5. Redes generativas adversarias (GANs)
6. Modelos basados en energía
7. Modelos de difusión
8. Modelos autoregresivos
9. Modelos de normalización de flujo

# Del dominio de las imágenes al dominio de las secuencias

- **Hasta ahora:** hemos trabajado con **modelos generativos de imágenes**, donde los datos se organizan en **matrices bidimensionales** de píxeles (espacio estructurado, continuo, y localmente correlacionado).




- **Nuevo tipo de información:** las **secuencias** son conjuntos **discretos y ordenados de elementos** (como palabras, notas, fotogramas o mediciones) donde **el orden temporal o posicional** resulta más relevante que la **proximidad espacial**.



- Esto exige **modelos que capturen dependencias temporales o secuenciales** — cada elemento depende de los anteriores.

# Del dominio de las imágenes al dominio de las secuencias

## Cambia el tipo de dependencia que aprenden los modelos

### En imágenes

- El modelo aprende **relaciones espaciales** (formas, texturas, bordes).
- En texto: el modelo aprende **relaciones temporales y semánticas** (gramática, coherencia, contexto).

### En texto

- El modelo aprende **relaciones temporales y semánticas** (gramática, coherencia, contexto).

## La noción de “vecino” cambia:

- En CNNs es un píxel cercano.
- En secuencias es una **palabra anterior o posterior**.

# Modelos Autoregresivos: ideal fundamental

- El **objetivo** de un modelo autoregresivo es **predecir el siguiente elemento de una secuencia** a partir de los anteriores.

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

Modelar la **probabilidad conjunta** de una secuencia completa $(x_1, x_2, \dots, x_T)$.

En lugar de aprender esa distribución directamente (lo cual es complejo), se **factoriza** en un producto de **probabilidades condicionales**.

Cada término $\mathrm{P}(x_t \mid x_1, \dots, x_{t-1})$ representa la **probabilidad de que ocurra el siguiente elemento** dado todo lo que ha ocurrido antes.

Este principio se conoce como **regla de la cadena (chain rule)** en probabilidad.

Durante la generación:

- se comienza con un token inicial $x_1$;
- se predice $x_2$ condicionado en $x_1$;
- luego $x_3$condicionado en $x_1, x_2$ ,y así sucesivamente.

# Modelos Autoregresivos: ideal fundamental

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

Paso 1: inicio

**Entrada:** <BOS>

**Predicción del modelo:** "El"

El modelo estima cuál palabra es más probable para comenzar una oración.

# Modelos Autoregresivos: ideal fundamental

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

Paso 1: inicio

**Entrada:** <BOS>
**Predicción del modelo:** "El"
El modelo estima cuál palabra es más probable para comenzar una oración.

Paso 2:

**Entrada:** <BOS> El
**Predicción del modelo:** "sol"
Ahora el modelo usa el contexto anterior "El" para predecir la siguiente palabra.

# Modelos Autoregresivos: ideal fundamental

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

Paso 1: inicio

**Entrada:** <BOS> → Marca **el inicio de la secuencia textual**.

**Predicción del modelo:** "El"

El modelo estima cuál palabra es más probable para comenzar una oración.

Paso 2:

**Entrada:** <BOS> El

**Predicción del modelo:** "sol"

Ahora el modelo usa el contexto anterior "El" para predecir la siguiente palabra.

Paso 3:

**Entrada:** <BOS> El sol

**Predicción del modelo:** "brilla"

Aprende que tras "El sol", es coherente un verbo como "brilla" o "sale".

# Modelos Autoregresivos: ideal fundamental

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

Paso 1: inicio

**Entrada:** <BOS>
**Predicción del modelo:** "El"
El modelo estima cuál palabra es más probable para comenzar una oración.

Paso 2:

**Entrada:** <BOS> El
**Predicción del modelo:** "sol"
Ahora el modelo usa el contexto anterior "El" para predecir la siguiente palabra.

Paso 3:

**Entrada:** <BOS> El sol
**Predicción del modelo:** "brilla"
Aprende que tras "El sol", es coherente un verbo como "brilla" o "sale".

Paso 4:

**Entrada:** <BOS> El sol brilla
**Predicción del modelo:** "sobre"
Sigue extendiendo la secuencia con una palabra que mantenga la coherencia gramatical.

...

# Modelos Autoregresivos: ideal fundamental

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, ..., x_{t-1})$$

Paso 1: inicio

**Entrada:** <BOS>
**Predicción del modelo:** "El"
El modelo estima cuál palabra es más probable para comenzar una oración.

Paso 2:

**Entrada:** <BOS> El
**Predicción del modelo:** "sol"
Ahora el modelo usa el contexto anterior "El" para predecir la siguiente palabra.

Paso 3:

**Entrada:** <BOS> El sol
**Predicción del modelo:** "brilla"
Aprende que tras "El sol", es coherente un verbo como "brilla" o "sale".

Paso 4:

**Entrada:** <BOS> El sol brilla
**Predicción del modelo:** "sobre"
Sigue extendiendo la secuencia con una palabra que mantenga la coherencia gramatical.

...

El proceso es **autoregresivo** porque cada nueva predicción **depende de sus propias salidas previas**.

# Modelos Autoregresivos: ideal fundamental

- En **texto**
  El modelo predice la siguiente palabra dadas las anteriores → base de los **modelos de lenguaje**.
- En **audio o video**
- En **series temporales**
- En **imágenes secuenciales** (como *PixelRNN*)
- En **danza o movimiento corporal**

# Modelos Autoregresivos: ideal fundamental

- En **texto**
  El modelo predice la siguiente palabra dadas las anteriores → base de los **modelos de lenguaje**.

- En **audio o video**
  Predice el siguiente *sample* o *frame*.

- En **series temporales**
  Predice el valor futuro a partir de los pasados.

- En **imágenes secuenciales** (como *PixelRNN*)
  Genera un píxel a la vez condicionado a los anteriores.

- En **danza o movimiento corporal**
  Predice la **siguiente postura o vector de articulaciones** dadas las posiciones previas → base de los **modelos generativos de movimiento humano** (p. ej., generación coreográfica o animación sintética).

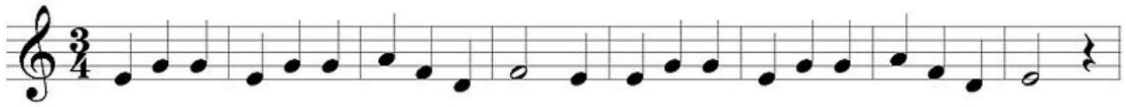
# Modelos Autoregresivos: ideal fundamental

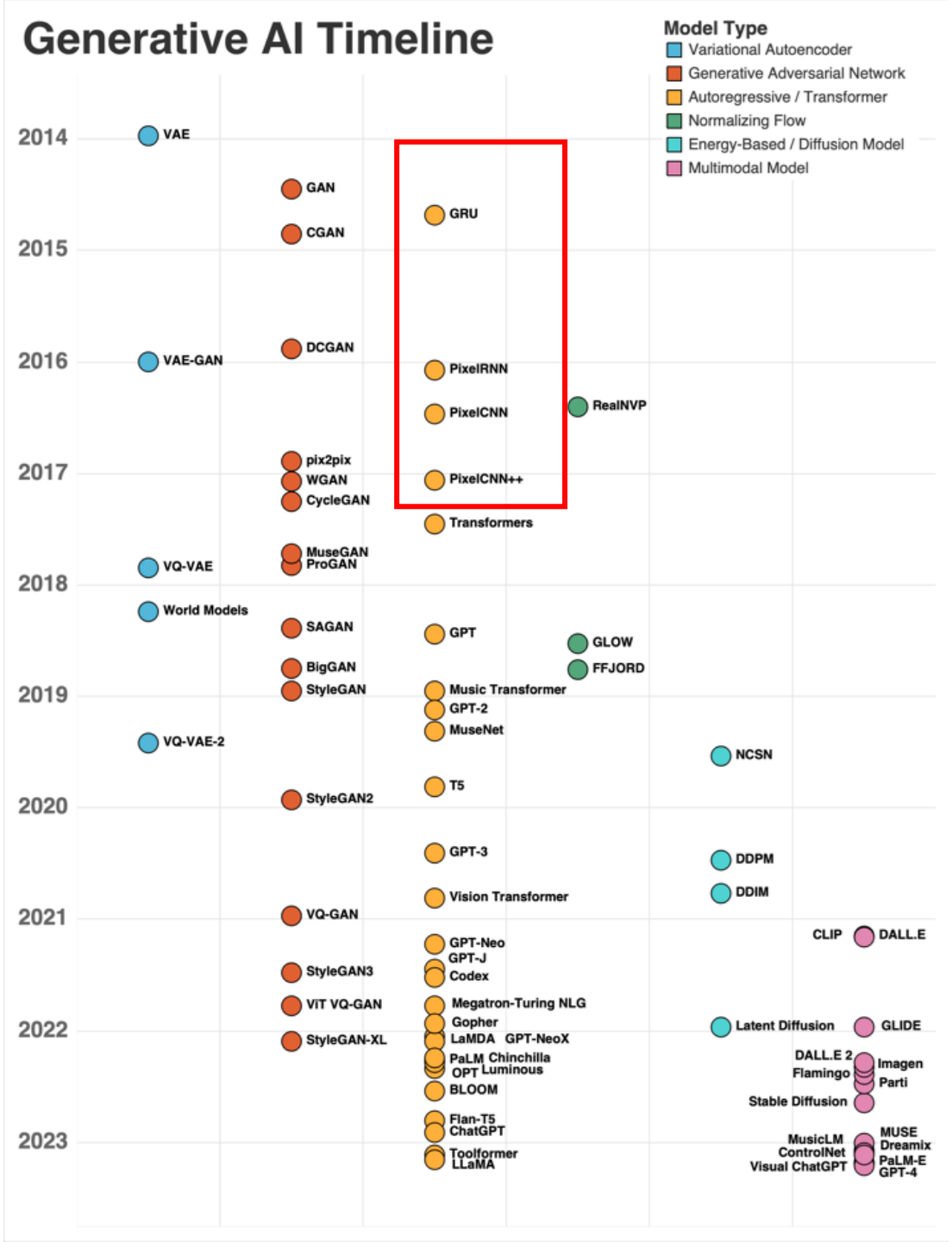
## Consideraciones

- Contexto causal
  La predicción en el paso *t* solo puede usar información de los pasos anteriores (no del futuro).

- Dependencias a largo plazo
  Los modelos simples (como RNN) olvidan el contexto distante; esto motiva arquitecturas más avanzadas.

- Entrenamiento y generación:
  Durante el entrenamiento, el modelo ve secuencias completas.
  Durante la generación, produce un token a la vez de manera iterativa.






# Modelos autoregresivos clásicos: Recurrentes

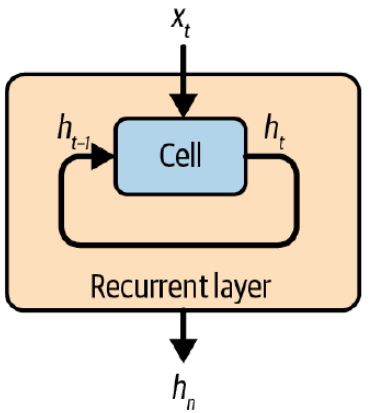
La **lógica autoregresiva**
(predecir lo siguiente dado lo anterior)

se implementa mediante

arquitecturas recurrentes

- La información del pasado se almacena en un **estado oculto** $h_t$.
- El modelo procesa **token por token** (no puede ver toda la secuencia a la vez).

- La dependencia se transmite de forma **secuencial y acumulativa**: $h_t = f(h_{t-1}, x_t)$.

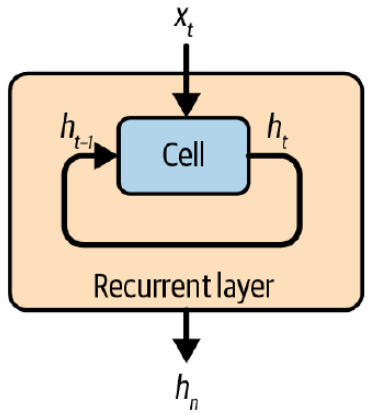



En cada paso $t$:

- El modelo recibe la **entrada actual** $x_t$.
- Combina esa entrada con el **estado oculto anterior** $h_{t-1}$ ,que resume la información del pasado.
- Genera una **salida actual** $h_t$ ,que se usará en el siguiente paso como nuevo estado.

# Limitaciones de los modelos autoregresivos clásicos

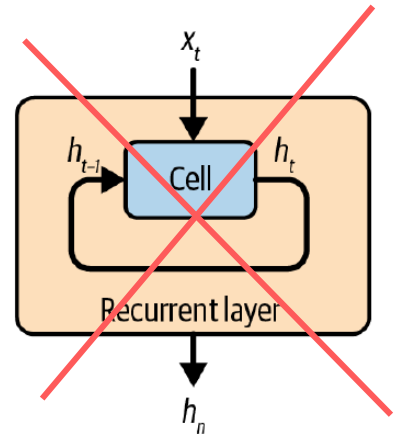
- **Procesamiento secuencial**:
  No puede paralelizar la generación → lento para secuencias largas.
- **Problema del desvanecimiento del gradiente** en RNNs
  Difícil aprender dependencias a largo plazo.
- **Memoria limitada**:
  Solo recuerdan un número finito de pasos anteriores.



# Transformers

Estas limitaciones dieron origen a los **Transformers**, que remplazan la recurrencia por **mecanismos de atención**.

## Durante el entrenamiento




### 1. Procesan toda la secuencia en paralelo

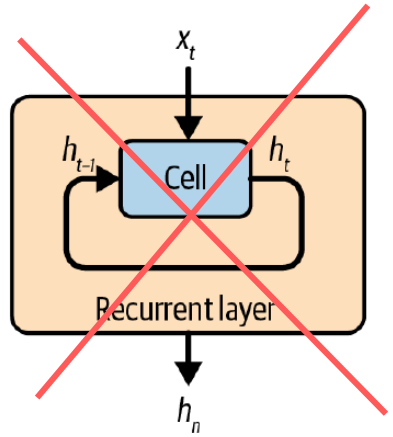
- En una **RNN**, el procesamiento es **estrictamente secuencial**:
  - el paso $t$ necesita el resultado del paso $t-1$ (el estado oculto $h_{t-1}$) antes de poder calcular $h_t$.
    → No se puede calcular $h_2$ sin haber terminado $h_1$.
    → Por eso el entrenamiento no puede paralelizarse fácilmente.
  - Los Transformers ven todos los tokens al mismo tiempo

    "El gato que persiguió el perro estaba cansado."

    → **Todas las posiciones de la secuencia pueden procesarse al mismo tiempo**, porque no dependen de un estado anterior.
    - Cada token **accede directamente** a todos los demás (a través de las matrices de atención).
- Esto permite **aprovechar el cómputo paralelo** y acelerar drásticamente el aprendizaje.

# Transformers

Estas limitaciones dieron origen a los **Transformers**, que remplazan la recurrencia por **mecanismos de atención**.




## 2. Cada token "mira" a los anteriores mediante *autoatención*

- Cada elemento de la secuencia **decide qué partes del contexto son más relevantes**.
- El mecanismo de **self-attention** calcula **qué tan importante** es cada token previo para entender el actual.

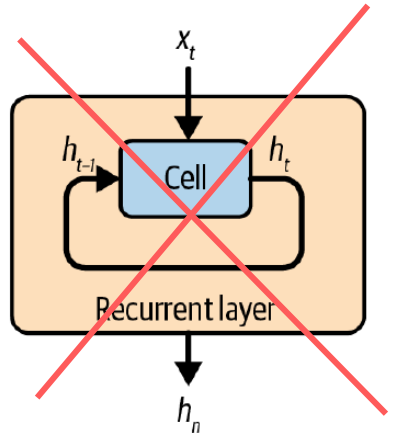
"El gato que persiguió el perro estaba cansado."

Cuando el modelo procesa la palabra "cansado", puede **"mirar" directamente** a "gato"
y asignarle una **alta atención**, porque es el verdadero sujeto de la oración.

- Así, el modelo **aprende dependencias largas o no lineales** que las RNN solían olvidar.

# Transformers

Estas limitaciones dieron origen a los **Transformers**, que remplazan la recurrencia por **mecanismos de atención**.

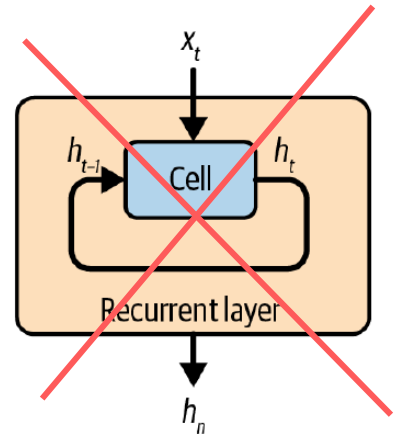



## 3. Mantienen la lógica autoregresiva

- Aunque eliminan la recurrencia, **siguen prediciendo cada token condicionado en los anteriores**.
- Se usa una **máscara causal** para que cada posición solo "vea" los tokens pasados y nunca el futuro.
- Por eso, los Transformers pueden **entrenarse en paralelo**, pero **generan secuencias paso a paso**.

# Transformers

Estas limitaciones dieron origen a los **Transformers**, que remplazan la recurrencia por **mecanismos de atención**.

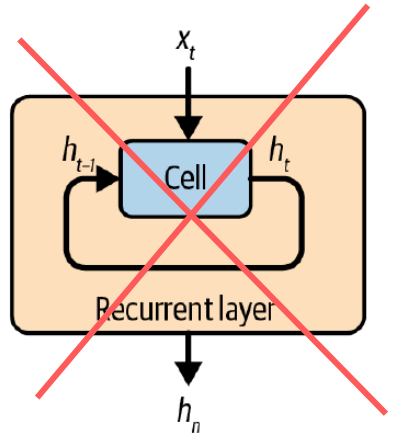



## 4. Memoria global y contexto dinámico

- En lugar de un único vector de estado, cada token **mantiene su propia representación contextual**.
- La atención permite construir una **memoria distribuida**:
  cada token recuerda lo que necesita, sin depender de un único estado acumulado.
- Esto les da **gran capacidad para capturar relaciones semánticas y gramaticales complejas**.

# Transformers

Estas limitaciones dieron origen a los **Transformers**, que remplazan la recurrencia por **mecanismos de atención**.




## 5. Resultado: un cambio de paradigma

- Conservan lo esencial del modelado autoregresivo, pero con una arquitectura **no secuencial, paralela y escalable**.
- Este diseño permitió modelos mucho más grandes y contextualmente potentes, como **GPT, BERT, o DALL·E**.

# Transformers

## Ventajas

- Paralelización total:
  Todos los tokens se procesan simultáneamente.

- Memoria contextual extendida:
  El modelo puede considerar toda la secuencia pasada.

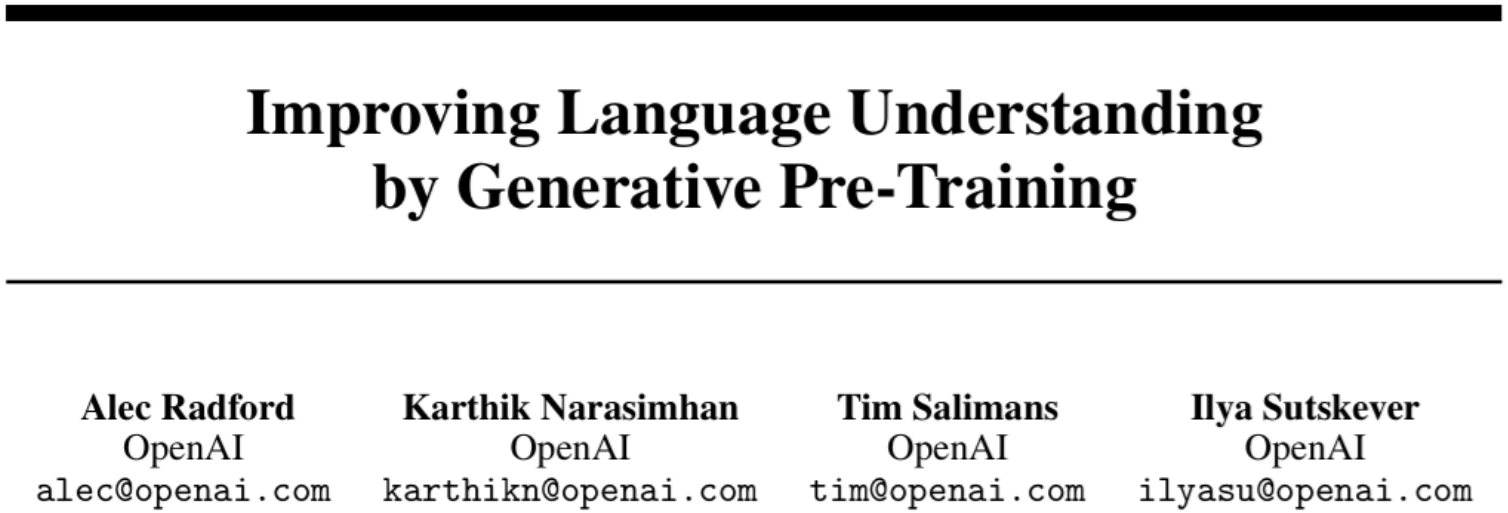
- Escalabilidad:
  Se adapta fácilmente a secuencias largas y grandes volúmenes de datos.

- Versatilidad:
  La misma arquitectura se aplica a texto, audio, video e incluso imágenes.

# GPT: Generative Pre-trained Transformer

- Presentado por **OpenAI en junio de 2018**, un año después del paper *"Attention Is All You Need"*.

- Primer modelo en aplicar la **arquitectura Transformer** al **modelado del lenguaje autoregresivo**.



**Improving Language Understanding by Generative Pre-Training**


**Alec Radford** OpenAI alec@openai.com

**Karthik Narasimhan** OpenAI karthikn@openai.com

**Tim Salimans** OpenAI tim@openai.com

**Ilya Sutskever** OpenAI ilyasu@openai.com

# GPT: Generative Pre-trained Transformer

- Entrenar un Transformer en **gran cantidad de texto** para **predecir la siguiente palabra** dada la secuencia anterior.

El entrenamiento previo se conoce como **pre-entrenamiento generativo (Generative Pre-Training)**.

- **Dataset: BookCorpus**, con **4.5 GB de texto** de más de **7,000 libros inéditos** de distintos géneros.
- **Objetivo:** Enseñar al modelo a **comprender contexto y coherencia** a nivel de palabra, oración y párrafo.

# GPT: Generative Pre-trained Transformer

## Fine-tuning o ajuste fino

- Tras el pre-entrenamiento general, el modelo puede **ajustarse a tareas específicas** usando datasets más pequeños.

- Este proceso **refina los pesos** del modelo para adaptarlo a tareas como:

    - Clasificación de texto
    - Análisis de similitud semántica
    - Respuesta a preguntas (*question answering*)

# GPT: Generative Pre-trained Transformer

## Evolución del modelo

- OpenAI amplió la arquitectura con modelos posteriores: **GPT-2, GPT-3, GPT-3.5, GPT-4, GPT-5** ...

Cada versión aumenta:

El **tamaño del modelo** (más parámetros).
La **cantidad y diversidad de datos**.
La **capacidad para generar texto coherente, creativo y contextualizado**.

Han transformado el campo del **procesamiento del lenguaje natural (NLP)** en investigación e industria.

# GPT: Generative Pre-trained Transformer

Construiremos una **versión reducida de GPT**,
usando menos datos pero aplicando los **mismos principios fundamentales**:
atención causal, entrenamiento autoregresivo y fine-tuning.

kaggle

## Wine Reviews

130k wine reviews with variety, location, winery, price, and description

**Número de registros:** ~130,000 reseñas.

**Columnas principales:**

- `country` : país de origen del vino.
- `description` : texto con la reseña o comentario del vino.
- `designation` : nombre o etiqueta específica del vino.
- `points` : puntuación otorgada (normalmente de 80 a 100).
- `price` : precio en dólares (cuando está disponible).
- `province` : región dentro del país.
- `region_1` / `region_2` : subregiones vitivinícolas.
- `variety` : tipo de uva (Cabernet Sauvignon, Chardonnay, etc.).
- `winery` : nombre de la bodega productora.

# Trabajo con texto

De lo continuo a lo discreto: cómo cambia el enfoque de modelado

## En sus inicios, los modelos generativos profundos se centraron en imágenes

- El dominio visual ofrecía **ventajas computacionales y conceptuales**:
    - Datos **continuos** (píxeles) → fáciles de optimizar con gradientes.
    - Estructura **espacial** simple (2D) → adecuada para CNNs y U-Nets.
- Las redes convolucionales y modelos como **GANs, VAEs y Diffusion Models** lograron resultados sorprendentes en generación de imágenes.

## El texto presentaba retos mucho más complejos

- Hasta hace pocos años, incluso los modelos más avanzados (como LSTM o GRU) **no lograban mantener coherencia en textos largos**.

# Trabajo con texto

De lo continuo a lo discreto: cómo cambia el enfoque de modelado

## Naturaleza discreta del texto

- El texto está formado por **unidades discretas** (caracteres o palabras), no por valores continuos como los píxeles.
- En imágenes, podemos modificar gradualmente un color (verde → azul), pero en texto **no existe una forma continua de pasar de "cat" a "dog"**.
- Por ello, **el gradiente no puede aplicarse directamente** sobre palabras → necesitamos **representaciones numéricas** (*embeddings*).

## Dimensión temporal y orden secuencial

- El texto tiene una **dimensión temporal (orden de aparición)**, no espacial.
- El **orden importa**: invertir las palabras cambia completamente el significado.
- A diferencia de las imágenes (que pueden rotarse o reflejarse sin perder sentido), en texto, el contexto depende de **la secuencia exacta** de palabras.
- Existen **dependencias a largo plazo** (p. ej., una pregunta y su respuesta o un pronombre que retoma un sujeto anterior).

# Trabajo con texto

De lo continuo a lo discreto: cómo cambia el enfoque de modelado

## Alta sensibilidad semántica

- En texto, **pequeños cambios** (una palabra o incluso una letra)
  pueden **alterar por completo el significado**
  o volverlo incoherente.
- En cambio, una imagen sigue siendo reconocible aunque cambien algunos píxeles.
- Por eso, **generar texto coherente es mucho más difícil**:
  cada palabra afecta la interpretación global.

## Estructura gramatical y reglas semánticas

- El lenguaje tiene **reglas gramaticales y semánticas** que deben cumplirse simultáneamente.
- El modelo debe aprender **no solo la sintaxis**, sino también **el significado contextual**,
  algo que no existe en los datos de imagen.

# Trabajo con texto

De lo continuo a lo discreto: cómo cambia el enfoque de modelado

## Implicación para el modelado

- Las redes que trabajan con texto deben aprender a manejar:
  - Datos discretos → embeddings continuos.
  - Dependencias temporales → mecanismos de memoria o atención.
  - Sensibilidad semántica → coherencia global en la generación.

# Trabajo con texto

Preprocesamiento

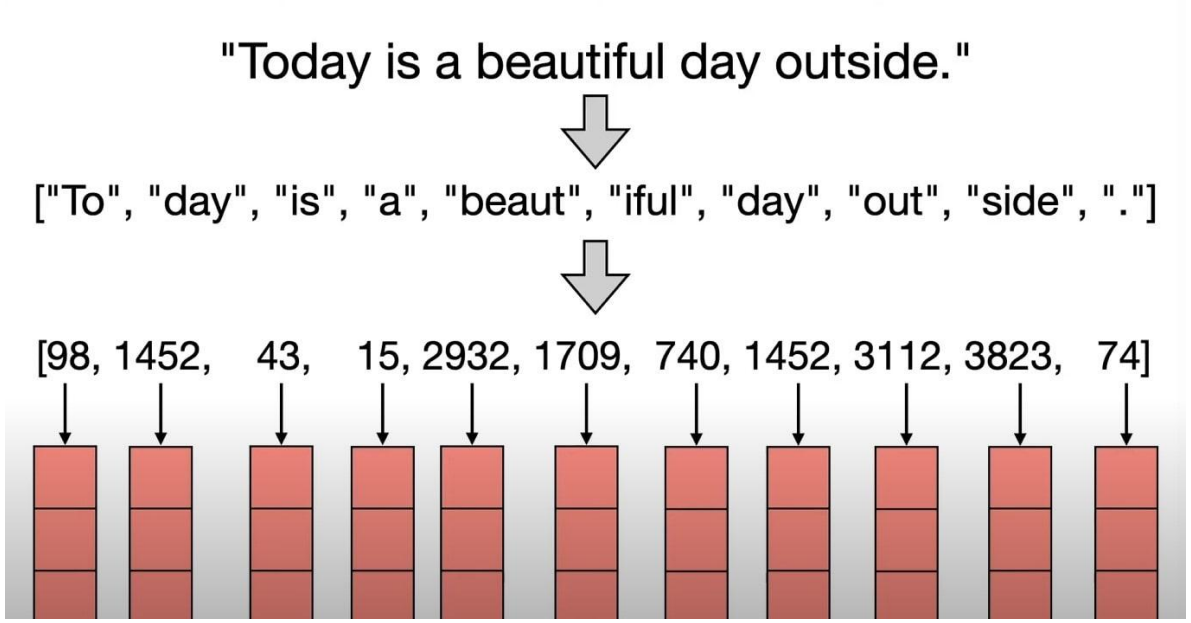
## Tokenización

- Es el **primer paso en el procesamiento del texto**:

  Dividirlo en unidades más pequeñas llamadas **tokens** (palabras, subpalabras o caracteres).

- Permite **convertir texto** en una forma que el modelo pueda representar numéricamente.

- La estrategia de tokenización **depende del tipo de modelo y del objetivo del entrenamiento**.



# Trabajo con texto

## Preprocesamiento

### Tokenización por palabras

- Cada palabra se convierte en un token independiente.

**Ventajas:**

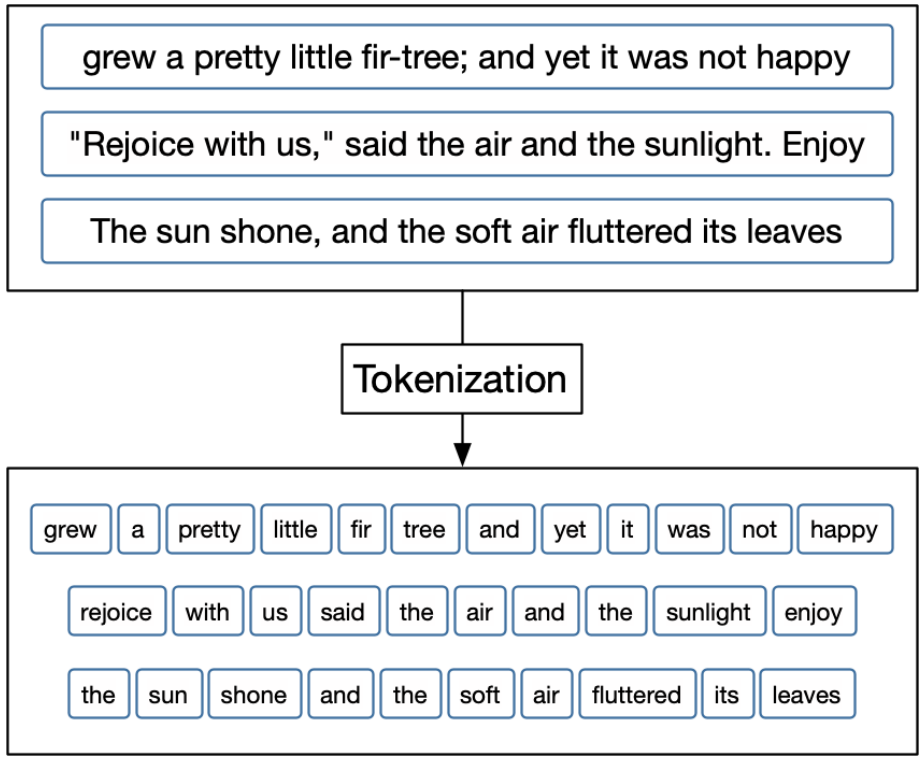
- Mantiene el significado semántico completo de cada palabra.

**Desventajas:**

- Vocabulario grande → más parámetros que aprender.
- Palabras raras o únicas deben reemplazarse por un token especial (UNK o *unknown word*).
- No puede generar palabras nuevas fuera del vocabulario de entrenamiento.

**Buenas prácticas:**

- Convertir a **minúsculas** (salvo nombres propios).
- **Stemming** opcional: reducir palabras a su raíz (*browsing*, *browsed* → *brows*).
- Decidir si conservar o eliminar **puntuación** (según el tipo de texto).



# Trabajo con texto

## Preprocesamiento

### Tokenización por caracteres

- Cada carácter (letra, signo, espacio) es un token.

Ventajas:

- **Vocabulario pequeño** → entrenamiento más rápido.
- Puede **inventar palabras nuevas** combinando caracteres.

Desventajas:

- Pierde significado semántico a nivel palabra.
- Requiere **secuencias más largas** y entrenamiento más extenso.

Decisiones típicas:

- Usar todo en minúsculas o mantener mayúsculas como tokens separados.

### Elección para nuestro modelo

- Se utilizará **tokenización por palabras en minúsculas**,
  sin *stemming*,
  y **con puntuación incluida** (para que el modelo aprenda cuándo usar comas o puntos).

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

1. **Cargar los datos** y crear una lista de cadenas de texto que contengan las **descripciones de cada vino**.

```
#Subir el archivo kaggle.json
from google.colab import files
files.upload()

# Crear la carpeta .kaggle y mover el archivo allí
!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/

# Cambiar los permisos (muy importante)
!chmod 600 ~/.kaggle/kaggle.json

# Instalar Kaggle CLI (por si no está instalada)
!pip install -q kaggle

# Descargar el dataset de Wine Reviews
!kaggle datasets download -d zynicide/wine-reviews

# Descomprimir el archivo descargado
!unzip -q wine-reviews.zip
```

```
# Cargar el dataset completo}
with open("winemag-data-130k-v2.json") as json_data:
    wine_data = json.load(json_data)

wine_data[10]
```

```
{'points': '87',
 'title': 'Kirkland Signature 2011 Mountain Cuvée Cabernet Sauvignon (Napa Valley)',
 'description': 'Soft, supple plum envelopes an oaky structure in this Cabernet, supp
 'taster_name': 'Virginie Boone',
 'taster_twitter_handle': '@vboone',
 'price': 19,
 'designation': 'Mountain Cuvée',
 'variety': 'Cabernet Sauvignon',
 'region_1': 'Napa Valley',
 'region_2': 'Napa',
 'province': 'California',
 'country': 'US',
 'winery': 'Kirkland Signature'}
```

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

```python
# Filtrar el dataset
filtered_data = [
    "wine review : "
    + x["country"]
    + " : "
    + x["province"]
    + " : "
    + x["variety"]
    + " : "
    + x["description"]
    for x in wine_data
    if x["country"] is not None
    and x["province"] is not None
    and x["variety"] is not None
    and x["description"] is not None
]
```

1

Recorre **todas las reseñas** del dataset.

2

Filtra reseñas incompletas

La condición `if ... is not None` asegura que **solo se incluyan las reseñas que tienen todos los campos necesarios:**

- `country`
- `province`
- `variety`
- `description`

Elimina entradas con datos faltantes.

3

Para cada reseña válida, crea una **cadena unificada** con formato:

`"wine review : [country] : [province] : [variety] : [description]"`

"wine review : France : Burgundy : Pinot Noir : This wine has notes of cherry and spice with a long finish."

`filtered_data` se convierte en una **lista de textos planos**, donde cada elemento combina metadatos (país, región, variedad) y descripción.

```
[
  "wine review : Italy : Tuscany : Sangiovese : A rich red with cherry and herbal notes.",
  "wine review : USA : Napa Valley : Cabernet Sauvignon : Dense and layered with blackberry and cocoa.",
  ...
]
```

gpt.ipynb

Google colab

# Trabajo con texto

## Preprocesamiento

```
# Cuenta las reseñas de vinos
n_wines = len(filtered_data)
print(f"{n_wines} reseñas cargadas")
```

```
129907 recipes loaded
```

```
example = filtered_data[25]
print(example)
```

```
wine review : US : California : Pinot Noir : Oak and earth intermingle around robust aromas
```

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

2. Tokenizar los datos

```
# Separar la puntuación con espacios, para tratarla como si fuera una palabra
# independiente.
def pad_punctuation(s):

    s = re.sub(f"([{string.punctuation}, '\n'])", r" \1 ", s)
    s = re.sub(" +", " ", s)
    return s
text_data = [pad_punctuation(x) for x in filtered_data]
```

- Usa una expresión regular para buscar todos los caracteres de puntuación definidos en string.punctuation
- Luego añade un espacio antes y después de cada signo de puntuación encontrado.

Reemplaza múltiples espacios consecutivos por un solo espacio, para dejar el texto limpio y uniforme.

Para cada elemento x en filtered_data, aplica la función pad_punctuation(x) y guarda el resultado en una nueva lista llamada text_data.

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

2. Tokenizar los datos

```
# Separar la puntuación con espacios, para tratarla como si fuera una palabra
# independiente.
def pad_punctuation(s):

    s = re.sub(f"([{string.punctuation}, '\n'])", r" \1 ", s)
    s = re.sub(" +", " ", s)
    return s
text_data = [pad_punctuation(x) for x in filtered_data]
```

- Usa una expresión regular para buscar todos los caracteres de puntuación definidos en string.punctuation
- Luego añade un espacio antes y después de cada signo de puntuación encontrado.

Reemplaza múltiples espacios consecutivos por un solo espacio, para dejar el texto limpio y uniforme.

Para cada elemento x en filtered_data, aplica la función pad_punctuation(x) y guarda el resultado en una nueva lista llamada text_data.

```
# Muestra el ejemplo de una reseña
example_data = text_data[25]
example_data
```

```
'wine review : US : California : Pinot Noir : Oak and earth intermingle around robust aromas of wet forest floor
steeped in smoky spice and smooth texture . '
```

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

```
# Lo convierte a un Tensorflow Dataset
text_ds = (
    tf.data.Dataset.from_tensor_slices(text_data)
    .batch(BATCH_SIZE)
    .shuffle(1000)
)
```

Convierte la lista `text_data` en un **objeto `tf.data.Dataset`,** es decir, una **estructura de datos optimizada** para el entrenamiento en TensorFlow.
Cada elemento del dataset será **una reseña de vino individual** (una cadena de texto).

Agrupa los ejemplos en **lotes (batches)** de tamaño `BATCH_SIZE`

Mezcla aleatoriamente los ejemplos antes de cada época de entrenamiento..
El argumento `1000` indica el **tamaño del buffer de barajado**:
TensorFlow mantiene un buffer de 1000 ejemplos y los mezcla continuamente.
Esto evita que el modelo aprenda un **orden artificial** de los datos.

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

La capa `TextVectorization` realiza **automáticamente el preprocesamiento del texto** antes de entrar al modelo.

1. **Estandariza (limpia) el texto.**
2. **Tokeniza** (divide en palabras o subpalabras).
3. **Convierte cada token en un número entero** según un vocabulario aprendido.

```python
# Crea una capa de vectorización
vectorize_layer = layers.TextVectorization(
    standardize="lower",
    max_tokens=VOCAB_SIZE,
    output_mode="int",
    output_sequence_length=MAX_LEN + 1,
)
```

Convierte **todo el texto a minúsculas**, sin **eliminar puntuación.**

Define el **tamaño máximo del vocabulario**. Solo se conservarán las VOCAB_SIZE palabras más frecuentes del corpus. Cualquier palabra fuera del vocabulario se reemplaza por un **token especial "<UNK>" (unknown)**.

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

La capa `TextVectorization` realiza **automáticamente el preprocesamiento del texto** antes de entrar al modelo.

1. **Estandariza (limpia) el texto.**
2. **Tokeniza** (divide en palabras o subpalabras).
3. **Convierte cada token en un número entero** según un vocabulario aprendido.

```python
# Crea una capa de vectorización
vectorize_layer = layers.TextVectorization(
    standardize="lower",
    max_tokens=VOCAB_SIZE,
    output_mode="int",
    output_sequence_length=MAX_LEN + 1,
)
```

Indica que la salida de la capa debe ser una **secuencia de enteros**, donde cada entero representa un token en el vocabulario.

"red wine dry" → [45, 102, 678]

Controla la **longitud fija** de las secuencias numéricas de salida.
Si una oración es más corta, se **rellena con ceros (padding)**.
Si es más larga, se **recorta (trunca)** al máximo especificado.
El +1 suele usarse porque se necesita una posición extra para el token de desplazamiento en la predicción siguiente.

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

```
# Adapta la capa al conjunto de entrenamiento
vectorize_layer.adapt(text_ds)
vocab = vectorize_layer.get_vocabulary()
```

"Entrena" la capa de vectorización sobre el dataset de texto `text_ds`.

Lo que hace es **recorrer todos los textos** y:

- Identificar las palabras (o tokens) que aparecen.
- Contar su frecuencia.
- Construir un **vocabulario ordenado por frecuencia**, limitado a `max_tokens` (el que se definió al crear la capa).

**Aprende qué palabras existen y cómo indexarlas**.

Por ejemplo, internamente crea algo como:

```
{"<PAD>": 0, "<UNK>": 1, "wine": 2, "red": 3, "fruit": 4, "tannin": 5, ...}
```

Después de este paso, la capa `vectorize_layer` ya sabe **cómo convertir texto a números**.

gpt.ipynb

Google colab

# Trabajo con texto

## Preprocesamiento

```
# Adapta la capa al conjunto de entrenamiento
vectorize_layer.adapt(text_ds)
vocab = vectorize_layer.get_vocabulary()
```

Extrae el **vocabulario aprendido** por la capa, como una lista de strings.

El índice de cada palabra en la lista corresponde al número entero que usará el modelo.

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

```
# Adapta la capa al conjunto de entrenamiento
vectorize_layer.adapt(text_ds)
vocab = vectorize_layer.get_vocabulary()
```

Extrae el **vocabulario aprendido** por la capa, como una lista de strings.

El índice de cada palabra en la lista corresponde al número entero que usará el modelo.

```
# Visualizar parte del vocabulario aprendido: mapeos de palabras
for i, word in enumerate(vocab[:10]):
    print(f"{i}: {word}")
```

```
0:
1: [UNK]
2: :
3: ,
4: .
5: and
6: the
7: wine
8: a
9: of
```

**vocab**

- Es la lista de palabras (tokens) que la capa `TextVectorization` aprendió al ejecutar:
- Selecciona **solo los primeros 10 elementos** del vocabulario, para mostrar un resumen en lugar de toda la lista completa (que puede tener miles de palabras).
- Ordenados por mayor frecuencia.

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

```
# Despliega el mismo ejemplo convertido en enteros
example_tokenised = vectorize_layer(example_data)
print(example_tokenised.numpy())
```

```
[   7   10    2   20    2   29    2   43   62    2   55    5  243 4145
  453  634   26    9  497  499  667   17   12  142   14 2214   43   25
 2484   32    8  223   14 2213  948    4  594   17  987    3   15   75
  237    3   64   14   82   97    5   74 2633   17  198   49    5  125
   77    4    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0]
```

¿Cómo interpretamos el 7?

Recordemos que:

```
# Muestra el ejemplo de una reseña
example_data = text_data[25]
example_data
```

```
'wine review : US : California : Pinot Noir : Oak and earth intermingle around
steeped in smoky spice and smooth texture . '
```

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

### 3. Crear el conjunto de entrenamiento

```
# Crear el conjunto de entrenamiento con las reseñas y el mismo texto desplazado
#una palabra hacia adelante.

def prepare_inputs(text):
    text = tf.expand_dims(text, -1)
    tokenized_sentences = vectorize_layer(text)
    x = tokenized_sentences[:, :-1]
    y = tokenized_sentences[:, 1:]
    return x, y
```

- Aumenta una dimensión al final del tensor de texto.
- Esto convierte el texto en la forma esperada por TextVectorization, que requiere una entrada con al menos 2 dimensiones ([batch, 1]).

Aplica la capa de vectorización que **convierte el texto en una secuencia de enteros** (tokens).

Crea la **entrada del modelo**: todos los tokens **excepto el último**.

Crea la **salida objetivo** (*target*): los mismos tokens pero **desplazados una posición hacia adelante**.

```
train_ds = text_ds.map(prepare_inputs)
```

- Aplica la función prepare_inputs() a cada elemento del dataset text_ds.
- El resultado train_ds es un **TensorFlow Dataset** donde cada elemento es una tupla (x, y) de tensores numéricos listos para model.fit().

Por ejemplo:

Si el texto es: "wine review : France : Burgundy"

Después del vectorizado: [10, 15, 23, 9, 23, 0]

Entonces:

x = [10, 15, 23, 9, 23]
y = [15, 23, 9, 23, 0]

El modelo aprende:

10 → 15
10,15 → 23
10,15,23 → 9
...

gpt.ipynb

# Trabajo con texto

## Preprocesamiento

```
example_input_output = train_ds.take(1).get_single_element()
```

- Toma **solo el primer elemento** del dataset (es decir, un *batch*).
- Si BATCH_SIZE = 64, ese elemento contendrá 64 pares (x, y).

```
# Ejemplo de entrada
example_input_output[0][0]
```

Muestra la **primera secuencia de entrada numérica** del lote, es decir, una lista de índices enteros que representan los tokens de una reseña de vino después de la vectorización y el padding.

```
<tf.Tensor: shape=(80,), dtype=int64, numpy=
array([   7,   10,    2,  152,    2,  479,    2,  479,    2,    8, 1680,
         27,   32,  387,   14,   63,  479,  412,  600,   51,  519,   51,
       2880,  331,  824, 2847,  127,  971,  117,    4,    6, 1021,   13,
          8,  949, 1255,   63,    3,   11,  361, 1882,   11,    8,   78,
          5,  137,  110,    4,   15,   13,   72,    3,    6,  539,  361,
          8, 2208,    4,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0])>
```

```
# Ejemplo de salida (desplazado por un token)
example_input_output[1][0]
```

```
<tf.Tensor: shape=(80,), dtype=int64, numpy=
array([  10,    2,  152,    2,  479,    2,  479,    2,    8, 1680,   27,
         32,  387,   14,   63,  479,  412,  600,   51,  519,   51, 2880,
        331,  824, 2847,  127,  971,  117,    4,    6, 1021,   13,    8,
        949, 1255,   63,    3,   11,  361, 1882,   11,    8,   78,    5,
        137,  110,    4,   15,   13,   72,    3,    6,  539,  361,    8,
       2208,    4,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
          0,    0,    0])>
```

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

- Al generar una palabra, **no consideramos todas las palabras previas por igual**.
- Algunas son **cruciales para el significado**, otras solo aportan forma gramatical.
- Ejemplo:

The pink elephant tried to get into the car but it was too …"

¿Qué palabra esperamos al final?

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

- Al generar una palabra, **no consideramos todas las palabras previas por igual**.
- Algunas son **cruciales para el significado**, otras solo aportan forma gramatical.
- Ejemplo:

The pink elephant tried to get into the car but it was too …"

¿Qué palabra esperamos al final?

- Claramente, la siguiente palabra debería ser **"big"** o algo con significado similar.
- Pero **¿cómo lo sabemos?**
- Depende de **qué palabras de la oración estamos "atendiendo"**.

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

- Al generar una palabra, **no consideramos todas las palabras previas por igual**.
- Algunas son **cruciales para el significado**, otras solo aportan forma gramatical.
- Ejemplo:

The pink elephant tried to get into the car but it was too …"

¿Qué palabra esperamos al final?

Palabras que **sí son relevantes**:

Palabras que **no son relevantes**:

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

- Al generar una palabra, **no consideramos todas las palabras previas por igual**.
- Algunas son **cruciales para el significado**, otras solo aportan forma gramatical.
- Ejemplo:

The pink elephant tried to get into the car but it was too …"

¿Qué palabra esperamos al final?

Palabras que **sí son relevantes**:

- **elephant** → nos indica que el sujeto es grande, no pequeño.
- **car** → nos sugiere que el problema tiene que ver con el tamaño o el espacio.
- **tried to get into** → implica que alguien está intentando entrar en algo.

De este contexto, inferimos que el elefante **no cabe en el coche**, por lo tanto la palabra **"big"** tiene más probabilidad que otras.

Palabras que **no son relevantes**:

- **pink** → un detalle descriptivo sin impacto en el problema de tamaño.
- **the, but, it** → palabras funcionales necesarias para la gramática, pero **no aportan significado semántico** para elegir la palabra final.

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

Y si cambiáramos de contexto y la frase fuera:

"The pink elephant tried to get into the *swimming pool* but it was too ..."

¿Qué palabra esperamos al final?

# Atención

## El core de los transformers

Intuición:

¿cómo los humanos prestamos atención al escribir?

Y si cambiáramos de contexto y la frase fuera:

"The pink elephant tried to get into the *swimming pool* but it was too …"

¿Qué palabra esperamos al final?

Podríamos elegir **"scared"**

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

Y si cambiáramos de contexto y la frase fuera:

"The pink elephant tried to squash the car but it was too ..."

¿Qué palabra esperamos al final?

# Atención

## El core de los transformers

### Intuición:

¿cómo los humanos prestamos atención al escribir?

Y si cambiáramos de contexto y la frase fuera:

"The pink elephant tried to squash the car but it was too ..."

¿Qué palabra esperamos al final?

Ahora el sujeto de *it* sería el **car**, y la palabra más natural podría ser **"fast"** o **"strong"**.

# Atención

## El core de los transformers

Intuición:

¿cómo los humanos prestamos atención al escribir?

Y si cambiáramos de contexto y la frase fuera:

"The pink elephant tried to squash the car but it was too ..."

¿Qué palabra esperamos al final?

Ahora el sujeto de *it* sería el **car**, y la palabra más natural podría ser **"fast"** o **"strong"**.

En cada caso, **solo algunas palabras del contexto** son determinantes para elegir la siguiente.

El mecanismo de atención (también llamado **cabeza de atención**)permite que un modelo, igual que nosotros,

**"mire" las palabras relevantes** e **"ignore" las que no aportan información**.