

# **Sinergia entre Redes GAN y Transformers: Avances y Aplicaciones en la Generación de Imágenes**

Lorenzo Martínez Agudo

Fecha: Septiembre 2024

# Resumen

Dentro del campo de la Inteligencia Artificial, la generación de imágenes es una parte muy desarrollada y relevante. La capacidad de crear imágenes convincentes y de alta calidad a partir de datos existentes se ha convertido en un desafío cuyo resultado se palpa en aplicaciones de diversas industrias, como la publicidad, el diseño o el arte. Este trabajo se enfoca en la generación de imágenes de 10 clases diferentes utilizando primero un tipo de red neuronal, la red neuronal GAN, y posteriormente, esta misma mezclada con un transformer sobre dichas redes se han sentado las bases para lograr resultados óptimos. El objetivo principal es la implementación, entrenamiento y comparación de estas dos redes neuronales para generar imágenes similares a las del conjunto de datos público. Estos modelos se describen detalladamente, explicando sus arquitecturas, fundamentos teóricos y técnicas específicas aplicadas en sus entrenamientos. Como cada modelo presenta unas características y enfoques particulares, se evaluará su rendimiento para determinar cuál de ellos es más adecuado para la generación de imágenes sintéticas, mostrando sus ventajas y sus limitaciones, explicando cuáles son las aplicaciones que pueden tener estas redes y su posible impacto a futuro. Si el entrenamiento es adecuado debe arrojar como resultado una mejora en la segunda red sobre la primera. La implementación de los dos modelos se lleva a cabo mediante el uso de alguna de las librerías de programación más relevantes de Python en este área que son: Tensorflow, Keras y Pytorch. Para poder generar las imágenes se emplea un repositorio de datos público que contiene imágenes ,CIFAR-10, las cuales se preprocesan para un mejor entrenamiento. Para comprobar de manera objetiva que las imágenes generadas por los modelos se consideran imágenes convincentes se ha utilizado la métrica FID, siendo los resultados obtenidos favorables a lo que se esperaba a nivel teórico.

# Índice

|   |           |
|---|-----------|
| <b>1. Introducción</b>  | <b>3</b>  |
| <b>2. Indice de figuras</b>                                     | <b>4</b>  |
| <b>3. Estructura del TFG</b>                                    | <b>5</b>  |
| <b>4. Marco teórico</b>   | <b>6</b>  |
| 4.1. Introduccion a las redes neuronales artificiales . . . . . | 6         |
| 4.2. Red neuronal multicapa . . . . .                           | 8         |
| 4.3. Backpropagation . . . . .                                  | 8         |
| 4.3.1. Funcionamiento del backpropagation . . . . .             | 9         |
| 4.4. Tensores . . . . .   | 11        |
| 4.5. Redes Generativas Adversariales(GAN) . . . . .             | 11        |
| 4.5.1. Arquitectura y Funcionamiento de la GAN . . . . .        | 12        |
| 4.5.2. Función objetivo . . . . .                               | 14        |
| 4.5.3. Optimización de los parametros . . . . .                 | 15        |
| 4.5.4. Técnicas de mejora . . . . .                             | 16        |
| 4.6. Transformers . . . . .                                     | 18        |
| 4.6.1. Arquitectura de un Transformer . . . . .                 | 18        |
| 4.6.2. Encoder . . . . .  | 19        |
| 4.6.3. Decoder . . . . .  | 21        |
| <b>5. Planteamiento del problema</b>                            | <b>25</b> |
| <b>6. Solución propuesta y resultados</b>                       | <b>28</b> |
| 6.1. Conjunto de datos y preprocessado . . . . .                | 28        |
| 6.2. Implementacion de las GAN . . . . .                        | 30        |
| 6.2.1. Preprocesado de datos la red GAN . . . . .               | 30        |
| 6.2.2. Arquitectura e hiperparámetros . . . . .                 | 30        |
| 6.2.3. Entrenamiento y optimización . . . . .                   | 38        |
| 6.3. Conclusiones . . . . .                                     | 44        |
| 6.3.1. Uso de técnicas de mejora . . . . .                      | 45        |
| 6.3.2. Conclusiones una vez hecha la mejora . . . . .           | 47        |
| 6.3.3. Conclusiones del uso de tecnicas de mejora . . . . .     | 53        |
| 6.4. Implementacion de las GAN con los transformers . . . . .   | 54        |
| 6.4.1. Arquitectura de un Transformer . . . . .                 | 54        |
| 6.4.2. Transformer en las redes GAN . . . . .                   | 55        |
| 6.4.3. Conclusiones . . . . .                                   | 58        |
| <b>7. Conclusiones</b>  | <b>59</b> |
| <b>8. Lineas futuras de investigación</b>                       | <b>60</b> |
| <b>9. Bibliografía</b>  | <b>61</b> |

# 1. Introducción

En los últimos años, la inteligencia artificial ha experimentado muchos avances, especialmente en el campo del aprendizaje profundo o deep learning. Entre las técnicas más innovadoras se encuentran las Redes Generativas Antagónicas o también conocidas como GAN, y los Transformers, ambas reconocidas por su capacidad para aprender y generar datos. Las GANs significaron un gran avance en la IA por la generación de imágenes, permitiendo crear contenido visual que imita con gran realismo imágenes del mundo real llegando a tal punto que no se puedan diferenciar de las reales. Por otra parte, los transformers, originalmente diseñados para tareas de procesamiento de lenguaje natural, han demostrado ser extraordinariamente versátiles y efectivos en diversas aplicaciones, incluyendo la generación de imágenes y es de esta sinergia de lo que va a tratar mi tfg.

El propósito de este trabajo es explorar la sinergia entre GANs y Transformers, combinando ambas redes para mejorar las redes de generación de imágenes. Mientras que las GANs son extremadamente efectivas en la generación de imágenes de alta calidad, presentan dificultades en cuanto a la estabilidad del entrenamiento y en la mayoría de los casos las imágenes generadas si bien son buenas no son óptimas y existen diferencias leves. Por otra parte, los Transformers, con su capacidad para modelar dependencias a largo plazo y capturar relaciones complejas dentro de los datos, ofrecen una prometedora mejora en estos aspectos.

La motivación detrás de este proyecto radica en la creciente demanda de datos sintéticos de alta calidad en diversas áreas, tales como la investigación científica o los estudios de mercado entre otros. Generar datos realistas y coherentes puede reducir costos y tiempos asociados con la recolección de datos reales, ofreciendo una alternativa viable en situaciones donde estos son difíciles o costosos de obtener.

El principal objetivo de este trabajo de fin de grado es abordar el modelo de la red GAN mencionada y mezclarlo con un transformer, viendo así las múltiples ventajas que esto conlleva y entrenando dichos modelos para generar imágenes parecidas al conjunto de datos público(CIFAR-10). Para poder cumplir con este objetivo se van a llevar a cabo un conjunto de subobjetivos:

- **Descripción del funcionamiento de las redes neuronales**

A lo largo de este trabajo se estudiará la arquitectura de la red GAN, de los transformer y de la implementación de la segunda en la primera, viendo así las ventajas que conlleva

- Buen empleo de las librerías de Python Tensorflow, Keras y Pytorch para la implementación de las distintas redes neuronales.
- Búsqueda y empleo de un conjunto de datos público de imágenes adecuado para el entrenamiento de las dos redes a estudiar y obtención de resultados.
- Realización de pruebas para mostrar el rendimiento de las redes a estudiar, exponiendo las limitaciones y ventajas de cada una de ellas.

## 2. Índice de figuras

### Índice de figuras

|     |   |    |
|-----|---|----|
| 1.  | Imagen de una red neuronal . . . . .  | 6  |
| 2.  | Imagen de una red neuronal multicapa . . . . .  | 8  |
| 3.  | Imagen del algoritmo backpropagation . . . . .  | 9  |
| 4.  | Imagen de varios tensores y su cambio en función del rango . . . . .                              | 11 |
| 5.  | Imagen de la arquitectura . . . . .   | 12 |
| 6.  | Imagen juego discriminador y generador de manera sencilla . . . . .                               | 13 |
| 7.  | Imagen que muestra la función objetivo . . . . .  | 15 |
| 8.  | Resumen de la técnica Dropout . . . . .   | 17 |
| 9.  | Arquitectura de un transformer . . . . .  | 18 |
| 10. | Imagen encoder . . . . .  | 19 |
| 11. | Imagen decoder . . . . .  | 22 |
| 12. | Imagen que resume el dataset . . . . .  | 26 |
| 13. | Imagen del conjunto de datos preprocesadas . . . . .  | 29 |
| 14. | Imagen que muestra arquitectura de un generador . . . . .   | 33 |
| 15. | Resumen de Tensorflow de la arquitectura del generador implementado para la GAN . . . . .         | 34 |
| 16. | Resumen de Tensorflow de la arquitectura del discriminador implementado para la red GAN . . . . . | 37 |
| 17. | Ejemplo análogo de la arquitectura de la red GAN que se va a implementar . . . . .                | 38 |
| 18. | Representación del Dropout . . . . .  | 46 |
| 19. | Gráfica de la pérdida del generador y discriminador . . . . .                                     | 48 |
| 20. | Imagen epoch 1 . . . . .  | 49 |
| 21. | Imagen epoch 40 . . . . .   | 49 |
| 22. | Imagen epoch 80 . . . . .   | 50 |
| 23. | Imagen epoch 150 . . . . .  | 50 |
| 24. | Imagen epoch 250 . . . . .  | 51 |
| 25. | Imagen epoch 320 . . . . .  | 51 |
| 26. | Imagen epoch 450 . . . . .  | 52 |
| 27. | Imagen epoch 500 . . . . .  | 52 |
| 28. | Evolución del FID a medida que van avanzando las epochs . . . . .                                 | 53 |
| 29. | Gráfica del FID en el entrenamiento . . . . .   | 58 |

### **3. Estructura del TFG**

Dentro de este TFG se va a llevar a cabo la siguiente estructura:

- En el capítulo 4, marco teórico primero se resume la teoría básica tras las redes neuronales. Además, en él se trata individualmente cada uno de los dos modelos, explicando sus bases, arquitecturas, mecanismos de entrenamiento.
- El capítulo 5, el planteamiento del problema, describe en profundidad los objetivos del trabajo de fin de grado, el conjunto de datos a emplear y los problemas a resolver.
- El capítulo 6 lleva a cabo la solución propuesta, explicando cómo se ha implementado los dos modelos neuronales a nivel de arquitectura e hiperparámetros, cómo se ha preprocesado el conjunto de datos y los resultados que se han obtenido.
- El capítulo 7 concluye con el modelo que resulta más adecuado mediante una justificación analítica, para resolver el problema planteado de la generación de imágenes del conjunto de datos público
- El capítulo 8 plantea futuros trabajos que se pueden desarrollar a partir de este
- El capítulo 9 incluye la bibliografía empleada para la realización de este trabajo de fin de grado.

## 4. Marco teórico

En esta sección se va a realizar una breve introducción a las redes neuronales, para posteriormente realizar una explicación más detallada de estas redes neuronales abarcando así un conjunto de conceptos los cuales son clave a la hora de entender el posterior entrenamiento de la red neuronal.

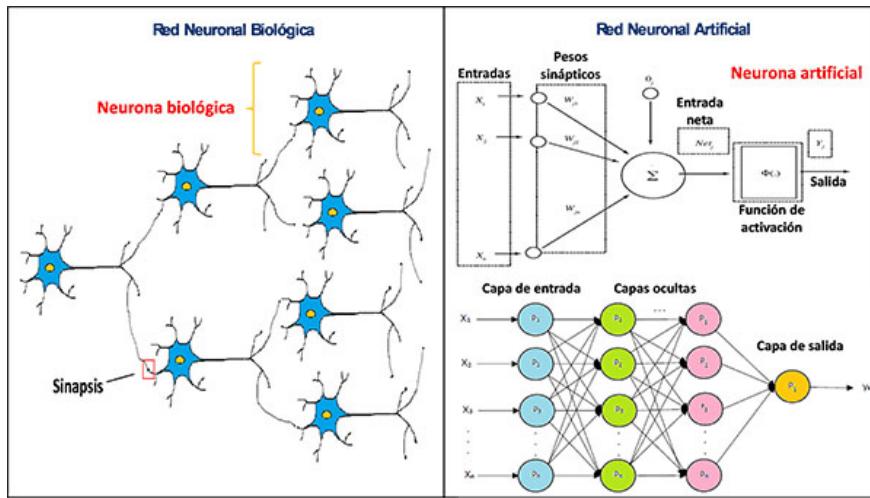


Figura 1: Imagen de una red neuronal

### 4.1. Introducción a las redes neuronales artificiales

Las redes neuronales son un tipo de modelo de aprendizaje automático inspirado en la estructura y el funcionamiento del cerebro humano. Están diseñadas para reconocer patrones y aprender a partir de datos, lo que las hace extremadamente útiles en tareas como el reconocimiento de imágenes, la clasificación de texto y la predicción de series temporales. Una red neuronal se compone de capas de neuronas artificiales, que se organizan en una arquitectura que permite a la red transformar la entrada en una salida deseada.

En 1958, introducido por Frank Rosenblatt surge el perceptrón que es el componente más básico y fundamental de una red neuronal artificial. El perceptrón es una unidad computacional que modela una neurona artificial. Se utiliza como un clasificador binario y es capaz de tomar una decisión basándose en la ponderación de las entradas que recibe. Aunque inicialmente fue concebido como un modelo sencillo, el perceptrón sienta las bases de las redes neuronales más complejas que utilizamos hoy en día y de ahí su importancia.

Un perceptrón contiene los siguientes elementos:

- **Entradas (Inputs):**

Denotadas generalmente como  $x_1, x_2, \dots, x_n$ . Estas entradas representan las características de los datos que el perceptrón debe procesar.

- **Pesos (Weights):**

Asociados a cada entrada, los pesos  $w_1, w_2, \dots, w_n$  determinan la importancia relativa de cada entrada. Un peso alto indica que la entrada correspondiente tiene una influencia significativa en la decisión del perceptrón.

Los pesos controlan la influencia que cada entrada tiene en la salida del perceptrón. Si un peso es grande, significa que la entrada correspondiente tiene una gran importancia en la decisión final. Durante el entrenamiento, los pesos se ajustan para minimizar el error entre la salida predicha y la salida esperada, permitiendo que el perceptrón aprenda de los datos.

#### ■ Bias (Sesgo):

El bias  $b$  es un término adicional que se suma a la combinación ponderada de las entradas. Actúa como un umbral que ajusta la facilidad con la que la neurona se activa, permitiendo que el perceptrón modele funciones más complejas.

El bias permite desplazar la función de activación hacia la derecha o izquierda, lo que efectivamente ajusta el umbral de activación del perceptrón. Esto es crucial porque permite que el perceptrón modele funciones que no pasan por el origen (no lineales) y toma decisiones incluso cuando todas las entradas son cero.

#### ■ Función de Activación:

Es una función matemática que se aplica al resultado de la combinación ponderada de las entradas y el bias. La función de activación determina si la neurona se activa (produce una salida) o no.

El bias permite desplazar la función de activación hacia la derecha o izquierda, lo que efectivamente ajusta el umbral de activación del perceptrón. Esto es crucial porque permite que el perceptrón modele funciones que no pasan por el origen (no lineales) y toma decisiones incluso cuando todas las entradas son cero.

El perceptrón calcula primero una combinación lineal de las entradas ponderadas y el bias:

$$z = \sum_{i=1}^n w_i x_i + b$$

Donde  $w_i$  son los pesos,  $x_i$  son las entradas,  $b$  es el bias,  $z$  es el valor total que se pasa a la función de activación.

Luego, el valor  $z$  se pasa a través de la función de activación  $\sigma(z)$  para producir la salida del perceptrón:

$$y = \sigma(z)$$

Donde  $y$  es la salida del perceptrón y  $\sigma(z)$  es la función de activación aplicada a  $z$ .

## 4.2. Red neuronal multicapa

En redes neuronales más complejas, los perceptrones se organizan en capas. Cada capa consiste en múltiples perceptrones (neuronas), y la salida de una capa se convierte en la entrada para la siguiente. Esto se conoce como una red neuronal multicapa (MLP).

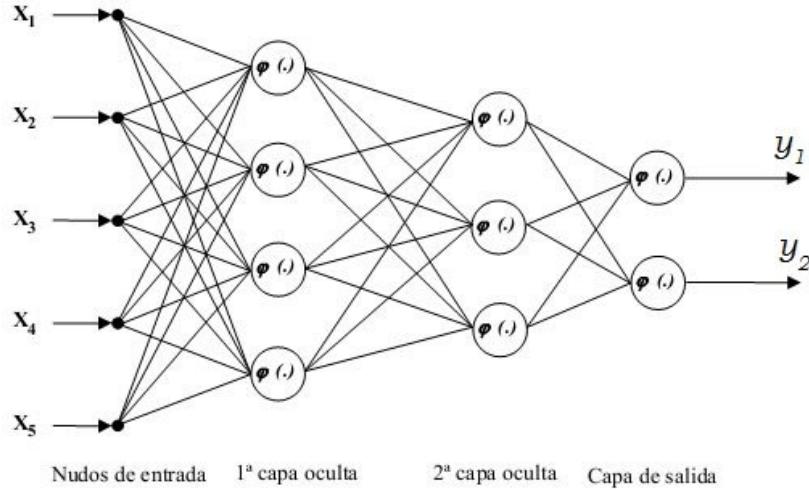


Figura 2: Imagen de una red neuronal multicapa

- **Capa de Entrada:** Contiene los perceptrones que reciben las entradas originales del modelo.
- **Capas Ocultas:** Estas capas intermedias contienen perceptrones que procesan las salidas de la capa anterior y extraen características más abstractas.
- **Capa de Salida:** Los perceptrones en esta capa producen las predicciones finales del modelo.

Cada neurona en las capas ocultas y de salida realiza una combinación ponderada de sus entradas, aplica un bias y pasa el resultado a través de una función de activación, similar a un perceptrón simple.

A lo largo de esta sección te habrás preguntado cómo ajustar los pesos y bias de la red para que aprenda a hacer predicciones correctas. Aquí es donde el backpropagation entra en juego.

## 4.3. Backpropagation

El backpropagation o retropropagación del error es un algoritmo fundamental en el entrenamiento de redes neuronales artificiales. Introducido por David E. Rumelhart, Geoffrey Hinton y Ronald J. Williams en su trabajo de 1986, "Learning representations by back-propagating errors.", este trabajo marcó un hito en el campo de las redes neuronales, ya que proporcionó un método eficiente para entrenar redes neuronales multicapa, dicho método

consistía en el hecho de poder ajustar los parámetros de una red neuronal (como los pesos y biases) para minimizar el error en las predicciones revolucionando el campo del aprendizaje automático. Aunque las ideas básicas detrás del backpropagation, como la regla de la cadena, ya existían en matemáticas, su aplicación práctica para entrenar redes neuronales no se había desarrollado completamente hasta este momento.

#### 4.3.1. Funcionamiento del backpropagation

Es fundamental entender que el backpropagation es un algoritmo de optimización que permite entrenar redes neuronales mediante el ajuste iterativo de los pesos y bias de la red.

Funciona calculando el gradiente de la función de pérdida con respecto a cada uno de los parámetros de la red, y luego actualizando esos parámetros en la dirección que minimiza la pérdida. La "propagación hacia atrás" se refiere al proceso de calcular estos gradientes comenzando desde la capa de salida de la red y avanzando hacia atrás a través de las capas ocultas hasta llegar a la capa de entrada.

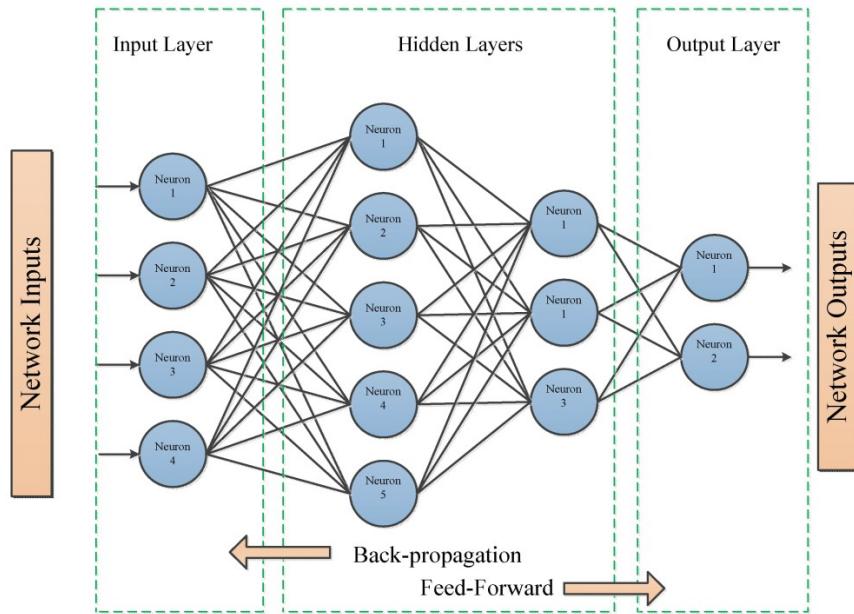


Figura 3: Imagen del algoritmo backpropagation

Los pesos determinan la fuerza de la conexión entre las neuronas en capas adyacentes, mientras que los bias ajustan el umbral de activación de las neuronas. Estos parámetros se actualizan para reducir el error de la red a medida que avanza el entrenamiento.

A su vez dicho algoritmo necesita de una función de pérdida, ya explicada y del gradiente, que es el vector de derivadas parciales de la función de pérdida con respecto a los pesos y bias, es fundamental porque indica la dirección y magnitud del cambio necesario para reducir la pérdida.

Este algoritmo se compone de dos fases:

- **Forward pass**

En esta fase, los datos de entrada pasan a través de la red, capa por capa, hasta que se produce una salida. Esta salida se compara con la salida esperada mediante la función de pérdida, y se calcula el error de la red.

- **Backward pass**

Aquí es donde ocurre la retropropagación del error. Se utiliza la regla de la cadena para calcular los gradientes de la función de pérdida con respecto a cada peso y bias, comenzando desde la capa de salida y retrocediendo hasta la capa de entrada. Estos gradientes se utilizan para actualizar los pesos y biases en una dirección que reduce la función de pérdida.

Una vez que se han calculado los gradientes, se utilizan para actualizar los pesos y biases mediante un algoritmo de optimización, como el descenso de gradiente.

En cada iteración del entrenamiento, los parámetros de la red se ajustan un poco más hacia los valores que minimizan la pérdida, permitiendo el aprendizaje de los datos de entrada.

Se ha hecho especial énfasis en este algoritmo porque permite que las redes neuronales multicapa (también conocidas como perceptrones multicapa) se entrenen de manera eficiente. Antes de la introducción del backpropagation, entrenar redes profundas era extremadamente difícil y computacionalmente costoso.

## 4.4. Tensores

Por último, antes de empezar a explicar las redes GAN se va introducir el concepto de los tensores, los cuales son las estructuras de datos fundamentales utilizadas para representar las entradas, pesos, salidas y otros parámetros intermedios que se manejan durante el proceso de entrenamiento y predicción.

Un tensor es una generalización de los conceptos de escalares, vectores y matrices a más dimensiones. Matemáticamente, un tensor es una estructura de datos multidimensional que puede tener 0, 1, 2 o más dimensiones (también conocidas como rangos o "órdenes").

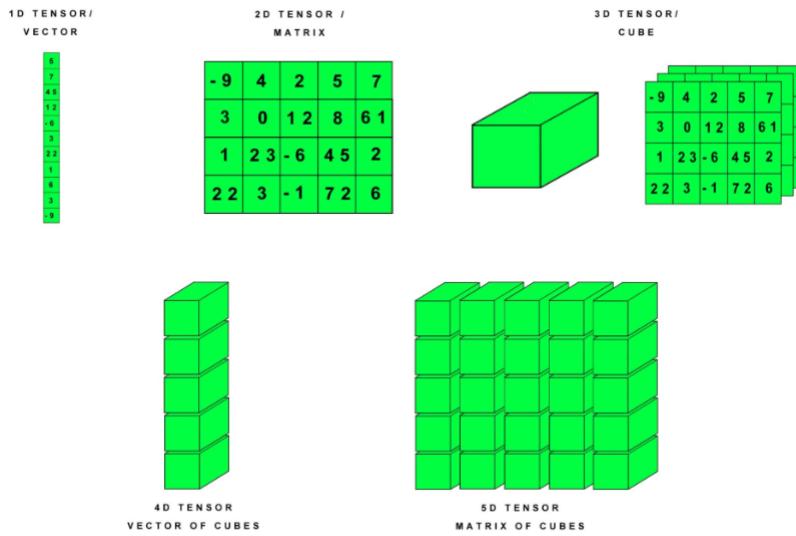


Figura 4: Imagen de varios tensores y su cambio en función del rango

Un tensor de rango 0 es simplemente un solo número, es decir, un escalar. Un tensor de rango 1, es una lista de números, es decir, un vector. Un tensor de rango 2, es una tabla de números con filas y columnas, es decir, una matriz. Un tensor de rango 3 o superior es una estructura de datos con tres o más dimensiones. Un ejemplo podría ser un conjunto de imágenes en color las cuales se podrían representar como un tensor de rango 4 con dimensiones (número de imágenes, altura, anchura, canales de color). Esto, es fundamental en el entrenamiento ya que se va a trabajar con un conjunto de imágenes que se transforman en tensores.

## 4.5. Redes Generativas Adversariales(GAN)

Las Redes Generativas Antagónicas, comúnmente conocidas como GANs por sus siglas en inglés (Generative Adversarial Networks), representan uno de los avances más significativos en el campo del aprendizaje profundo desde su introducción por Ian Goodfellow y sus colegas en 2014. Estas redes han transformado la manera en que se conciben y se crean modelos generativos, abriendo nuevas posibilidades en una amplia gama de aplicaciones, desde la generación de imágenes y videos realistas hasta la creación de datos sintéticos y la mejora

de la privacidad de los datos. En esta sección, exploraremos en detalle la estructura, el funcionamiento y las aplicaciones de las GANs.

Las Redes Generativas Antagónicas han revolucionado el campo de la inteligencia artificial y el aprendizaje profundo, ofreciendo nuevas herramientas para la generación de datos y la creación de contenido sintético. Su capacidad para producir datos realistas a partir de un conjunto de datos ha abierto innumerables vías de investigación y aplicaciones. Sin embargo, a medida que se desarrollan y perfeccionan, las GANs siguen enfrentando desafíos técnicos que requieren soluciones innovadoras, y este es el tema central de mi tfg, el cómo optimizar una red generativa a través de un tecnologías innovadoras como son los transformers.

#### 4.5.1. Arquitectura y Funcionamiento de la GAN

La arquitectura básica de una red GAN se muestra a continuación:

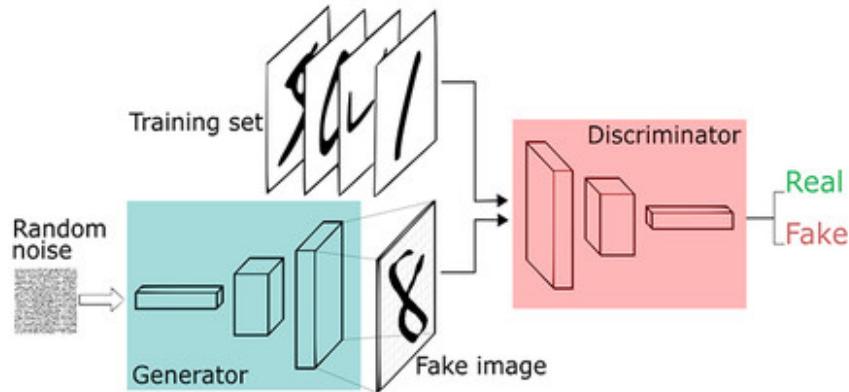


Figura 5: Imagen de la arquitectura

En la arquitectura se observan dos modelos de perceptrones multicapa.

El primero es el generador, que recibe de entrada un vector ruido (como se observa en la figura el ruido introducido es equivalente a una imagen de píxeles aleatorios) la estructura del generador suele ser una red neuronal deconvolucional (o transpuesta convolucional), que toma como entrada un vector de ruido aleatorio y lo transforma en una salida que tiene la misma dimensión que los datos reales, como una imagen.

El segundo es el discriminador, mostrado a la derecha en la figura, que recibe la salida del generador junto a imágenes muestradas del conjunto de datos de entrenamiento.

El proceso de entrenamiento de una GAN se basa en una dinámica de juego de suma cero entre el generador y el discriminador. Donde el generador intenta engañar al discriminador creando datos que sean indistinguibles de los datos reales, teniendo como objetivo minimizar

la función de pérdida que mide la habilidad del discriminador para distinguir entre datos reales y generados. Y el discriminador, por otro lado, busca maximizar su capacidad para diferenciar correctamente entre los datos reales y los sintéticos. Intenta maximizar una función de pérdida que detecta las falsas clasificaciones.

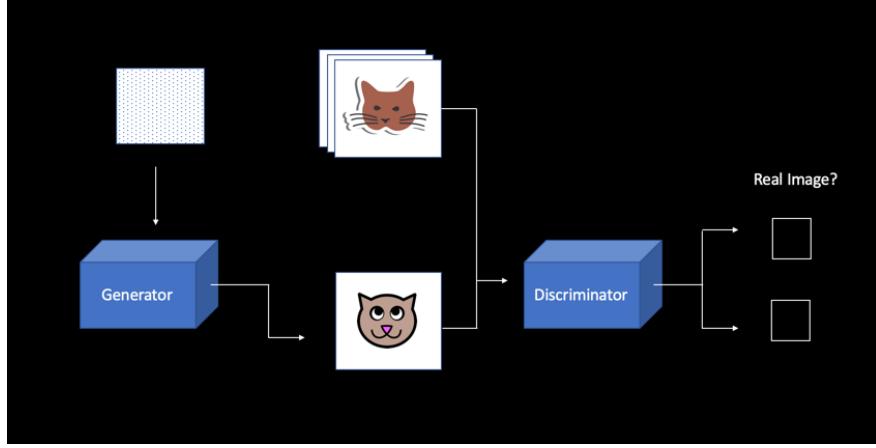


Figura 6: Imagen juego discriminador y generador de manera sencilla

La dinámica de juego de suma cero es un concepto central en la teoría de juegos, que describe una situación en la que las ganancias de un participante son exactamente iguales a las pérdidas de otro. En el contexto de las Redes Generativas Antagónicas (GANs), este concepto se aplica a la relación entre el generador y el discriminador.

La dinámica de juego de suma cero en una GAN se aplica matemáticamente utilizando funciones de pérdida para el generador y el discriminador. La función de pérdida generalizada para el discriminador se puede expresar como:

$$\mathcal{L}_D = -\mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] - \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

Aquí,  $D(x)$  representa la probabilidad estimada por el discriminador de que la muestra  $x$  sea real, mientras que  $D(G(z))$  es la probabilidad de que la muestra generada  $G(z)$  sea real. El discriminador intenta maximizar esta función de pérdida.

La función de pérdida del generador se define como:

$$\mathcal{L}_G = -\mathbb{E}_{z \sim p_z(z)}[\log D(G(z))]$$

El generador intenta minimizar esta función, es decir, intenta generar datos  $G(z)$  que maximicen la probabilidad de que el discriminador  $D$  los clasifique como reales.

El entrenamiento de una GAN como se ha explicado implica un proceso iterativo en el que el generador y el discriminador se entrena alternadamente, cada uno optimizando su función de pérdida. Se puede ver los pasos de manera explícita tal que así:

Primero, se actualiza el **Discriminador** dando datos reales al discriminador y calculando la probabilidad de que estos sean reales, generando así datos sintéticos que se pasan al discriminador, calculando la probabilidad de que estos sean reales. Por último, se calcula la función de pérdida del discriminador y se actualizan los pesos del discriminador para maximizar esta función.

Después se actualiza el **Generador** donde primero se generan un lote de datos sintéticos que se pasan al discriminador y se calcula la función de perdida del generador y se actualizan los pesos del generador para minimizar esta función.

Para terminar, se repite este proceso de manera iterativa tantas como epochs tenga mi red neuronal. Para un correcto entrenamiento de una red neuronal se necesitan mínimo 100 epochs, pero lo recomendable son 500 con lo que supone esto a nivel computacional, es decir, horas de entrenamiento.

El objetivo ideal de una GAN es alcanzar un equilibrio de Nash, un concepto de la teoría de juegos que ocurre cuando ninguno de los jugadores (en este caso, el generador y el discriminador) puede mejorar su estrategia mientras el otro mantenga la suya constante. O lo que es lo mismo en una red GAN, un generador y un discriminador óptimos.

Dentro de este juego surgen muchos desafíos como el **desbalance en el juego** (generalmente producido por el discriminador) otro desafío es la **inestabilidad en el entrenamiento** fruto de unos hiperparámetros que se necesitan reajustar. Sin embargo, estos desafíos y sus correcciones se verán mas adelante.

#### 4.5.2. Función objetivo

La función objetivo global de la GAN se puede expresar como un problema de optimización tipo minimax, donde el generador intenta minimizar su pérdida mientras que el discriminador intenta maximizar la suya. Su expresion matematica es:

$$\min_G \max_D \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

Esta ecuación representa un juego entre  $G$  y  $D$ :

Maximización: El discriminador  $D$  trata de maximizar la probabilidad de asignar la etiqueta correcta a las entradas (reales o generadas). Minimización: El generador  $G$  trata de minimizar la capacidad del discriminador para distinguir entre datos reales y generados.

La función objetivo de una GAN es fundamental para su funcionamiento, ya que define la competencia entre el generador y el discriminador. A través de este juego de minimax, la GAN aprende a generar datos sintéticos que se asemejan cada vez más a los datos reales.

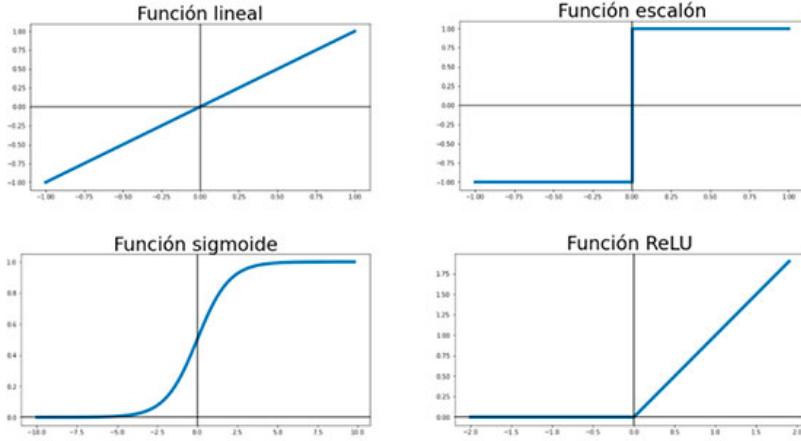


Figura 7: Imagen que muestra la función objetivo

Estos son algunos ejemplos de las funciones objetivo que se pueden llegar a usar, de hecho ReLU y sigmoide van a ser usadas en el entrenamiento.

#### 4.5.3. Optimización de los parámetros

Los parámetros de la red discriminadora  $\theta_d$  son actualizados mediante el gradiente ascendente (maximización) y los parámetros de la red generadora  $\theta_g$  son actualizados mediante el gradiente descendente (minimización).

El objetivo del discriminador es maximizar su capacidad para diferenciar entre imágenes reales y falsas. O lo que es lo mismo, maximizar la función de pérdida:

$$\mathcal{L}_D = -\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

Donde:

- $\mathbf{x}$  son las imágenes reales.
- $\mathbf{z}$  es el vector de ruido (espacio latente) que se introduce al generador.
- $G(\mathbf{z})$  son las imágenes generadas por el generador.
- $p_{\text{data}}(\mathbf{x})$  es la distribución de los datos reales.
- $p_{\mathbf{z}}(\mathbf{z})$  es la distribución del ruido.

Para maximizar la función de pérdida  $\mathcal{L}_D$ , se aplican técnicas de optimización como el gradiente descendente, ajustando los parámetros  $\theta_d$  del discriminador de la siguiente manera:

$$\theta_d \leftarrow \theta_d - \eta \nabla_{\theta_d} \mathcal{L}_D$$

Donde:

- $\eta$  es la tasa de aprendizaje (learning rate).

- $\nabla_{\theta_d} \mathcal{L}_D$  es el gradiente de la función de pérdida del discriminador con respecto a sus parámetros  $\theta_d$ .

El objetivo del generador  $G$  es producir imágenes que el discriminador clasifique como reales, o lo que es lo mismo minimizar la siguiente función de pérdida:

$$\mathcal{L}_G = -\mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log D(G(\mathbf{z}))]$$

**Actualización de los parámetros del generador:** Para minimizar la función de pérdida  $\mathcal{L}_G$ , se aplican técnicas de optimización como el gradiente descendente, ajustando los parámetros  $\theta_g$  del generador:

$$\theta_g \leftarrow \theta_g - \eta \nabla_{\theta_g} \mathcal{L}_G$$

Donde:

- $\eta$  es la tasa de aprendizaje (learning rate).
- $\nabla_{\theta_g} \mathcal{L}_G$  es el gradiente de la función de pérdida del generador con respecto a sus parámetros  $\theta_g$ .

#### 4.5.4. Técnicas de mejora

Esta demostrado que las redes GAN tienen la capacidad de llegar al punto de equilibrio de Nash en este juego de suma cero, sin embargo, como ya se mencionó anteriormente, existen muchos desafíos en este juego de suma cero, el más común es el desbalance en el juego que generalmente sucede en el discriminador, esto ocurre cuando uno de los jugadores se vuelve demasiado fuerte dominando el juego, llevando al colapso del generador (por ejemplo, mode collapse). Esto ocurre cuando el generador produce solo un conjunto limitado de resultados, ya que no puede aprender a mejorar debido a un discriminador demasiado fuerte.

Es por esto por lo que vamos a ver técnicas que evitan que el discriminador se vuelva demasiado fuerte. Las cuales son

- Dropout

El dropout es una técnica de regularización que, durante el entrenamiento, aleatoriamente desactiva (pone a cero) un porcentaje de neuronas en la red. Esto evita que las neuronas se vuelvan demasiado dependientes de otras neuronas específicas y promueve una mayor redundancia y generalización.

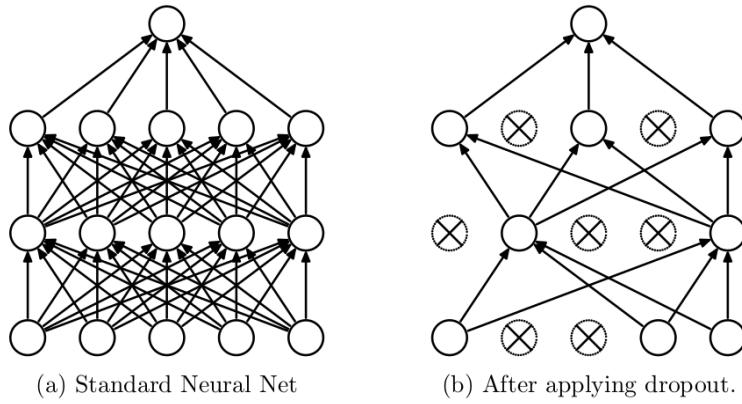


Figura 8: Resumen de la técnica Dropout

Esto, reduce el riesgo de que el modelo se sobreajuste al memorizar patrones específicos del conjunto de entrenamiento.

- Label Smoothing (Suavizado de Etiquetas)

El label smoothing consiste en no utilizar etiquetas completamente duras (por ejemplo, 0 y 1) para entrenar el discriminador, sino etiquetas suavizadas (por ejemplo, 0.9 en lugar de 1 para datos reales). Esto evita que el discriminador se vuelva demasiado confiado y fomenta una mayor generalización.

- Ajuste de la tasa de aprendizaje

La tasa de aprendizaje es un hiperparámetro que controla la velocidad a la que un modelo ajusta sus pesos en respuesta al error que se encuentra en cada iteración o batch del entrenamiento. En las GAN se puede ajustar la tasa de aprendizaje del discriminador para reducir su capacidad de aprender rápidamente, lo que permite que el generador aprenda mejor. A su vez, se puede ajustar este hiperparámetro aumentando la tasa de aprendizaje del generador

- Normalización espectral(Spectral Normalization)

Esta técnica se usa sobre todo en las redes GAN y fue introducida en el paper "Spectral Normalization for Generative Adversarial Networks" por Takeru Miyato et al., en 2018. Su objetivo es controlar los valores de los pesos de las capas de la red, asegurando que no crezcan demasiado. Matemáticamente se describe de la siguiente forma. Dado un peso  $W$  en una capa de la red, la normalización espectral ajusta este peso dividiéndolo por su **valor singular más grande** (también conocido como la norma espectral). Matemáticamente, esto se expresa como:

$$\bar{W} = \frac{W}{\sigma(W)}$$

donde  $W$  es la matriz de pesos de la capa. Y  $\sigma(W)$  es la **norma espectral** de  $W$ , que es el mayor valor singular de la matriz  $W$ .

Dicho valor singular se calcula de la siguiente manera:

El valor singular más grande  $\sigma(W)$  se obtiene de la descomposición en valores singulares (SVD) de  $W$ . Si  $W$  es una matriz de  $m \times n$ , la SVD de  $W$  se descompone como:

$$W = U\Sigma V^T$$

donde  $U$  es una matriz ortogonal de dimensión  $m \times m$ ,  $\Sigma$  es una matriz diagonal  $m \times n$  cuyos elementos diagonales son los valores singulares de  $W$  y por último  $V$  es una matriz ortogonal de dimensión  $n \times n$ .

El mayor valor singular es el máximo de los elementos diagonales de  $\Sigma$ .

## 4.6. Transformers

Los Transformers son una arquitectura de red neuronal introducida por Vaswani et al. en 2017 en el artículo “Attention is All You Need”. Esta arquitectura ha revolucionado el campo del procesamiento del lenguaje natural (NLP) y se ha extendido a otras áreas como la visión por computadora y la generación de datos.

### 4.6.1. Arquitectura de un Transformer

Los Transformers utilizan una arquitectura de codificador decodificador (encoder-decoder), que es fundamental para procesar y transformar secuencias de datos.

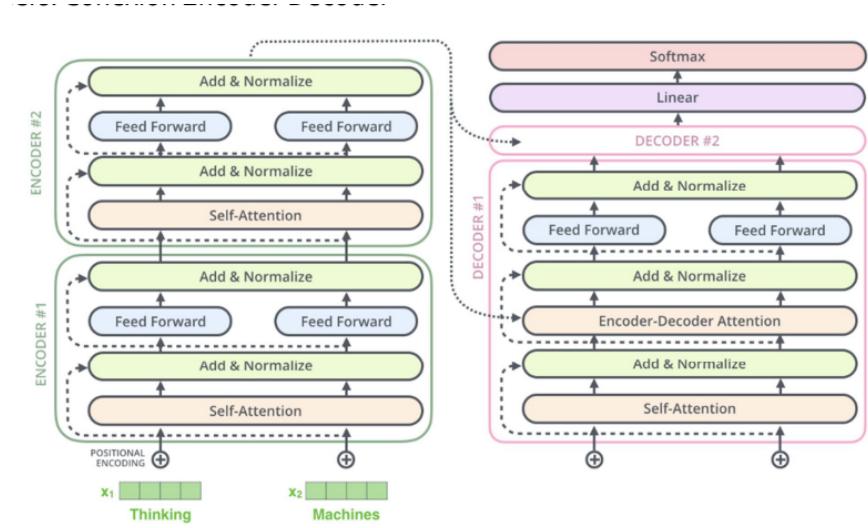


Figura 9: Arquitectura de un transformer

#### 4.6.2. Encoder

El **encoder** en un Transformer es una de las partes fundamentales de la arquitectura, responsable de procesar la secuencia de entrada y generar una representación interna que captura las relaciones entre los elementos de la secuencia. El encoder está compuesto por varias capas idénticas que se apilan una sobre otra. Cada capa de un encoder consiste en dos subcomponentes principales: el **mecanismo de atención auto-regresiva (self-attention)** y una **red feedforward completamente conectada**. A continuación, se explica detalladamente cada componente, incluyendo el mecanismo de atención y su formulación matemática.

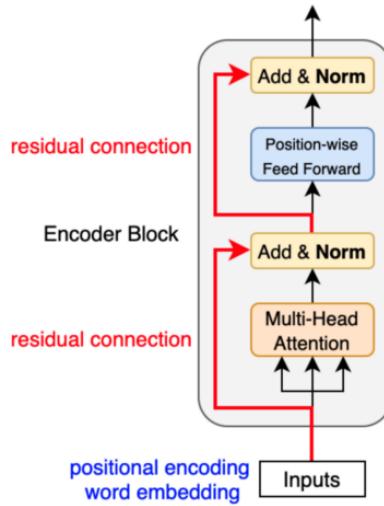


Figura 10: Imagen encoder

El mecanismo de atención auto-regresiva es el núcleo de la arquitectura del Transformer, permitiendo que cada token de la secuencia de entrada se enfoque en todos los demás tokens para capturar dependencias de largo alcance.

Dado un conjunto de vectores de entrada  $X$  (donde cada vector representa un token en la secuencia de entrada), el primer paso en el mecanismo de atención es transformar estos vectores en tres matrices: **Query (Q)**, **Key (K)**, y **Value (V)**.

Estas matrices se calculan mediante multiplicaciones lineales de los vectores de entrada  $X$  con matrices de pesos aprendibles  $W_Q$ ,  $W_K$ , y  $W_V$ :

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

Donde:

- $Q$  (Query) representa la consulta que cada token hace sobre los demás tokens.
- $K$  (Key) representa una clave que cada token ofrece para la consulta.
- $V$  (Value) representa el valor que cada token transmite en respuesta a las consultas.

Las dimensiones de  $Q$ ,  $K$ , y  $V$  son las mismas, igual a la dimensión de los vectores de entrada  $X$ .

La puntuación de atención entre un token  $i$  y un token  $j$  se calcula como el producto punto entre la query del token  $i$  y la key del token  $j$ . Esta puntuación se normaliza mediante la raíz cuadrada de la dimensión de las keys (denotada como  $d_k$ ) para estabilizar los gradientes durante el entrenamiento:

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

Donde:

- $\frac{QK^T}{\sqrt{d_k}}$  es una matriz que contiene las puntuaciones de atención para cada par de tokens.
- La función **softmax** se aplica para convertir estas puntuaciones en probabilidades, asegurando que las probabilidades asignadas a todos los tokens sumen 1.

Las puntuaciones de atención resultantes se utilizan para hacer una suma ponderada de los valores  $V$ , donde los pesos son las probabilidades calculadas por la softmax. Esto produce un nuevo conjunto de vectores que representan la entrada procesada, teniendo en cuenta la influencia de todos los tokens de la secuencia:

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V$$

Este vector resultante se convierte en la entrada para la siguiente capa del encoder o la red feedforward dentro de la misma capa.

Para capturar diferentes tipos de relaciones entre los tokens, el Transformer utiliza un mecanismo llamado **multi-head attention**. En lugar de realizar la atención solo una vez, se realizan múltiples atenciones en paralelo con diferentes conjuntos de pesos  $W_Q$ ,  $W_K$ , y  $W_V$ . Los resultados de estas múltiples cabezas de atención se concatenan y pasan por una proyección lineal adicional para producir la salida final:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W_O$$

Donde:

- $\text{head}_i = \text{Attention}(QW_{Q_i}, KW_{K_i}, VW_{V_i})$
- $W_O$  es una matriz de pesos que transforma la concatenación de las múltiples cabezas en la forma de salida deseada.

Este enfoque permite que el modelo aprenda diferentes aspectos o relaciones dentro de los datos en cada cabeza de atención.

Después del mecanismo de atención, cada token pasa por una red feedforward completamente conectada, que se aplica de manera independiente a cada token.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Donde:

- $W_1$  y  $W_2$  son matrices de pesos aprendibles.

- $b_1$  y  $b_2$  son vectores de sesgo.
- $\max(0, \cdot)$  representa la función de activación ReLU.

La red feedforward transforma los datos procesados por el mecanismo de atención, permitiendo que el modelo capture no linealidades complejas.

Cada subcomponente (self-attention y feedforward) está rodeado por una capa de normalización de batch (batch normalization) y una conexión residual (skip connection):

$$\text{Output} = \text{LayerNorm}(x + \text{SubLayer}(x))$$

Donde  $\text{SubLayer}(x)$  representa la salida de la capa de self-attention o feedforward. La normalización de capa (Layer Normalization) asegura que las entradas a cada capa tengan una distribución consistente, mientras que la conexión residual permite un mejor flujo de gradientes y facilita el entrenamiento de redes profundas.

El **encoder** de un Transformer procesa la secuencia de entrada utilizando el mecanismo de atención auto-regresiva para capturar dependencias entre tokens, seguido de una red feed-forward que transforma estas representaciones. Este enfoque permite que el modelo capture y utilice tanto la información local como global de manera efectiva, lo que es crucial para tareas que requieren comprensión de contextos complejos. El uso de técnicas como multi-head attention y conexiones residuales mejora aún más la capacidad del modelo para aprender relaciones ricas en los datos.

#### 4.6.3. Decoder

El **decodificador** (decoder) es la parte del Transformer encargada de generar la secuencia de salida a partir de la representación intermedia obtenida del encoder. Al igual que el encoder, el decodificador consta de varias capas idénticas que se apilan una sobre otra. Sin embargo, el decodificador tiene una estructura ligeramente más compleja debido a la necesidad de manejar tanto la secuencia de entrada como la secuencia de salida generada hasta el momento. A continuación, se explica detalladamente cada componente del decodificador, incluyendo su formulación matemática.

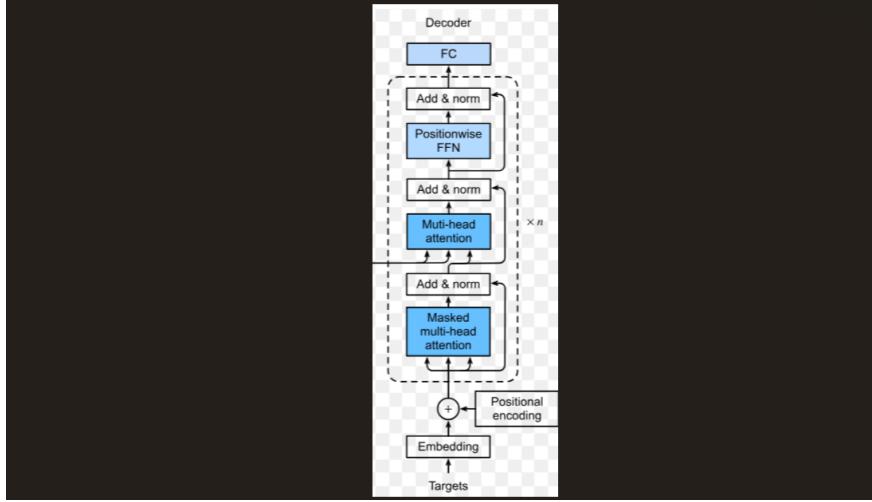


Figura 11: Imagen decoder

Cada capa del decodificador tiene tres subcomponentes principales:

- **Masked Self-Attention:** Una capa de atención auto-regresiva enmascarada que se aplica a la secuencia de salida generada hasta el momento.
- **Cross-Attention:** Una capa de atención que permite que cada token de la secuencia de salida se enfoque en la secuencia de entrada.
- **Red Feedforward:** Una red feedforward completamente conectada que procesa las representaciones resultantes.

Cada uno de estos subcomponentes está rodeado por capas de normalización de capa (*Layer Normalization*) y conexiones residuales.

El primer subcomponente del decodificador es la capa de **masked self-attention**. Esta es similar a la capa de self-attention en el encoder, pero con una diferencia importante: durante el entrenamiento, los tokens futuros en la secuencia de salida están enmascarados. Esto significa que, para predecir un token, el modelo solo puede ver los tokens anteriores en la secuencia, no los futuros.

Igual que en el encoder, la secuencia de salida hasta el momento se transforma en matrices de queries ( $Q$ ), keys ( $K$ ) y values ( $V$ ) mediante multiplicaciones lineales con matrices de pesos aprendibles:

$$Q = YW_Q, \quad K = YW_K, \quad V = YW_V$$

Donde  $Y$  es la secuencia de salida generada hasta el momento.

Se aplica una máscara triangular inferior a la matriz de atención calculada ( $QK^T$ ) para asegurarse de que el modelo no pueda ver tokens futuros en la secuencia de salida:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} + M \right) V$$

Donde  $M$  es la máscara que asigna un valor muy negativo (e.g.,  $-\infty$ ) a las posiciones correspondientes a tokens futuros, de modo que la softmax las convierte en ceros, eliminando su influencia.

Como en el encoder, se utilizan múltiples cabezas de atención para capturar diferentes aspectos de la dependencia entre los tokens en la secuencia de salida:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W_O$$

El segundo subcomponente del decodificador es la capa de **cross-attention**. A diferencia de la self-attention, que solo mira la secuencia de salida, la cross-attention permite que cada token de la secuencia de salida se enfoque en la secuencia de entrada procesada por el encoder.

Aquí, las queries ( $Q$ ) provienen de la secuencia de salida generada hasta el momento, mientras que las keys ( $K$ ) y values ( $V$ ) provienen de la secuencia de entrada procesada por el encoder:

$$Q = YW_Q, \quad K = XW_K, \quad V = XW_V$$

Donde  $X$  es la representación intermedia generada por el encoder.

El proceso es similar al de la self-attention, pero esta vez se enfocan las queries de la secuencia de salida en los keys de la secuencia de entrada para producir las salidas ponderadas:

$$\text{CrossAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Esta operación permite que el decodificador incorpore información relevante de la secuencia de entrada al generar la secuencia de salida.

Después de la capa de atención cruzada, cada token en la secuencia de salida pasa por una red feedforward completamente conectada, que se aplica de manera independiente a cada token, igual que en el encoder:

$$\text{FFN}(y) = \max(0, yW_1 + b_1)W_2 + b_2$$

Donde:

- $W_1$  y  $W_2$  son matrices de pesos aprendibles.
- $b_1$  y  $b_2$  son vectores de sesgo.
- $\max(0, \cdot)$  representa la función de activación ReLU.

Como en el encoder, cada subcomponente del decodificador (masked self-attention, cross-attention, y feedforward) está rodeado por capas de normalización y conexiones residuales:

$$\text{Output} = \text{LayerNorm}(y + \text{SubLayer}(y))$$

Estas conexiones permiten un mejor flujo de gradientes y estabilidad durante el entrenamiento.

Finalmente, la salida del decodificador se proyecta a través de una capa lineal que reduce la dimensionalidad al tamaño del vocabulario, seguida de una función softmax para producir una distribución de probabilidad sobre las posibles palabras o tokens de salida:

$$\text{Probabilidades} = \text{softmax}(W_{\text{proj}} \cdot \text{Output})$$

Donde  $W_{\text{proj}}$  es una matriz de pesos aprendibles que proyecta la salida del decodificador en el espacio del vocabulario.

Una vez que ya hemos repasado todos estos conceptos estamos listos para proceder a explicar tanto el entrenamiento tanto de la red GAN como de la red GAN mejorada por un transformer. Pero, antes que nada vamos a plantear el problema en cuestión.

## 5. Planteamiento del problema

En el ámbito del machine learning, la calidad y cantidad de datos de entrenamiento son vitales para el buen funcionamiento de los modelos. Sin embargo, en muchos casos, la obtención de datos de alta calidad es costosa o limitada como puede ser en dominios como la medicina, la industria o el análisis de datos sensibles. Esto conlleva a que los modelos no consigan un buen desempeño debido a los problemas que se derivan de la falta de datos como puede ser la falta de generalización e incapacidad de capturar toda la diversidad de escenarios o problemas con el sobreajuste, entre otros muchos.

Además, recolectar datos reales puede ser especialmente costoso y complicado en contextos como experimentos científicos, estudios de mercado o investigaciones en áreas donde el acceso a grandes volúmenes de datos es o bien restringido o bien muy costoso ya que implica una gran inversión.

Por todo lo anterior, se entiende que la obtención de datos sintéticos de alta calidad donde la diferencia con los datos reales sea prácticamente nula supone un gran conjunto de ventajas en los modelos de la red neuronal, implicando asimismo ventajas económicas en aquellos proyectos donde se quiera acceder a datos que conlleven una gran inversión para acceder a un gran volumen de estos datos. El problema central de este proyecto es cómo mejorar la generación de datos sintéticos de tal manera que sean prácticamente idénticos a los reales. Para ello se va a emplear una combinación de redes GAN y transformers. Mientras que las GANs son efectivas para generar datos, suelen enfrentar dificultades en la captura de dependencias a largo plazo y la generación de secuencias coherentes, problemas que los Transformers, con su arquitectura basada en atención, pueden mitigar.

El desafío del proyecto consiste en desarrollar un modelo que combine las capacidades de las GANs para generar datos sintéticos con la habilidad de los Transformers para mantener coherencia en secuencias y mejorar la diversidad de los datos generados. Este modelo debe ser capaz de producir datos sintéticos que sean válidos para entrenar otras redes neuronales, optimizando estas mismas. El objetivo principal es que a través de esta red neuronal se creen datos sintéticos con suficiente calidad y diversidad para permitir que el rendimiento de las redes entrenadas con estos datos sea igual o superior al obtenido con los datos reales. Para el cumplimiento de este objetivo general se deben cumplir simultáneamente un conjunto de objetivos específicos los cuales son:

- **Desarrollar una arquitectura combinada de GAN y Transformer** que permita la generación de datos sintéticos, integrando las capacidades de ambos modelos para maximizar la calidad de los datos generados.
- **Implementar un pipeline de entrenamiento** que permita entrenar el modelo combinado, ajustando los parámetros y la configuración para optimizar la generación de datos sintéticos.
- **Evaluar la calidad de los datos generados** mediante métricas estándar en la generación de datos sintéticos, como FID y IS, así como su efectividad para entrenar otras redes neuronales.
- **Comparar el rendimiento** de las redes entrenadas con datos sintéticos frente a

redes entrenadas con datos reales, para validar la utilidad de los datos generados por el modelo combinado.

Además de todo lo anterior, este enfoque tiene el potencial de disminuir significativamente los costos asociados a experimentos científicos, estudios de mercado, y otros escenarios donde la adquisición de datos representa una barrera considerable. Esto no solo facilita el avance en investigaciones y desarrollo de productos, sino que también permite a las organizaciones y científicos centrar sus recursos en innovaciones más críticas. Para el desarrollo de este modelo se ha utilizado la tarjeta grafica BLA BLA BLA Para llevar a cabo estos objetivos es necesario disponer de un repositorio de datos donde se lleve a cabo el desarrollo del modelo. En este caso se va a hacer uso del conjunto de datos CIFAR-10” disponible en la plataforma kaggle:

(<https://www.kaggle.com/datasets/khoongweihao/cifar10100>)

El CIFAR-10 es un conjunto de datos que consiste en 60.000 imágenes de tamaño pequeño en resolución 32x32 a color que a su vez consta de 10 clases las cuales, se componen por 6000 imágenes cada una. Se ha elegido este conjunto de datos debido a que CIFAR-10 contiene imágenes de 10 clases diferentes (como aviones, automóviles, pájaros, gatos, etc.) asegurando de esta manera que se mantenga la diversidad de las diferentes clases y verificando que el modelo sirve para la generación de datos completamente diferentes

El conjunto de datos se divide en dos partes una de entrenamiento compuesta por 50.000 imágenes y una de validación compuesta por 10.000 imágenes. A continuación, en la figura 2000 se muestra una imagen general del dataset y dos imágenes extraídas del conjunto de datos utilizado para el entrenamiento.

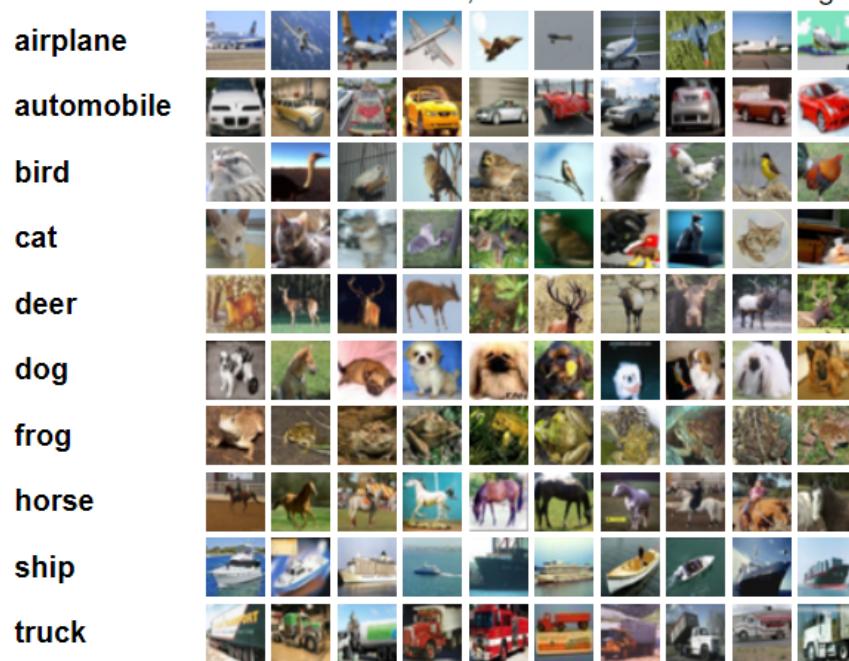


Figura 12: Imagen que resume el dataset

Antes de comenzar el entrenamiento, las imágenes del conjunto de datos se someten a

un proceso de preprocesamiento para tener la estructura correcta para tanto la arquitectura de la red GAN como la arquitectura del modelo que combina GAN y transformer y además, con el objetivo de acelerar el aprendizaje de las redes.

Una vez son entrenadas las redes se evalúan a tal fin de hacer una comparativa entre los dos modelos y así poder concluir que la combinación de red GAN con transformer supone un avance a la hora de producir datos sintéticos. Finalmente, se producirá en análisis de las imágenes generadas por los distintos modelos. Para una evaluación objetiva se utilizará la métrica Frechet Inception Distance(FID) que evalúa la similitud entre el conjunto de imágenes reales y las sintéticas. El FID se calcula utilizando un modelo de red neuronal preentrenado, la red Inception v3, con el objetivo de extraer características. Dichas características son luego modeladas mediante distribuciones normales multivariadas, y el FID mide la distancia entre estas dos distribuciones.

Matemáticamente, el FID se define como la distancia de Frechet entre las dos distribuciones de características:

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr} \left( \Sigma_r + \Sigma_g - 2\sqrt{\Sigma_r \Sigma_g} \right) \quad (1)$$

donde  $\mu_r$  y  $\mu_g$  son los vectores de media de las características de las imágenes reales y generadas, respectivamente.  $\Sigma_r$  y  $\Sigma_g$  son las matrices de covarianza de estas características. De esta forma se evaluarán las imágenes sintéticas de tal manera que un FID bajo implica una gran similitud entre las imágenes y un FID bajo supone baja similitud entre las imágenes.

## 6. Solución propuesta y resultados

Se va a ir explicando el cómo implementar estas redes y las conclusiones de su entrenamiento. Además, a lo largo de esta sección también se va a explicar cómo se han preparado el conjunto de 60.000 imágenes para entrenar ambas redes, dilucidando a su vez cómo implementar la arquitectura e hiperparámetros de los modelos y, por último, extrayendo conclusiones de los resultados obtenidos. Como métrica de evaluación del entrenamiento se hará uso de las técnicas FID e IS ya explicadas en el apartado anterior, con el objetivo de verificar la correcta creación de datos sintéticos.

### 6.1. Conjunto de datos y preprocessamiento

A pesar de que ambos modelos tienen parecidos entre sí requieren de un preprocessamiento diferente para el conjunto de imágenes. Sin embargo, hay determinados aspectos que tienen en común, como todo preprocessamiento de datos de una red neuronal, los cuales se desarrollarán en esta sección.

- **Normalización** Todas las imágenes del conjunto de datos son normalizadas antes de ser introducidas a la red. El objetivo de la normalización es ajustar los valores de los píxeles, que normalmente se encuentran en un rango de 0 a 255 indicando así su luminosidad, siendo el valor 0 el menos luminoso y el 255 el más luminoso. Cabe resaltar que en imágenes a color hay tres rangos de 0 a 255 para un mismo pixel ya que se superponen la imagen roja, verde y azul. Dichos píxeles se ajustan a un rango más adecuado para el modelo, en este caso al [0, 1] diviendo entre 255.

Es importante realizar este proceso por tres razones:

La primera de ellas es que supone una **mejora de la convergencia** debido a que al tener los valores de entrada estandarizados se asegura que ninguna característica domine a la otra haciendo que el modelo aprenda de manera más eficiente y evitando problemas como el desbordamiento numérico durante el cálculo de gradientes.

La segunda es que **facilita el entrenamiento** puesto que, cuando los datos están estandarizados las actualizaciones de los pesos son más uniformes y estables a lo largo de las diferentes capas de la red permitiendo así que algoritmos de optimización como el gradiente descendiente funcionen mejor.

Por último, se produce una **uniformidad en imágenes a color** ya que en el caso de imágenes a color la normalización permite que estén en el mismo rango, lo cual es crucial para el aprendizaje evitando que uno de los canales lo domine.

- **Redimensionamiento de las imágenes** Para el entrenamiento de las distintas redes se redimensionan las imágenes de entrada para que todas tengan el mismo tamaño (128x128). Las imágenes tras el redimensionamiento tendrán forma cuadrada y mantendrán sus tres canales RGB.

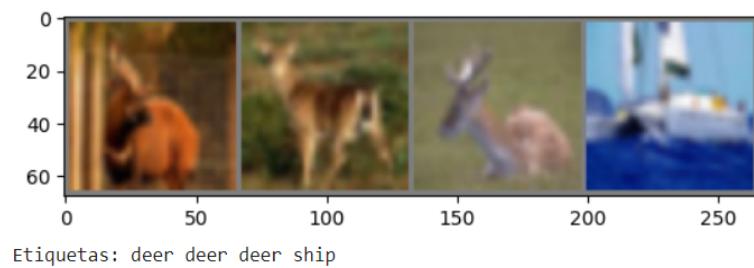


Figura 13: Imagen del conjunto de datos preprocesadas

## 6.2. Implementación de las GAN

La implementación de las dos redes que voy a utilizar para llevar a cabo la generación de imágenes se ha realizado con Tensorflow y Keras y se ha utilizado la GPU T4 implementada a través de Google Drive

### 6.2.1. Preprocesado de datos la red GAN

La red GAN preprocesa los datos de la manera explicada en el apartado 4.1 de este tfg, dicha red recibe un total 60.000 imágenes del conjunto CIFAR-10 que recibe lotes de 80 imágenes produciéndose así un total de 750 lotes de igual tamaño.

Se ha elegido un lote de ese tamaño ya que un lote más pequeño puede resultar en una calidad inferior de las imágenes generadas y que el entrenamiento sea menos estable al provocar grandes fluctuaciones en la actualización de pesos lo cual es muy importante en estas redes ya que se necesita un equilibrio entre generador y discriminador, como ya se vio en el apartado del marco teórico.

### 6.2.2. Arquitectura e hiperparámetros

La arquitectura e hiperparámetros de la red GAN se puede resumir en las dos redes que la componen, el generador y el discriminador.

Primeramente, se va a ver el generador cuyo código python es el siguiente:

```
1 #-----GENERADOR-----#
2 #1. importaciones
3 import torch.nn as nn
4 #2. Definicion de hiperparametros
5
6 latent_dim = 100          #tamaño del vector de entrada)
7 ngf = 64                  # Número de filtros en la primera capa
8 image_size = 64            # Tamaño de la imagen de salida 64x64 píxeles
9 num_channels = 3           # 3 porq es RGB
10
11 #3.Creacion de la red
12 class Generator(nn.Module):
13     def __init__(self, latent_dim, ngf, num_channels):
14         super(Generator, self).__init__()
15         self.main = nn.Sequential(
16
17     #
18
19         # Capa 1: ConvTranspose2d
20         nn.ConvTranspose2d(latent_dim, ngf * 8, kernel_size=4, stride=1,
21                         → padding=0, bias=False),
22         nn.BatchNorm2d(ngf * 8),
23         nn.ReLU(True),
24
25         # Capa 2: ConvTranspose2d
```

```

25     nn.ConvTranspose2d(n gf * 8, n gf * 4, kernel_size=4, stride=2,
26         ↳ padding=1, bias=False),
27     nn.BatchNorm2d(n gf * 4),
28     nn.ReLU(True),
29
29     # Capa 3: ConvTranspose2d
30     nn.ConvTranspose2d(n gf * 4, n gf * 2, kernel_size=4, stride=2,
31         ↳ padding=1, bias=False),
32     nn.BatchNorm2d(n gf * 2),
33     nn.ReLU(True),
34
34     # Capa 4: ConvTranspose2d
35     nn.ConvTranspose2d(n gf * 2, n gf, kernel_size=4, stride=2, padding=1,
36         ↳ bias=False),
37     nn.BatchNorm2d(n gf),
38     nn.ReLU(True),
39
39     # Capa de salida: ConvTranspose2d
40     nn.ConvTranspose2d(n gf, num_channels, kernel_size=4, stride=2,
41         ↳ padding=1, bias=False),
42     nn.Tanh() # Salida normalizada entre [-1, 1] para coincidir con la
        ↳ normalización del conjunto de datos
43
44 def forward(self, x):
45     return self.main(x)

```

Se definen los hiperparametros del generador que son principalmente 4:

- **latent dim:** Primeramente, se define el tamaño del vector de entrada que alimenta a la red (**latent dim**), dicho vector proviene de un espacio latente y es usado por el generador para producir una imagen, siendo un espacio latente una representación comprimida y abstracta de las características que el modelo aprenderá a generar como imágenes. La dimensión de este vector determina la complejidad y la capacidad de variación en las imágenes generadas.
- **nfg:** El **nfg** significa numero de generador de filtros traducido al español especifica el número de filtros o **kernels** en la primera capa convolucional del generador. Este valor determina la capacidad del generador para capturar y procesar características de las imágenes.
- **image size :** El **image size** define las dimensiones de las imágenes que el generador debe producir, en este caso de 64x64. Este valor es muy importante en el entrenamiento ya que determina el nivel de detalle que las imágenes pueden tener. Imágenes más grandes pueden representar detalles más finos pero a costa de un generador y discriminador mas complejos y de un uso mayor de la memoria
- **num channels:** **num channels** define el número de canales en las imágenes generadas.

En imágenes de color RGB, el número de canales es 3, porque cada píxel tiene tres componentes: rojo, verde y azul

Sin embargo, este generador contiene a su vez más hiperparámetros implícitos que se van a definir a continuación entre los cuales destacan los dos últimos que son funciones de activación de las diferentes capas, teniendo todas las capas de mi red la función de activacion RELU a excepción de la última que es la función de la tangente hiperbólica(tanh).

- **kernel size:** Indica el tamaño del filtro utilizado en cada operación de convolución transpuesta. En este caso, cada capa tiene un tamaño de kernel de 4x4. Esto afecta a la cantidad de detalles que se pueden capturar y generar en cada capa, pudiendo así capturar características mas grandes un kernel más grande mientras que, un kernel menor se enfoca en detalles más finos
- **stride:** Determina cuánto se desplaza el filtro en cada paso de la convolución, en este caso, el stride es 1 para la primera capa y 2 para las siguientes. Un stride más grande (como 2) incrementa el tamaño de la imagen de salida, mientras que un stride más pequeño (como 1) mantiene el tamaño más controlado. Este hiperparámetro es crucial a la hora de decir cómo se expande la imagen a medida que avanza por las capas de la red.
- **padding:** Añade bordes alrededor de la imagen antes de aplicar la convolución, lo cual permite controlar el tamaño de la imagen de salida. Un padding=0 implica que la imagen no se rellena mientras que un padding=1, añade un borde de un píxel alrededor de la imagen.
- **bias:** Es un término de sesgo que indica si se añade un término de sesgo (bias) en las operaciones de convolución. Si se ha configurado con el parámetro False no se añade un término de sesgo, lo cual es útil para simplificar la red y acelerar el entrenamiento especialmente cuando se utilizan capas de normalización por lotes como es en este caso.
- **nn.BatchNorm2d:** Es una capa que normaliza la salida de cada convolución para mejorar la estabilidad del entrenamiento y acelerar la convergencia. Este hiperparámetro al igual que el stride es crucial ya que ayuda a estabilizar la propagación del gradiente haciendo que el aprendizaje de la red sea más eficiente
- **nn.ReLU(True):** Es una función de activación que introduce no linealidad en la red. El argumento usado (inplace=True) tiene como objetivo optimizar el uso de la memoria al hacer que la operación sea en el momento
- **nn.Tanh():** Como ya se ha mencionado se usa en la capa de salida(la última) con el objetivo de escalar los valores de los píxeles entre -1 y 1. Esto hace que las salidas de la red estén en el rango correcto para compararse con las imágenes reales normalizadas, facilitando el trabajo del discriminador y optimizando la red

Es imposible entender el funcionamiento de la arquitectura de la red generadora sin entender las transformaciones que sufre la entrada a medida que va pasando por las diferentes capas que componen a la red. La secuencia de capas está diseñada para convertir el vector

latente de baja dimensión en una imagen de mayor resolución, aumentando gradualmente el tamaño espacial del tensor de salida a medida que pasa por las sucesivas capas.

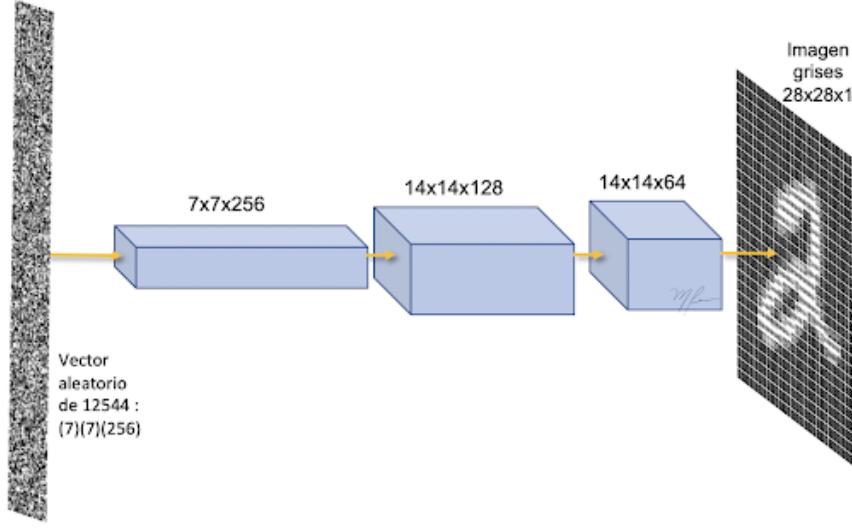


Figura 14: Imagen que muestra arquitectura de un generador

La primera capa tiene como entrada un vector latente y lo expande en un mapa de características con alta dimensionalidad de canales.

En las capas intermedias 2,3, y 4 se incrementa la resolución espacial de la imagen, mientras que simultáneamente reducen el número de canales. Esto tiene como consecuencia una simplificación de la representación a medida que se aproxima a la salida final.

Finalmente, en la capa de salida se genera una imagen en el espacio RGB (3 canales) con la resolución deseada, 64x64 píxeles en este caso.

Como se puede ver en el código cada una de las capas está compuesta a su vez por dos subcapas **ConvTranspose2d** y **BatchNorm2d**. La primera, la capa de convolución transpuesta toma una entrada de baja resolución y la expande a una resolución más alta a través de la operación de convolución transpuesta, que se encarga de aumentar la resolución espacial de la imagen.

La segunda, ajusta las salidas de la convolución transpuesta para tener una media cercana a 0 y una desviación estándar cercana a 1 ayudando así a estabilizar el entrenamiento y permitiendo que el modelo se entrene con más rapidez. De tal manera que la salida de la primera capa pasa por esta segunda capa normalizando las activaciones.

Todo esto tiene como resultado que la entrada pase a través de una capa ConvTranspose2d, que aumenta la resolución espacial de la imagen. Posteriormente, la salida de esa operación pasa a través de una capa BatchNorm2d, que normaliza las activaciones para asegurar que se mantengan dentro de un rango manejable y que el entrenamiento sea efectivo. Por último, se aplica una función de activación ReLU para introducir no linealidad, permitiendo que la red aprenda relaciones más complejas. Esto se cumple para todas las capas a excepción de la capa de salida donde la función de salida ReLU cambia a tanh con el objetivo de comprimir las salidas al intervalo  $[-1,1]$  el sentido de hacer esto es que las imágenes generadas tengan valores de píxel adecuados para compararse con las imágenes reales.

Se ha empleado esta combinación de ConvTranspose2d, BatchNorm2d, ReLU y Tanh por el hecho de que ha sido probada en muchos estudios.

Resumen del Generador:

| Layer (type)                          | Output Shape       | Param #   |
|---------------------------------------|--------------------|-----------|
| <hr/>                                 |                    |           |
| ConvTranspose2d-1                     | [ -1, 512, 4, 4]   | 819,200   |
| BatchNorm2d-2                         | [ -1, 512, 4, 4]   | 1,024     |
| ReLU-3                                | [ -1, 512, 4, 4]   | 0         |
| ConvTranspose2d-4                     | [ -1, 256, 8, 8]   | 2,097,152 |
| BatchNorm2d-5                         | [ -1, 256, 8, 8]   | 512       |
| ReLU-6                                | [ -1, 256, 8, 8]   | 0         |
| ConvTranspose2d-7                     | [ -1, 128, 16, 16] | 524,288   |
| BatchNorm2d-8                         | [ -1, 128, 16, 16] | 256       |
| ReLU-9                                | [ -1, 128, 16, 16] | 0         |
| ConvTranspose2d-10                    | [ -1, 64, 32, 32]  | 131,072   |
| BatchNorm2d-11                        | [ -1, 64, 32, 32]  | 128       |
| ReLU-12                               | [ -1, 64, 32, 32]  | 0         |
| ConvTranspose2d-13                    | [ -1, 3, 64, 64]   | 3,072     |
| Tanh-14                               | [ -1, 3, 64, 64]   | 0         |
| <hr/>                                 |                    |           |
| Total params: 3,576,704               |                    |           |
| Trainable params: 3,576,704           |                    |           |
| Non-trainable params: 0               |                    |           |
| <hr/>                                 |                    |           |
| Input size (MB): 0.00                 |                    |           |
| Forward/backward pass size (MB): 3.00 |                    |           |
| Params size (MB): 13.64               |                    |           |
| Estimated Total Size (MB): 16.64      |                    |           |
| <hr/>                                 |                    |           |

Figura 15: Resumen de Tensorflow de la arquitectura del generador implementado para la GAN

Por último la función forward se encarga de inicializar la red teniendo como salida la imagen sintética.

Una vez visto el generador en profundidad se va a proceder a explicar el código del discriminador de manera más ligera ya que la estructura de la red se parece.

```

1  #-----RED DISCRIMINADORA-----#
2  #Hiperparametros
3
4  ndf = 64
5  num_channels = 3
6
7  #Creacion de la red
8  class Discriminator(nn.Module):
9      def __init__(self, ndf, num_channels):
10         super(Discriminator, self).__init__()
11         self.main = nn.Sequential(
12             # Entrada: Imagen de num_channels x 64 x 64

```

```

13
14     # Capa 1: Conv2d
15     nn.Conv2d(num_channels, ndf, kernel_size=4, stride=2, padding=1,
16             → bias=False),
17     nn.LeakyReLU(0.2, inplace=True),
18
19     # Capa 2: Conv2d
20     nn.Conv2d(ndf, ndf * 2, kernel_size=4, stride=2, padding=1,
21             → bias=False),
22     nn.BatchNorm2d(ndf * 2),
23     nn.LeakyReLU(0.2, inplace=True),
24
25     # Capa 3: Conv2d
26     nn.Conv2d(ndf * 2, ndf * 4, kernel_size=4, stride=2, padding=1,
27             → bias=False),
28     nn.BatchNorm2d(ndf * 4),
29     nn.LeakyReLU(0.2, inplace=True),
30
31     # Capa 4: Conv2d
32     nn.Conv2d(ndf * 4, ndf * 8, kernel_size=4, stride=2, padding=1,
33             → bias=False),
34     nn.BatchNorm2d(ndf * 8),
35     nn.LeakyReLU(0.2, inplace=True),
36
37     # Capa de salida: Conv2d
38     nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=1, padding=0,
39             → bias=False),
40     nn.Sigmoid()  # Salida en el rango 0-1
41
42
43     def forward(self, x):
44         return self.main(x)

```

Debido a que la arquitectura del generador y el discriminador es parecida y la mayoría de conceptos que presenta el código se han explicado en el generador voy a explicar el código del discriminador viendo sus diferencias con el generador y detallando en aquellos conceptos que sean nuevos.

Como ya se ha explicado en el marco teórico la función del generador es la creación de imágenes cada vez más reales y la función del discriminador es la detección de imágenes las cuales sean a medida que avanza el entrenamiento mas difícil de distinguir hasta llegar al punto de no poder distinguirlas. Cumpliendo con la analogía del policía y el falsificador que se expuso.

Una vez hecha esta introducción se puede ver la primera diferencia y es que la salida del discriminador es un valor booleano(True/False) y su entrada es la imagen generada por el generador.

El discriminador utiliza capas de convolución (Conv2d) para reducir la resolución de la

imagen y extraer características a medida que los datos fluyen a través de la red ayudando así a identificar patrones complejos y determinar si la imagen es real o generada, mientras que el generador utilizaba capas de convolución transpuesta para aumentar la resolución. De tal manera que como se ha explicado, la entrada es, en este caso, una imagen de 64x64 píxeles que tiene como salida el valor booleano.

La última diferencia se encuentra en la función de activación usada en las sucesivas capas teniendo todas la misma a excepción de la capa de salida, de igual manera que pasaba en el generador. Las funciones de activación son:

#### ■ LeakyReLU

Es la función de activación de tanto la capa de entrada como de las capas intermedias.

Su expresión matemática es la siguiente:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha x & \text{si } x \leq 0 \end{cases}$$

Donde  $\alpha$  es normalmente un valor pequeño como 0.2, cuyo propósito es permitir que haya un pequeño gradiente para valores negativos en lugar de anularlos completamente, esto es muy importante ya que es la principal diferencia con la función ReLU, se va a ver más adelante el porqué del uso de esta función en vez de la función ReLU estándar.

Tiene como objetivo mejorar la estabilidad del entrenamiento y la capacidad del discriminador para aprender representaciones complejas, lo que es crucial para distinguir eficazmente entre imágenes reales y generadas introduciendo no linealidad en la red y superando algunas de las limitaciones de la función ReLU estándar.

En la función ReLU estándar, cualquier valor de entrada que sea negativo se convierte en cero, lo que puede llevar a un problema conocido como neurona muerta. Esto, ocurre cuando una neurona solo produce ceros y nunca se activa, lo que impide que participe en el aprendizaje, especialmente si los gradientes se desvanecen y es por esto por lo que se cambió a la función LeakyReLU la cual introduce una pequeña pendiente para los valores negativos en lugar de simplemente convertirlos en cero. Esto asegura que las neuronas sigan activas y puedan participar en el aprendizaje incluso si reciben valores negativos, ayudando así también al problema del desvanecimiento del gradiente.

#### ■ Sigmoid

La función sigmoid es la función de activación de la capa de salida y su expresión matemática es la siguiente:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Donde  $x$  es la entrada a la función, que puede ser cualquier número real. La función transforma este valor en un número entre 0 y 1.

Tiene como objetivo convertir la salida de la red, que puede ser cualquier número real, en un valor que represente una probabilidad haciendo posible que el discriminador pueda clasificar una imagen como real o como falsa.

Funciona de tal manera que cualquier valor de entrada lo convierte en un número cuyo rango estan entre 0 y 1 conviertiendolo asi en una probabilidad. Un valor cercano a 1 indica una alta probabilidad de que la imagen generada sea real, mientras que un valor cercano a 0 indica una alta probabilidad de que la imagen generada sea falsa.

| Resumen del Discriminador: |                    |           |
|----------------------------|--------------------|-----------|
| Layer (type)               | Output Shape       | Param #   |
| Conv2d-1                   | [ -1, 64, 32, 32]  | 3,072     |
| LeakyReLU-2                | [ -1, 64, 32, 32]  | 0         |
| Conv2d-3                   | [ -1, 128, 16, 16] | 131,072   |
| BatchNorm2d-4              | [ -1, 128, 16, 16] | 256       |
| LeakyReLU-5                | [ -1, 128, 16, 16] | 0         |
| Conv2d-6                   | [ -1, 256, 8, 8]   | 524,288   |
| BatchNorm2d-7              | [ -1, 256, 8, 8]   | 512       |
| LeakyReLU-8                | [ -1, 256, 8, 8]   | 0         |
| Conv2d-9                   | [ -1, 512, 4, 4]   | 2,097,152 |
| BatchNorm2d-10             | [ -1, 512, 4, 4]   | 1,024     |
| LeakyReLU-11               | [ -1, 512, 4, 4]   | 0         |
| Conv2d-12                  | [ -1, 1, 1, 1]     | 8,192     |
| Sigmoid-13                 | [ -1, 1, 1, 1]     | 0         |

|                       |           |
|-----------------------|-----------|
| Total params:         | 2,765,568 |
| Trainable params:     | 2,765,568 |
| Non-trainable params: | 0         |

|                                  |       |
|----------------------------------|-------|
| Input size (MB):                 | 0.05  |
| Forward/backward pass size (MB): | 2.31  |
| Params size (MB):                | 10.55 |
| Estimated Total Size (MB):       | 12.91 |

Figura 16: Resumen de Tensorflow de la arquitectura del discriminador implementado para la red GAN

En la siguiente imagen se puede ver un resumen de la arquitectura explicada:

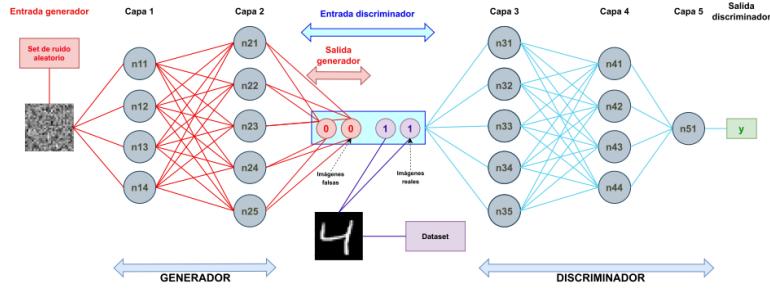


Figura 17: Ejemplo análogo de la arquitectura de la red GAN que se va a implementar

### 6.2.3. Entrenamiento y optimización

Antes de empezar con el entrenamiento de la red GAN es conveniente definir la inicialización de pesos y la optimización de la red.

La inicialización de pesos es el proceso de asignar valores iniciales a los pesos de las capas de una red neuronal antes de que comience el entrenamiento.

Esto, tiene como objetivo principal el hecho de garantizar que la red comience el proceso de entrenamiento de manera estable, lo cual es de vital importancia sobre todo en las redes GAN ya que son propensas la inestabilidad debido a la competencia generador-discriminador, favoreciendo de esta manera el aprendizaje.

Además la inicialización de pesos también implica tanto una gran ayuda a la hora de evitar dos problemas habituales ya explicados, la explosión del gradiente y el desvanecimiento del gradiente como una convergencia mas eficiente.

El hecho de evitar los problemas de explosión y desvanecimiento del gradiente se consigue porque una inicialización inadecuada puede causar que los gradientes se vuelvan extremadamente grandes (explosión) o extremadamente pequeños (desvanecimiento) a medida que se propagan a través de la red.

La convergencia más eficiente se consigue ya que una inicialización de pesos adecuada puede ayudar a que la red converja de manera mas rápida hacia una solución óptima, debido a que los pesos empiezan desde valores razonables que permiten el flujo adecuado de la información.

En nuestra red GAN se ha usado el optimizador Adam, que es una variante del descenso de gradiente estocástico que incorpora el momento de primer y segundo orden, lo que lo hace especialmente efectivo para entrenar redes neuronales profundas y modelos como GANs. El objetivo de la optimización en una red GAN es ajustar los pesos tanto de la rede generadora como de la discriminadora para minimizar la función de pérdida respectiva. En las redes GAN es comun elegir este optimizador por su eficacia en este tipo de escenarios debido a sus propiedades adaptativas y su capacidad para manejar gradientes ruidosos.

El codigo para implementar este optimizador es el siguiente:

```

1 # Hiperparámetros de los Optimizadores
2 learning_rate = 0.0002
3 beta1 = 0.5           # Parámetro beta1 de Adam
4 beta2 = 0.999         # Parámetro beta2 de Adam
5
6 #Opt. de generador y discriminador
7 # optimizador generador
8 optimizerG = optim.Adam(G.parameters(), lr=learning_rate, betas=(beta1, beta2))
9
10 # optimizador discri.
11 optimizerD = optim.Adam(D.parameters(), lr=learning_rate, betas=(beta1, beta2))
```

Donde el learning rate es un hiperparámetro que tiene como objetivo controlar los ajustes que se realizan a los pesos en cada paso del entrenamiento. Es usual utilizar valores pequeños como 0.0002 en redes GAN ya que así se garantiza un aprendizaje estable no haciendo que los pesos cambien de manera abrupta.

Los parámetros beta1 y beta2 controlan las tasas de decaimiento exponencial de las estimaciones del momento de primer y segundo orden en el optimizador Adam. Beta2=0.999 es el valor predeterminado para Adam y controla el momento de segundo orden, ayudando a manejar la variación en los gradientes. Beta1= 0.5 reduce la influencia del momento en comparación con su valor predeterminado (0.9), se cambia este valor para ayudar a estabilizar el entrenamiento en las redes GAN esto se debe a que el equilibrio entre generador y discriminador es delicado.

Se han elegido esos valores a los parámetros debido a que se han demostrado efectivos en numerosos estudios y aplicaciones. Sin embargo, a la hora de implementar la red se recomienda experimentar con estos valores dependiendo de la arquitectura de la red o del dataset empleado.

Por último se procede a la definición de los optimizadores del generador y del discriminador que tienen como objetivo ajustar los parámetros para, respectivamente, minimizar su pérdida y maximizar su capacidad para distinguir imágenes reales y generadas.

Una vez se han definido la inicialización de pesos y los optimizadores de la red generadora y discriminadora se procede al entrenamiento de la red GAN cuyo código se va a ir explicando poco a poco y es el siguiente:

```

1 num_epochs = 500
2 fixed_noise = torch.randn(64, latent_dim, 1, 1, device=device) # Vector de ruido
   ↳ fijo para ver la evolución del generador
3
4 G.to(device)
5 D.to(device)
6 # Almacenar métricas
7 losses_D = []
8 losses_G = []
9 fid_scores = [] # Lista para almacenar los puntajes FID
10 is_scores = [] # Lista para almacenar los puntajes IS
11
12 # Bucle de entrenamiento
13
14 for epoch in range(num_epochs):
15     for i, data in enumerate(train_loader, 0):
16
17         # (1) Actualizar el Discriminador
18
19         D.zero_grad()
20
21         # Entrenamiento con imágenes reales
22         real_images, _ = data
23         real_images = real_images.to(device)
24         batch_size = real_images.size(0)
25         real_labels = torch.ones(batch_size, device=device) # Etiquetas reales
26         output = D(real_images).view(-1) # Asegúrate de que `output` tenga
27           ↳ tamaño [batch_size]
28         loss_real = criterion(output, real_labels)
29         loss_real.backward()
30
31         # Entrenamiento con imágenes generadas
32         noise = torch.randn(batch_size, latent_dim, 1, 1, device=device) #
   ↳ Generar ruido
33         fake_images = G(noise)
34         fake_labels = torch.zeros(batch_size, device=device) # Etiquetas falsas
35         output = D(fake_images.detach()).view(-1) # Asegúrate de que `output` 
   ↳ tenga tamaño [batch_size]
36         loss_fake = criterion(output, fake_labels)
37         loss_fake.backward()
38
39         loss_D = loss_real + loss_fake
40         optimizerD.step() # Actualizar el discriminador

```

```

41      # (2) Actualizar el Generador
42
43      G.zero_grad()
44      real_labels = torch.ones(batch_size, device=device) # Queremos que el
45      ↵ discriminador piense que las imágenes falsas son reales
46      output = D(fake_images).view(-1) # Asegúrate de que `output` tenga
47      ↵ tamaño [batch_size]
48      loss_G = criterion(output, real_labels)
49      loss_G.backward()
50      optimizerG.step() # Actualizar el generador
51
52
53      # (3) Almacenar las pérdidas
54
55      losses_D.append(loss_D.item())
56      losses_G.append(loss_G.item())
57
58      # (4) Registro de las pérdidas
59
60      if i % 50 == 0:
61          print(f'Época [{epoch+1}/{num_epochs}], Paso
62          ↵ [{i}/{len(train_loader)}], Pérdida D: {loss_D.item():.4f},
63          ↵ Pérdida G: {loss_G.item():.4f}')
64
65      # (3) Almacenar las metricas
66
67      # Calcular las métricas después de cada época o en intervalos específicos
68      if epoch % 10 == 0: # Calcular cada 10 epochs
69          with torch.no_grad():
70              fake_images = G(fixed_noise).detach().cpu()
71              real_images = next(iter(train_loader))[0].to(device).cpu()
72
73              fid_value = calculate_fid(real_images, fake_images)
74              print(f'Época {epoch}, FID: {fid_value:.4f}')
75
76      fid_scores.append(fid_value)
77
78      # (4) Generar y Guardar Imágenes
79      if epoch % 10 == 0:
80          with torch.no_grad():
81              fake_images = G(fixed_noise).detach().cpu() # `fixed_noise` ya
82              ↵ está en GPU, las imágenes falsas se mueven a CPU para
83              ↵ visualización

```

```

82     grid = torchvision.utils.make_grid(fake_images, padding=2,
83         → normalize=True)
84     plt.imshow(grid.permute(1, 2, 0))
85     plt.title(f'Época {epoch+1}')
86     plt.show()

```

Primero, se procede a la definición de un conjunto de valores. Se definen el número de epochs del entrenamiento, o lo que es lo mismo, las veces que se va a entrenar la red. Posteriormente, se define el vector de ruido que se mantiene constante a lo largo de las épocas a lo que se llama fixed noise, se utiliza para generar un conjunto fijo de imágenes que permiten visualizar cómo evoluciona el generador a lo largo del entrenamiento. Para terminar con las definiciones se procede a definir un conjunto de listas los valores de las perdidas y de la métrica del valor FID a lo largo de las epochs.

Además, se ha añadido un checkpoint y el cómo cargarlo al principio del entrenamiento, esto es muy importante ya que en redes neuronales donde el entrenamiento de la red neural contiene un numero de epochs grande como es el caso el entrenamiento dura horas lo que requiere primero cada un numero de epochs guardar el entrenamiento por si ocurre cualquier fallo y poder cargarlo en caso de que ocurra este mencionado fallo o en caso de que por otras razones se haya querido desconectar el entorno de ejecución durante el entrenamiento.

Después para entender mejor la funcionalidad se ha dividido el entrenamiento en los siguientes pasos:

#### ■ Actualización del Discriminador

En esta parte de código se usa zero\_grad() para reiniciar los gradientes acumulados por el discriminador, para posteriormente iniciar el entrenamiento con imágenes reales donde se pasa como entrada un lote de imágenes reales al discriminador, calcula la pérdida (loss real) basada en la diferencia entre las etiquetas reales (real labels = 1) y la predicción del discriminador, y realiza la retropropagación para actualizar los gradientes.

Llegados a este punto se realiza un breve recordatorio de qué es la función de pérdida ya explicado en el marco teórico. La función de perdida se encarga de medir la diferencia entre las predicciones del modelo y los valores reales o deseados midiendo el desempeño del modelo. El objetivo es minimizar este valor.

Más tarde, se realiza el entrenamiento con imágenes falsas donde se genera un lote de imágenes falsas usando el generador, luego las pasa al discriminador. Se calcula la pérdida (loss fake) basada en la diferencia entre las etiquetas falsas (fake labels = 0) y la predicción del discriminador. Se realiza la retropropagación para actualizar los gradientes.

Por último se usa optimizerD.step() donde se aplica los gradientes acumulados para actualizar los pesos del discriminador.

#### ■ Actualización del Generador y almacenamiento de métricas

Se reinician los gradientes de igual manera que en el discriminador queremos engañar al discriminador para que piense que las imágenes generadas son reales. Se calculan las

pérdidas (loss G) y se realiza la retropropagación para ajustar los pesos del generador. Se realiza la retropropagación para actualizar los gradientes de igual manera que en el discriminador.

Se almacenan las métricas en las listas declaradas al inicio, esto es importante ya que lleva un registro del entrenamiento de la red neuronal que luego va a ser evaluado a través de las diferentes gráficas. Además se muestra por pantalla al ejecutar el código la epoch en la que te encuentras y los valores de pérdida del generador y del discriminador

- **Cálculo de la métrica FID**

Primero se utiliza `torch.no_grad()` para desactivar el cálculo de gradientes los cuales son innecesarios para la evaluación. Posteriormente, se generan imágenes usando fixed noise y se comparan con imágenes reales para calcular el FID y se almacena dicho valor.

- **Generación de imágenes**

Cada 10 epochs se usa el generador con el objetivo de crear un conjunto de imágenes a partir de fixed noise para ver de manera explícita la evolución de la calidad de las imágenes a medida que se entrena el modelo mostrándose en pantalla dichas imágenes.

De esta manera, este código implementa un ciclo de entrenamiento para una red GAN, donde se actualizan y evalúan tanto el generador como el discriminador. Además se incluye una evaluación de las imágenes generadas a través de la metrica FID y se guarda un listado de metricas que servirán posteriormente para evaluar el correcto funcionamiento del modelo.

### 6.3. Conclusiones

La conclusion del entrenamiento es que con la red establecida el discriminador es más potente que el generador, esto lleva a que la función de perdida del discriminador sea mínima pudiendo así diferenciar a la perfección si las imágenes son verdaderas o falsas. Esto, trae como contrapartida que el generador no es capaz de generar imágenes de buena calidad. Este problema ya se vio expuesto en la sección del marco teórico debido a que es un problema muy típico en las redes GAN. Es por ello por lo que para hacer que la red GAN funcione correctamente se van a emplear un conjunto de técnicas de mejora sobre el entrenamiento de la red, y mas concretamente, sobre el discriminador para que así se vayan retroalimentando no siendo el discriminador más potente.

Es fácil identificar este problema ya que con un número considerable de epochs va a aparecer de manera similar a esta:

Época [36/100], Paso [0/391], Pérdida D: 0.0043, Pérdida G: 6.6283  
Época [36/100], Paso [50/391], Pérdida D: 0.0044, Pérdida G: 6.8348  
Época [36/100], Paso [100/391], Pérdida D: 0.0004, Pérdida G: 10.0321  
Época [36/100], Paso [150/391], Pérdida D: 0.0011, Pérdida G: 7.4722  
Época [36/100], Paso [200/391], Pérdida D: 0.0006, Pérdida G: 8.3871  
Época [36/100], Paso [250/391], Pérdida D: 0.0006, Pérdida G: 9.3170  
Época [36/100], Paso [300/391], Pérdida D: 0.0008, Pérdida G: 7.7678  
Época [36/100], Paso [350/391], Pérdida D: 0.0005, Pérdida G: 9.8860  
Época [37/100], Paso [0/391], Pérdida D: 0.0139, Pérdida G: 8.9170  
Época [37/100], Paso [50/391], Pérdida D: 0.0050, Pérdida G: 24.4590  
Época [37/100], Paso [100/391], Pérdida D: 0.0031, Pérdida G: 7.2434  
Época [37/100], Paso [150/391], Pérdida D: 0.0001, Pérdida G: 12.0377  
Época [37/100], Paso [200/391], Pérdida D: 0.0015, Pérdida G: 7.7977  
Época [37/100], Paso [250/391], Pérdida D: 0.0008, Pérdida G: 10.5534  
Época [37/100], Paso [300/391], Pérdida D: 0.0005, Pérdida G: 9.6090  
Época [37/100], Paso [350/391], Pérdida D: 0.0002, Pérdida G: 11.5883  
Época [38/100], Paso [0/391], Pérdida D: 0.0016, Pérdida G: 8.4204  
Época [38/100], Paso [50/391], Pérdida D: 0.0000, Pérdida G: 35.6920  
Época [38/100], Paso [100/391], Pérdida D: 0.0000, Pérdida G: 35.5249  
Época [38/100], Paso [150/391], Pérdida D: 0.0000, Pérdida G: 35.5307  
Época [38/100], Paso [200/391], Pérdida D: 0.0000, Pérdida G: 35.3514  
Época [38/100], Paso [250/391], Pérdida D: 0.0000, Pérdida G: 35.2266  
Época [38/100], Paso [300/391], Pérdida D: 0.0000, Pérdida G: 34.9620  
Época [38/100], Paso [350/391], Pérdida D: 0.0000, Pérdida G: 34.7640  
Época [39/100], Paso [0/391], Pérdida D: 0.0000, Pérdida G: 34.5031  
Época [39/100], Paso [50/391], Pérdida D: 0.0000, Pérdida G: 34.1386  
Época [39/100], Paso [100/391], Pérdida D: 0.0000, Pérdida G: 33.3650  
Época [39/100], Paso [150/391], Pérdida D: 0.0022, Pérdida G: 7.4535

En este registro de valores de la función de pérdida se ve la evolución de la red entre la epoch 36 y 39 viendo perfectamente cómo la función de perdida del generador se dispara y la del discriminador es prácticamente 0.

### 6.3.1. Uso de técnicas de mejora

Como se ha podido ver en las conclusiones del entrenamiento se ha producido un problema típico de las redes GAN, el overfitting, donde el discriminador se vuelve tan fuerte que el generador no puede mejorar, lo que lleva a que el generador produzca imágenes de muy baja calidad, haciendo que el entrenamiento no sea válido ya que el generador no tiene margen de mejora.

Como ya se ha mencionado este es un problema común y se va a ver un conjunto de técnicas para mitigarlo y el cómo implementarlas:

#### ■ Suavizado de etiquetas

Es una técnica sencilla pero efectiva para mitigar el overfitting. La idea básica es que en lugar de etiquetar las imágenes reales como 1 y las imágenes generadas como 0, les asignas etiquetas ligeramente modificadas, como 0.9 y 0.1 respectivamente. Esto evita que el discriminador se vuelva excesivamente confiado y le dificulta memorizar las etiquetas exactas.

Su implementación es la siguiente:

```
1 # Entrenamiento con imágenes reales
2 real_labels = torch.full((batch_size,), 0.9, device=device) # antes era 1.0
3 output = D(real_images).view(-1) # Asegúrate de que `output` tenga tamaño
4     ↪ del batch_size
5 loss_real = criterion(output, real_labels)
6 loss_real.backward()
7
7 # Entrenamiento con imágenes generadas
8 fake_labels = torch.full((batch_size,), 0.1, device=device) #antes era 0.0
9 output = D(fake_images.detach()).view(-1)
10 loss_fake = criterion(output, fake_labels)
11 loss_fake.backward()
```

De tal manera que el entrenamiento, por ahora, permanece exactamente igual y las únicas sentencias que se han cambiado son:

```
1 real_labels = torch.full((batch_size,), 0.9, device=device)
2 fake_labels = torch.full((batch_size,), 0.1, device=device)
```

#### ■ Técnica Dropout

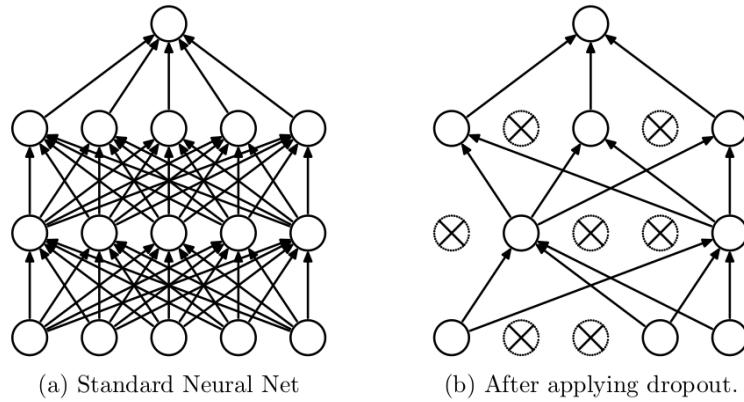


Figura 18: Representacion del Dropout

Esta técnica funciona desactivando, es decir, estableciendo a cero aleatoriamente una fracción de las unidades en una capa durante el entrenamiento, ignorando así algunas neuronas. Esto, es útil porque obliga a la red a no depender demasiado de ninguna característica particular, promoviendo la generalización.

La implementación también es muy sencilla. Consiste en, donde se ha definido el discriminador y sus sucesivas capas, en este caso 5, poner la siguiente sentencia a aquellas capas donde se quiera aplicar esta técnica en mi entrenamiento se ha aplicado a todas las capas menos a la capa de salida, la cual no tiene sentido aplicarle esta técnica ya que su función no está relacionada con el overfitting. La sentencia mencionada es:

```
1    nn.Dropout(0.3)
```

Donde el valor 0.3 representa la probabilidad de desactivar cada neurona de forma individual. Esto no significa que exactamente el treinta por ciento de las neuronas se desactiven en cada pasada, sino que cada neurona individualmente tiene un treinta por ciento de probabilidad de ser apagada.

Resaltar el hecho de que también se puede aplicar, como es lógico, dropout al generador. Sin embargo, como ocurre en muchas redes GAN, no tiene sentido aplicarlo.

## ■ Ajustar la tasa de aprendizaje

La tasa de aprendizaje .

A la hora de definirla junto con el optimizador antes se definía la misma tasa de aprendizaje para las dos redes, pero, debido al overfitting se va a proceder a bajar la tasa de aprendizaje del discriminador, teniendo como implementacion en mi red neuronal:

```
1 #-----Optimizadores-----#
2 import torch.optim as optim
3
4 # Hiperparámetros
5 learning_rate_G = 0.0002    #
6 learning_rate_D = 0.0001    # Tasa de aprendizaje del Discriminador (ajustada
    ↳ para evitar overfitting)
7 beta1 = 0.5
8 beta2 = 0.999
9
10 #Opt. de generador y discriminador
11 optimizerG = optim.Adam(G.parameters(), lr=lr=learning_rate_G, betas=(beta1,
    ↳ beta2))
12 optimizerD = optim.Adam(D.parameters(), lr=learning_rate_D, betas=(beta1,
    ↳ beta2))
```

También se podría haber aumentado la tasa de aprendizaje del generador para ayudar a mitigar este problema, pero no se ha considerado necesario

## ■ Normalización espectral(Spectral Normalization)

Es una técnica utilizada para estabilizar el entrenamiento de redes generativas adversariales (GANs) al restringir las singularidades de las capas del discriminador evitando que el discriminador se vuelva demasiado poderoso y cause problemas como el colapso del modelo.

La implementación, como mucha de las técnicas vistas es muy sencilla hay que añadir la siguiente sentencia al conjunto de capas del discriminador:

```
1 utils.spectral_norm(nn.Conv2d(num_channels, ndf, kernel_size=4, stride=2,
    ↳ padding=1, bias=False))
```

Esta función restringe el valor máximo de la singularidad de la matriz de pesos de cada capa, estabilizando el entrenamiento del discriminador. Esta tecnica se aplica a todas las capas convolucionales del discriminador.

### 6.3.2. Conclusiones una vez hecha la mejora

Para tomar conclusiones del etrenamiento, primero vamos a analizar la grafica de la perdida del discriminador

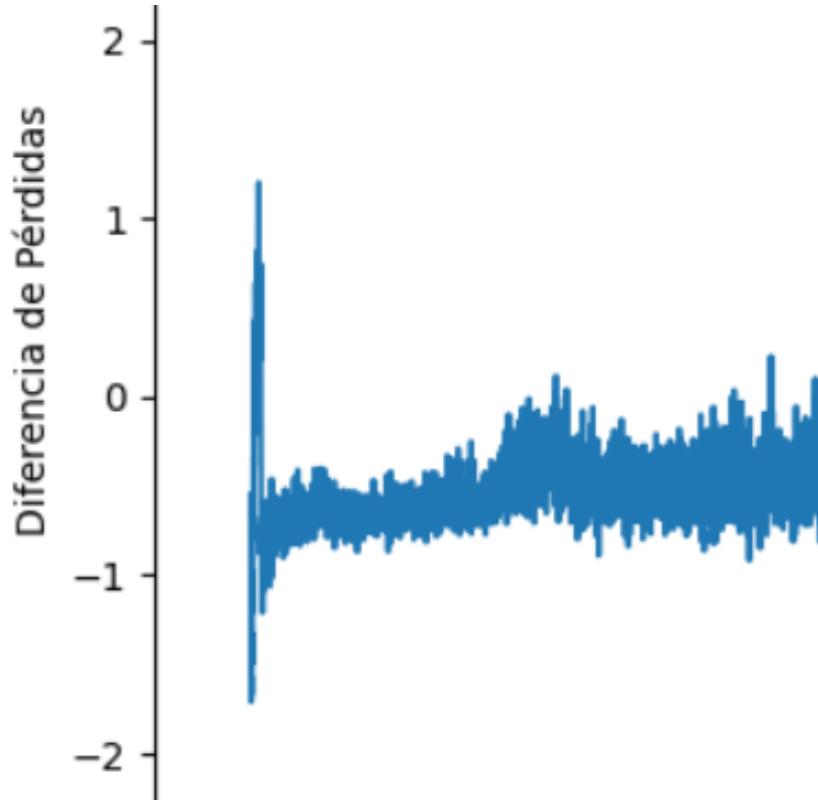


Figura 19: Gráfica de la pérdida del generador y discriminador

Este gráfico muestra cómo evolucionan las pérdidas del generador y del discriminador a lo largo del entrenamiento. Indica si el modelo está entrenando de manera estable de tal manera que a través de ella se detectan desequilibrios entre el generador y el discriminador. Si, por ejemplo una de las pérdidas es mas alta que la otra indica que una de las redes esta dominando el entrenamiento y hay que reajustar o bien los hiperparámetros o bien otros aspectos de la arquitectura de la red como el optimizador.

Seguidamente para verlo de manera más explícita se van a ir mostrando resultados del generador a medida que avanzan las epochs y sus respectivos valores de la métrica FID.

Cabe resaltar el hecho de que, a excepción del principio donde es palpable la mejora va a llegar un punto donde no se nota casi las mejoras a pesar de que existen tal y como muestra la métrica FID. Esto, se debe a dos razones, la primera es debido a que el conjunto de datos muestra imágenes borrosas de por sí al estar en una resolución baja por lo tanto si se parecen a las reales, tienen que estar borrosas. El segundo motivo es mencionar el hecho de que, en el punto óptimo, un FID de valor 80.7495 sigue siendo un FID alto.

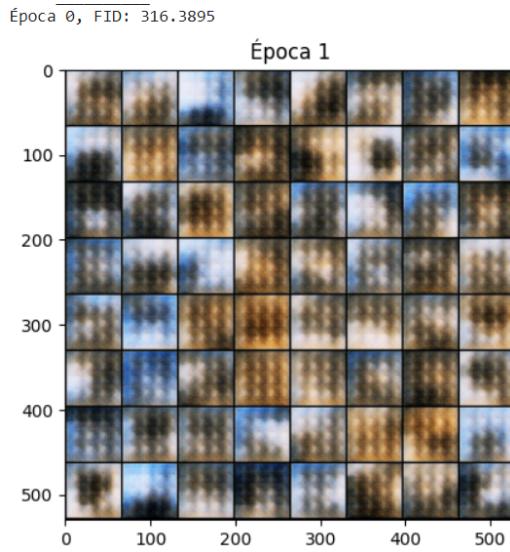


Figura 20: Imagen epoch 1

En la época 1 la métrica FID de la red toma el valor de 316.3895 obteniéndose las imágenes mostradas en la figura 19

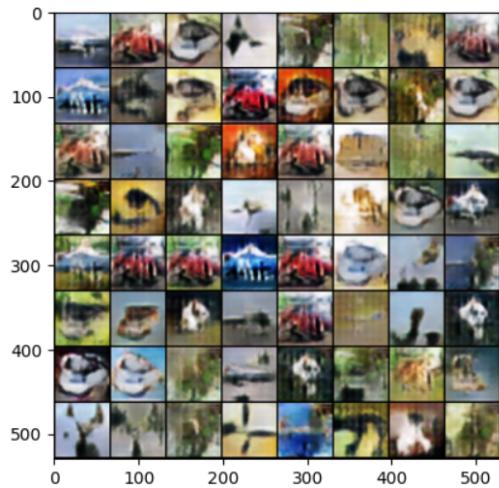


Figura 21: Imagen epoch 40

En la época 40 la métrica FID de la red toma el valor de 190.7537 obteniéndose las imágenes mostradas en la figura 20

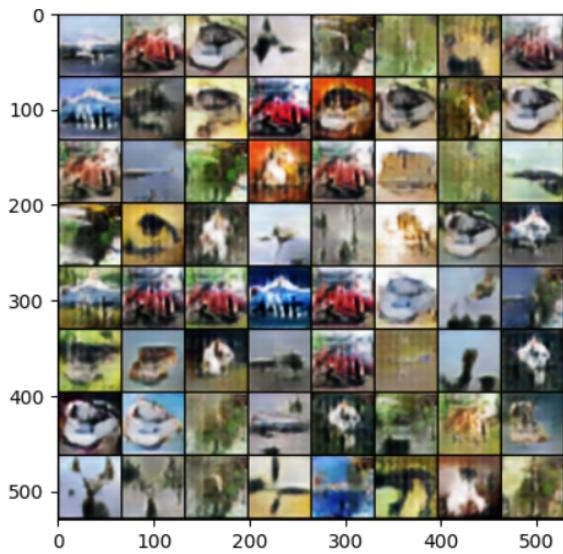


Figura 22: Imagen epoch 80

En la época 80 la métrica FID de la red toma el valor de 167.5387 obteniéndose las imágenes mostradas en la figura 20

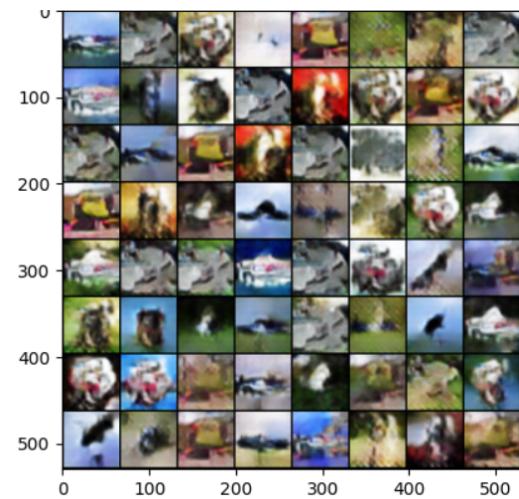


Figura 23: Imagen epoch 150

En la época 150 la métrica FID de la red toma el valor de 127.4375 obteniéndose las imágenes mostradas en la figura 20



Figura 24: Imagen epoch 250

En la época 250 la métrica FID de la red toma el valor de 105.6409 obteniéndose las imágenes mostradas en la figura 20

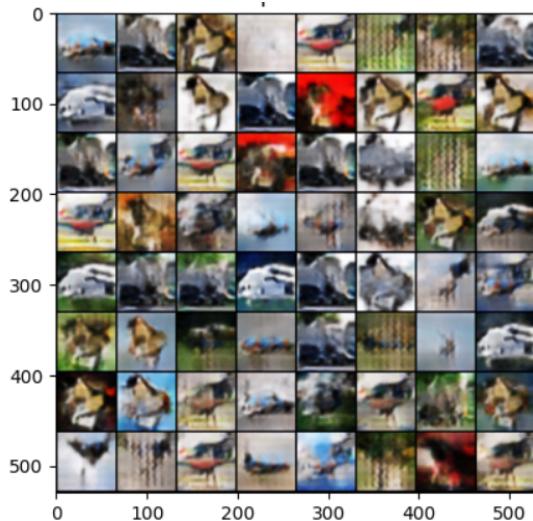


Figura 25: Imagen epoch 320

En la época 320 la métrica FID de la red toma el valor de 98.9053 obteniéndose las imágenes mostradas en la figura 20

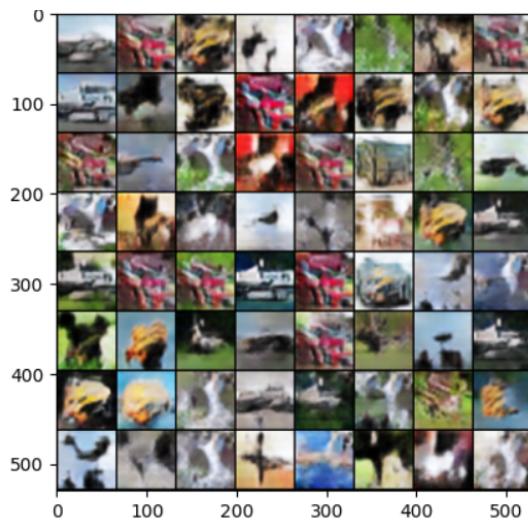


Figura 26: Imagen epoch 450

En la época 450 la métrica FID de la red toma el valor de 190.7537 obteniéndose las imágenes mostradas en la figura 20

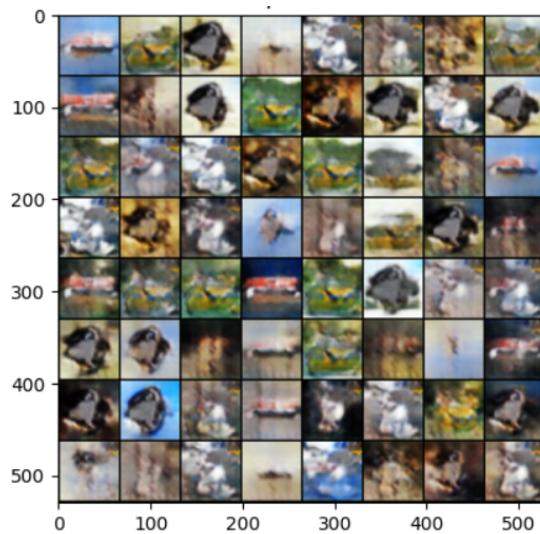


Figura 27: Imagen epoch 500

En la época 500 la métrica FID de la red toma el valor de 190.7537 obteniéndose las imágenes mostradas en la figura 20.

### 6.3.3. Conclusiones del uso de tecnicas de mejora

Como conclusión de haber aplicado las técnicas de mejora se tiene un discriminador el cual ya no es tan potente, como es lógico. Esto, produce que el generador continuar su aprendizaje produciendo imágenes cada vez más realistas. Sin embargo, a pesar del gran avance que suponen estas medidas ya que permiten un entrenamiento estable, todavía no son lo suficientemente buenas para el conjunto de datos de entrada tal y como se puede ver en la gráfica del FID:

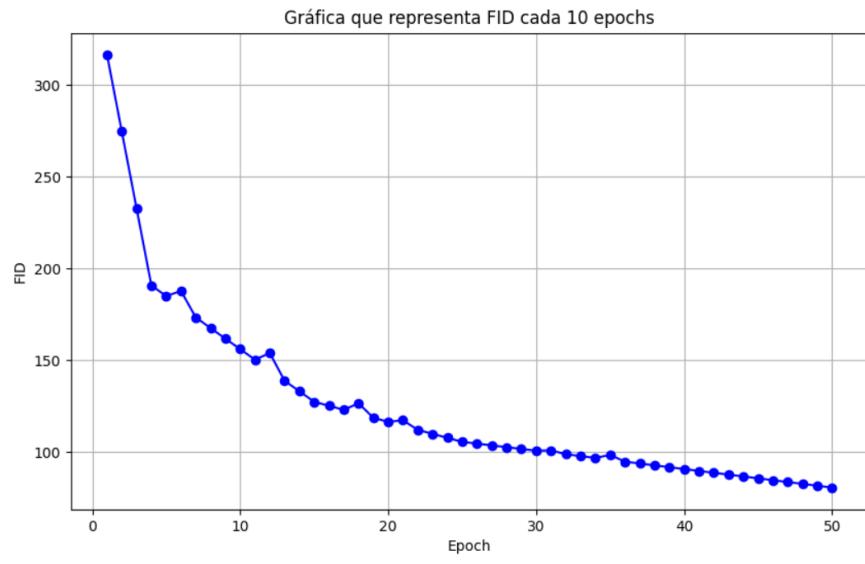


Figura 28: Evolución del FID a medida que van avanzando las epochs

Hay que tener en cuenta que, como ya se vio en el código la métrica FID se guarda cada 10 epochs por lo tanto si hemos ejecutado 500 epochs tenemos un total de 50 valores de métricas en los cuales apenas hay variación y es decreciente en su mayoría debido a que entre un valor y otro hay mucha distancia. Sin embargo si esta gráfica mostrara epoch a epoch veríamos mucha más variación y no siendo decreciente, como es normal y lógico en el entrenamiento.

De esto, tomamos como conclusión que todavía esta red neuronal si bien produce imágenes que se parecen al valor del FID más bajo es de 80.743 el cual, sigue siendo un valor muy alto no produciendo imágenes lo suficientemente buenas para considerarse casi idénticas a las reales. Es por esta misma razón por la cual se quiere hacer uso de los transformers para mejorar estas redes.

## 6.4. Implementación de las GAN con los transformers

En esta sección se va a explicar el cómo implementar un transformer a una red GAN y las múltiples ventajas que esto conlleva. Antes de empezar a explicar detalladamente se recuerda que un transformer es un tipo de arquitectura de red neuronal diseñada para manejar datos secuenciales, aunque es más conocida por su capacidad de trabajar con secuencias de texto en tareas como traducción automática, resumen de texto y generación de lenguaje. Fue introducido en el paper “Attention is All You Need” por Vaswani et al. en 2017, y desde entonces ha revolucionado el campo del procesamiento del lenguaje natural (NLP), a pesar de esto, vamos a ver que no es el único campo donde tiene utilidad y vamos a proceder a ver su utilidad en una red GAN. A pesar de esta breve introducción, para tener una explicación teórica más general recomiendo ir a la parte de marco teórico en la cual viene una descripción más específica. Además también se recuerda que el preprocesamiento de los datos es análogo al de la red GAN, ya explicado en este capítulo.

### 6.4.1. Arquitectura de un Transformer

El Transformer es una arquitectura que se basa en mecanismos de atención, eliminando la necesidad de recurrencias o convoluciones. Su estructura está diseñada para manejar secuencias de datos, permitiendo que el modelo procese todos los elementos de la secuencia de forma paralela, en lugar de secuencialmente.

Su arquitectura está dividida en dos partes el codificador y el decodificador.

#### ▪ Codificador(encoder)

El codificador se encarga de procesar la secuencia de entrada y generar una representación de las características.

Cada bloque del codificador en el Transformer se compone de diferentes partes que se van a ver a continuación.

Para empezar se usa un **mecanismo de atención** en el que se compara cada token de la secuencia de entrada con el resto, permitiendo al modelo literalmente, prestar atención, a diferentes partes de la secuencia mientras se procesa el token de entrada. El funcionamiento detallado de esto se puede ver en el apartado del marco teórico.

Seguidamente, se usa **mecanismo de atención(Multi-Head Attention)** donde en lugar de calcular una única matriz de atención el transformer divide las matrices Q, K y V en múltiples cabezas de atención, de ahí el nombre multi-head. Esto se realiza con el objetivo de que el modelo capture diferentes tipos de relaciones donde cada cabeza es una instancia de operación de atención

Posteriormente, se **normaliza** mediante una capa de normalización por lotes la suma de la salida de la atención multi-cabeza con la entrada original. Su fórmula es la siguiente:

$$\text{Output} = \text{LayerNorm}(X + \text{MultiHead}(Q, K, V)) \quad (2)$$

Cada token procesado por el bloque de atención pasa por una red completamente conectada con dos capas lineales y una función de activación intermedia (generalmente ReLU).

Por último, similar a la atención, la salida de la FFN se suma con la entrada original del bloque de atención y se normaliza nuevamente

- **Decodificador(decoder)** El decoder se encarga de generar la secuencia de salida a partir de la representación interna proporcionada por el codificador donde funciona igual que el codificador pero con la diferencia que existe un mecanismo de "máscara" para evitar que un token vea los tokens futuros en la secuencia. Esto, tiene como objetivo que los tokens futuros no influyan en la predicción de los tokens actuales.

Para terminar con la arquitectura del transformer

#### 6.4.2. Transformer en las redes GAN

Seguramente tanto en el marco teórico como en la arquitectura del apartado anterior, te has preguntado cómo es posible mezclar los transformer con las redes GAN si los transformer originalmente sirven para tareas de procesamiento de secuencias, como en el caso de textos o secuencias temporales.

Sin embargo, esta combinación hace posible una gran mejora en la generación de imágenes debido a la capacidad de los transformers para capturar dependencias a largo plazo en los datos. Esto es especialmente útil en el contexto de GANs, donde la generación de imágenes de alta calidad depende en gran medida de la capacidad de modelar relaciones complejas entre diferentes partes de la imagen.

Al combinar un generador basado en GAN con un modelo de transformer, se puede aprovechar la arquitectura de autoatención del transformer para mejorar la coherencia espacial de las imágenes generadas. El transformer puede ayudar a aprender representaciones más globales de la imagen, lo que permite al generador producir imágenes más realistas y detalladas. Esta sinergia entre ambas tecnologías no solo permite mejorar la calidad de las imágenes generadas, sino que también abre nuevas posibilidades para la creación de datasets sintéticos que se asemejan aún más a los datos reales. Esto es de gran importancia en aplicaciones donde la recolección de datos reales es costosa o limitada como puede ser en experimentos científicos o estudios de mercado.

Ahora vamos a ver cómo se implementan en las redes GAN:

Primero se define el **Transformer Encoder** el cual procesa secuencias de datos y se puede adaptar para procesar secuencias de características de imágenes que se define tal que así:

```
1 class TransformerEncoder(nn.Module):
2     def __init__(self, input_dim, num_heads, ff_dim, num_layers):
3         super(TransformerEncoder, self).__init__()
4         self.layers = nn.ModuleList([
5             nn.TransformerEncoderLayer(d_model=input_dim, nhead=num_heads,
6                 ↳ dim_feedforward=ff_dim)
7             for _ in range(num_layers)
8         ])
9         self.norm = nn.LayerNorm(input_dim)
10
11    def forward(self, x):
12        for layer in self.layers:
13            x = layer(x)
14        return self.norm(x)
```

Posteriormente se incorpora este transformer encoder en el generador para procesar un vector latente y generar características que se convertirán en una imagen esto se realiza tal que así:

```
1 class Generator(nn.Module):
2     def __init__(self, latent_dim, ngf, num_channels, num_heads, ff_dim,
3      ↳ num_layers):
4         super(Generator, self).__init__()
5         self.fc = nn.Linear(latent_dim, ngf * 8 * 4 * 4)
6         self.transformer = TransformerEncoder(ngf * 8, num_heads, ff_dim,
7             ↳ num_layers)
8         self.deconv = nn.Sequential(
9             nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size=4, stride=2,
10                 ↳ padding=1),
11             nn.BatchNorm2d(ngf * 4),
12             nn.ReLU(True),
13             nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size=4, stride=2,
14                 ↳ padding=1),
15             nn.BatchNorm2d(ngf * 2),
16             nn.ReLU(True),
17             nn.ConvTranspose2d(ngf * 2, ngf, kernel_size=4, stride=2, padding=1),
18             nn.BatchNorm2d(ngf),
19             nn.ReLU(True),
20             nn.ConvTranspose2d(ngf, num_channels, kernel_size=4, stride=2,
21                 ↳ padding=1),
22             nn.Tanh()
23         )
```

```

20     def forward(self, z):
21         x = self.fc(z).view(-1, 8 * 8, 8)
22         x = self.transformer(x)
23         x = x.view(-1, 8, 8, 8)
24         x = x.permute(0, 3, 1, 2)
25         return self.deconv(x)
26

```

El discriminador no varía con respecto a las redes GAN estándar, por lo tanto se mantiene el que se vio en un inicio previo a las técnicas de mejora.

Por último se definen los hiperparámetros que son diferentes con respecto a las redes GAN:

```

1 latent_dim = 100
2 ngf = 64
3 ndf = 64
4 num_channels = 3
5 num_heads = 8
6 ff_dim = 256
7 num_layers = 6
8
9 # Crear modelos
10 G = Generator(latent_dim, ngf, num_channels, num_heads, ff_dim, num_layers)
11 D = Discriminator(ndf, num_channels)
12
13 # Inicialización de optimizadores y pérdidas
14 optimizerG = optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
15 optimizerD = optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
16 criterion = nn.BCELoss()
17 #Aquí ya sigue el entrenamiento de la red como se explicó en el punto anterior
18

```

Por lo tanto, los transformer se implementan con el objetivo de mejorar la red generadora esto, tiene mucho sentido, ya que uno de los problemas usuales es el overfitting producido por una potencia mayor del discriminador sobre el generador y de ahí todo el conjunto de técnicas de mejora que se han visto. Lo innovador de este enfoque es, en vez de usar las técnicas de mejora para disminuir la potencia del discriminador, por qué no aumentamos la potencia del generador y de esto es precisamente de lo que se encargan los transformers.

Esta, no es la única manera de implementar la fusión de los transformers con las redes GAN ya que también se puede: implementar sobre el discriminador, utilizar redes GAN como StyleGAN o BigGAN en vez de la GAN usual, enfocar los transformers como una capa de fusión en modelos multi-modal, entre otros. Sin embargo, todos estos enfoques se anima a su estudio ya que implican avances sobre este tipo de fusión. Como se ha podido ya observar de primera instancia en esta sección las líneas futuras de investigación son enormes puesto a sus muchas aplicaciones y a su también amplio margen de mejora.

#### 6.4.3. Conclusiones

Para tomar conclusiones del desempeño de la fusión de las redes GAN y los transformers se va a usar la métrica FID.

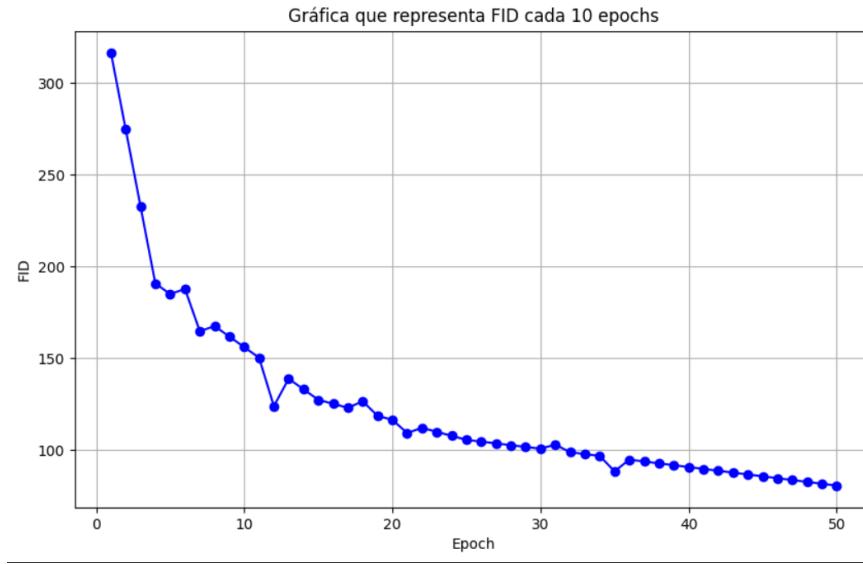


Figura 29: Gráfica del FID en el entrenamiento

Esta gráfica nos muestra que el FID de la red GAN al combinarla con un transformer es mas bajo lo cual indica que la fusión de estas dos conceptos es posible y no sólo es sino que es eficiente, haciendo que mejore. Sin embargo, como se puede observar todavía no produce un FID lo suficientemente bajo para poder asegurar que esta tecnología es ideal a la hora de crear datos sintéticos, haciendo así que todavía exista margen de mejora hasta llegar al punto ideal, que no exista diferencia alguna entre los datos reales y los datos sintéticos.

## 7. Conclusiones

A lo largo de este trabajo de fin de grado, se han implementado de manera detallada tanto las redes GAN como la combinación de las redes GAN y los transformers. Además se ha realizado una explicación teórica de los conceptos usados que encierran tanto a las redes GAN como a los transformers. Con respecto a la implementación, se ha hecho uso de Tensorflow, Keras y Pytorch. Ambos modelos se han ajustado en base a prueba y error, ya que para cada uno de ellos es necesario elegir los hiperparámetros específicos que favorecen el proceso de entrenamiento de las redes hasta obtener unos resultados correctos.

Además ha sido revelador el estudio ya que ha confirmado la suposición inicial, que la combinación de transformers y redes GAN produce como resultado una red generativa que se acerca de mas ágil al objetivo final: la producción de datos sintéticos. Esto se puede comprobar sobre todo gracias a la métrica implementada FID.

Todo este estudio deja como consecuencia un gran conjunto de aplicaciones que se van a desarrollar en el apartado de líneas futuras de investigación.

También se toma como conclusión del trabajo la importancia de hacer checkpoints a lo largo de la ejecución ya que una simple ejecución dura un gran conjunto de horas y por ello es necesario que se ejecute en diferentes días como ha sido en mi caso, ya que 500 epochs de la red producen este resultado. Hay que recalcar también el hecho de que no se haya producido un FID bajo no implica que el entrenamiento haya sido el incorrecto porque un dataset más apropiado para el entrenamiento o bien una tarjeta gráfica podrían haber arrojado resultados mejores de lo que lo han hecho.

## 8. Lineas futuras de investigación

Este trabajo de fin de grado permite seguir avanzando en el desarrollo de los siguientes modelos generativos de redes de neuronas.

- **Optimización de Hiperparámetros en la Combinación de GAN y Transformers**

Investigar cómo los diferentes hiperparámetros del Transformer (como el número de capas, cabezas de atención, dimensiones de embeddings, etc.) afectan el rendimiento de la GAN. AutoML para GANs y Transformers: Implementar técnicas de AutoML para la selección automática de los mejores hiperparámetros, lo que podría optimizar tanto la eficiencia como la eficacia del modelo.

- **Aplicación en Datos Multimodales**

Esto se logra a través de extender la investigación para generar imágenes a partir de descripciones textuales, utilizando Transformers para manejar la relación entre texto e imágenes.

- **Mejoras en la Estabilidad del Entrenamiento**

Explorar métodos adicionales de regularización para mejorar la estabilidad del entrenamiento en la combinación de GANs y Transformers, como la normalización de espectro, suavizado de etiquetas más avanzado, o dropout adaptativo.

- **Escalabilidad y Eficiencia Computacional**

Dado que los Transformers pueden ser costosos computacionalmente, una línea de investigación podría centrarse en hacer más eficiente la arquitectura, posiblemente mediante Transformers eficientes tales como Perfomer.

- **Mejora de la Calidad de las Imágenes Generadas**

Esto se puede lograr a través de la incorporación de modelos preentrenados, es decir, usando transformers preentrenados en tareas relacionadas para mejorar la calidad de las imágenes generadas, como utilizar un modelo preentrenado en visión para extraer características más detalladas. También se pueden investigar otras variantes de Transformers, como Vision Transformers (ViT) o Swin Transformers, pueden integrarse en la arquitectura de la GAN para mejorar la calidad de las imágenes.

Estos, son sólo algunos de los ejemplos de líneas futuras de investigación en las que se recomienda seguir contribuyendo a la mejora de la inteligencia artificial a través de los transformers.

## 9. Bibliografía

- 1.ERTEL, Wolfgang. Introduction to artificial intelligence. Springer, 2018.
- 2.XU, Yongjun, et al. Artificial intelligence: A powerful paradigm for scientific research. *The Innovation*, 2021, vol. 2, no 4.
- 3.KANG, Minguk, et al. Scaling up gans for text-to-image synthesis. En Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023. p. 10124-10134.
- 4.CHAKRABORTY, Tanujit, et al. Ten years of generative adversarial nets (GANs): a survey of the state-of-the-art. *Machine Learning: Science and Technology*, 2024, vol. 5, no 1, p. 011001.
- 5.KANG, Minguk; SHIN, Joonghyuk; PARK, Jaesik. StudioGAN: a taxonomy and benchmark of GANs for image synthesis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.
- 6.ROSENBLATT, Frank. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 1958, vol. 65, no 6, p. 386.
- 7.MANRIQUE GAMO, Daniel. From artificial cells to deep learning. An evolutionary story. 2021.
- 8.DU, Ke-Lin, et al. Perceptron: Learning, generalization, model selection, fault tolerance, and role in the deep learning era. *Mathematics*, 2022, vol. 10, no 24, p. 4730.
- 9.KARRAS, Tero; LAINE, Samuli; AILA, Timo. A style-based generator architecture for generative adversarial networks. En Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019. p. 4401-4410.
- 10.NOWOZIN, Sebastian; CSEKE, Botond; TOMIOKA, Ryota. f-gan: Training generative neural samplers using variational divergence minimization. *Advances in neural information processing systems*, 2016, vol. 29.
- 11.MIYATO, Takeru, et al. Spectral normalization for generative adversarial networks. arXiv preprint arXiv:1802.05957, 2018.
- 12.WANG, Zhengwei; SHE, Qi; WARD, Tomas E. Generative adversarial networks in computer vision: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, 2021, vol. 54, no 2, p. 1-38.
- 13.ISLAM, Saidul, et al. A comprehensive survey on applications of transformers for deep learning tasks. *Expert Systems with Applications*, 2023, p. 122666.
- 14.TURNER, Richard E. An introduction to transformers. arXiv preprint arXiv:2304.10557, 2023.
- 15.VASWANI, Ashish, et al. Attention is all you need. *Advances in neural information processing systems*, 2017, vol. 30.
- 16.RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning representations by back-propagating errors. *nature*, 1986, vol. 323, no 6088, p. 533-536.