

# **Grado en Ingeniería Informática**

# **Informatikako Ingeniaritzako gradua**

**Proyecto fin de grado**

**Gradu amaierako proiekta**

## Grado en Ingeniería Informática Informatikako Ingeniaritzako gradua

Proyecto fin de grado  
Gradu amaierako proiekta

GOMEZ  
LARRAKOET  
XEA NEREA  
- 45751435L

Firmado  
digitalmente por  
GOMEZ  
LARRAKOETXEA  
NEREA - 45751435L  
Fecha: 2024.06.20  
08:37:57 +01'00'

## Resumen

Este proyecto se ha enfocado en abordar el desafío de la falta de datos en ciertas industrias o sectores, lo cual impide la generación de modelos de Inteligencia Artificial robustos. Se busca proporcionar apoyo a estos casos mediante la generación de imágenes nuevas, con el objetivo de enriquecer los conjuntos de datos y mejorar la robustez de los modelos de IA. El enfoque adoptado se basa en el entrenamiento adversario, destacando el uso de Redes Generativas Antagónicas (GANs) para la generación de ejemplos adversariales.

El proyecto ha seguido una metodología estructurada que incluye el estudio de la teoría subyacente, la planificación y desarrollo de experimentos, y la evaluación de la efectividad del entrenamiento adversario en diversos modelos de IA. Se han empleado tecnologías avanzadas, incluyendo Python para la programación, y frameworks de aprendizaje profundo como PyTorch y TensorFlow. Además, se han explorado una variedad de arquitecturas de GANs, como Simple GAN, DCGANs, ProGANs y StyleGANs, para la generación de un amplio espectro de ejemplos adversariales.

El objetivo del proyecto es la implementación y validación de este enfoque en modelos específicos de IA, demostrando que la integración de ejemplos adversariales en los conjuntos de datos no solo incrementa la robustez de los modelos frente a ataques similares sino que también mejora su capacidad de generalización. El objetivo principal del proyecto es determinar si el entrenamiento adversario, respaldado por la generación de ejemplos mediante GANs, constituye una estrategia eficaz para el desarrollo de sistemas de IA más seguros, confiables y precisos.

## Descriptores

Redes Generativas Antagónicas (GANs), Inteligencia Artificial (IA), Deep Learning, Generación de imágenes, Modelos de IA, Entrenamiento Adversario.

## Índice

<b>1. Introducción.....</b>	<b>1</b>
<b>2. Antecedentes y Objetivos.....</b>	<b>1</b>
2.1 Antecedentes.....	1
2.2 Objetivo General.....	2
2.3 Objetivos Específicos.....	2
<b>3. Fundamentos Teóricos.....</b>	<b>3</b>
3.1 Introducción a la Inteligencia Artificial.....	3
3.2 Deep Learning.....	4
<b>4. Fundamentos Redes Generativas adversariales.....</b>	<b>6</b>
4.1 Introducción.....	6
4.1.1 Modelo Generativo.....	7
4.1.2 Modelo Discriminativo.....	8
4.1.3 Generador vs Discriminador.....	9
4.1.4 función de pérdida.....	11
4.2 funciones de activación.....	13
4.2.1 ReLU (Rectified Linear Unit).....	14
4.2.2 Leaky ReLU.....	14
4.2.3 Tangente Hiperbolica(TanH).....	15
4.2.4 Sigmo ID.....	16
4.3 Convoluciones.....	17
4.3.1 Capas Convolucionales.....	17
4.3.2 Strides y Padding.....	19
4.3.3 Capas Convolucionales Transpuestas.....	20
4.4 Layers/Capas.....	22
4.4.1 Batch Normalization.....	22
4.4.2 Dense Layer.....	23
4.4.3 Flatten Layer.....	26
4.5 Variables Fundamentales.....	27
4.5.1 Gradientes.....	27
4.5.2 Learning Rate/ Tasa de Aprendizaje.....	27
4.5.3 Optimizador ADAM.....	28
<b>4. Tipos de redes Generativas Adversariales.....</b>	<b>30</b>
5.1 Deep Convolutional Gans.....	30
5.2 Wasserstein Gradient Penalty Gans.....	30
5.2.1 función de pérdida.....	30

5.2.2 Gradient Penalty.....	31
5.3 Progressive Gans.....	35
5.3.1 Introduccion.....	35
5.3.2 Crecimiento Progresivo.....	35
5.3.3 Transicion.....	36
5.3.4 MiniBatchSTD.....	37
5.3.5 PixelWise Normalization.....	38
5.3.6 EQLR / Tasa de Aprendizaje Ecualizada.....	39
<b>6.Entornos.....</b>	<b>39</b>
6.1 Pytorch.....	39
6.2 Cuda.....	40
6.3 Vscode /GoogleColab.....	40
6.4 Keras +Tensorflow.....	41
6.5 TensorBoard.....	41
<b>7.Implementación Técnica de las Gans.....</b>	<b>42</b>
7.1 Implementación Técnica DcGan.....	43
7.1.1 Dataset.....	43
7.1.2 Estructura General.....	44
7.1.3 Generador.....	45
7.1.4 Discriminador.....	47
7.1.5 hiperparámetros e Imports.....	48
7.1.6 Preparacion de datos y Definicion de Componentes.....	51
7.1.7 Entrenamiento.....	52
7.1.8 Ejecucion.....	56
8.2 Implementación Técnica WGAN-GP.....	59
8.2.1 Dataset.....	60
8.2.2 Entorno.....	60
8.2.3 hiperparámetros.....	62
8.2.4 Preprocesamiento de Datos.....	63
8.2.5 Discriminador/crítico.....	65
8.2.6 Generador.....	68
8.2.7 Definiendo el Entrenamiento(Clase WGAN-GP).....	71
8.2.8 Instanciamos y Compilamos.....	79
8.2.9 CallBacks y Checkpoint.....	80
8.2.10 Ejecutamos el Entrenamiento.....	82
8.2.11 Analisis Resultados.....	83
9.3 Implementación Técnica ProGan.....	85
9.3.1 Dataset.....	85
9.3.2 Entorno.....	85

9.3.3 Estructura Del Código.....	85
9.3.4 Configuracion Inicial e hiperparámetros.....	86
9.3.5 Arquitectura (Model.py).....	87
9.3.6 Utils.py.....	100
9.3.7 Train.py(Entrenamiento).....	102
9.3.8 Analisis Resultados.....	114
<b>10. Mejora De modelos.....</b>	<b>115</b>
10.1 Entorno y Herramientas.....	115
10.1.1EfficientNet.....	115
10.1.2 Regularizacion L1/L2.....	116
10.2 Mejora de Clasificador De tumores Cerebrales.....	117
10.2.1 CNN.....	117
10.2.2 Preparación de Datos.....	117
10.2.3 Clasificador de Cerebros.....	119
10.2.4 Resultados y Conclusiones.....	126
<b>11. Conclusiones y Trabajo Futuro.....</b>	<b>132</b>
<b>12. Principios Eticos.....</b>	<b>133</b>
<b>12. ¿Es la Tecnología Neutra?.....</b>	<b>134</b>
<b>13. Plan de trabajo y presupuesto.....</b>	<b>136</b>
13.1 Plan de Trabajo.....	136
13.2 Presupuesto.....	138
13.2.1 Presupuesto Personal.....	138
13.2.2 Presupuesto Caso de Uso Real.....	138
<b>Bibliografia.....</b>	<b>140</b>

## Índice de Figuras

Figura 1: Esquema de una red neuronal profunda[1].....	5
Figura 2: Modelo Generativo a partir de vector ruido[16].....	7
Figura 3: Ruido aleatorio.....	7
Figura 4: Modelo Discriminativo[16].....	9
Figura 5: ReLU.....	14
Figura 6: LeakyReLU.....	15
Figura 7: Tangente Hiperbolica.....	16
Figura 8: Sigmoid.....	16
Figura 9 : Ejemplo práctico de aplicación de filtro.....	18
Figura 10: Ejemplo de patrones de Filtros.....	19
Figura 11: Ejemplo Discriminador [19].....	21
Figura 12: Proceso de Generador con Convoluciones Transpuestas.....	22

Figura 13: Funcionamiento Batch Normalization.....	24
Figura 14: Capa Densa.....	25
Figura 15: Capa Flatten.....	27
Figura 16: Constante Lipschitz.....	32
Figura 17: Ajuste de pesos y penalización de gradientes.....	33
Figura 18: Gráfica de funcionamiento de muestras interpoladas.....	34
Figura 19: Ejemplo Red Generativa Adversarial Wasserstein.....	35
Figura 20: Imágenes generadas por ProGan.....	36
Figura 21: Generador y Discriminador en ProGan.....	36
Figura 22: Transicion.....	37
Figura 23: Estabilización.....	38
Figura 24: Esquema de Generador y Discriminador ProGan.....	38
Figura 25: Version Cuda.....	41
Figura 26: Ejemplo Visualizacion Tensorboard.....	43
Figura 27: Mnist Dataset.....	44
Figura 28: Estructura General Codigo DcGan.....	45
Figura 29: Import Generador DcGan.....	46
Figura 30: Código Generador DcGan.....	47
Figura 31: Código Discriminador DcGan.....	49
Figura 32: Import De módulos de Torch DcGan.....	50
Figura 33: Valores de HiperParametros DcGan.....	51
Figura 34: Preparación de datos y definición de componentes codigo DcGan.....	52
Figura 35:Código de entrenamiento DcGan.....	53
Figura 36: Actualización del Discriminador codigo DcGan.....	54
Figura 37: Actualización Generador código DcGan.....	56
Figura 38: Código para el registro de las estadísticas DcGan.....	57
Figura 39:Código de logueo de imágenes en Tensorboard para DcGan.....	57
Figura 40: Progreso en cada iteración DcGan.....	58
Figura 41: Tensorboard –logdir runs.....	58
Figura 42: Ejecucion servidor Tensorboard en localhost.....	59
Figura 43: Evolución funciones de pérdida implementación técnica DcGan.....	59
Figura 44: Resultados en tensor board de la implementación Técnica DcGan.....	59
Figura 45: Comparación ejemplos Mnist generados vs reales.....	60
Figura 46: Gráfica evolución Función de Pérdida del Discriminador como del Generador, en la implementación técnica DcGan.....	60
Figura 47: Métricas Durante el entrenamiento de la DcGAN.....	61
Figuras 48: Dataset Cerebros.....	62
Figura 49: Estructura simple Dataset cerebros.....	62
Figura 50: Gpu para el entrenamiento WGAN-GP.....	62
Figura 51: Unzip Brains.zip.....	63
Figura 52: Estructura del Dataset Brains en Collab.....	63

Figura 53: Imports necesarios para el código de la implementación WGAN-GP.....	63
Figura 54: Hiperparametros implementación Wgan-GP.....	64
Figura 55: Preprocesamiento de Datos.....	65
Figura 56: Creación del dataset a partir del Path.....	66
Figura 57: Cargar datos para el entrenamiento de el WGan-GP.....	66
Figura 58: Mostrar ejemplos de Cerebros del Dataset.....	67
Figura 59: Ejemplos de cerebros del Dataset y sus propiedades.....	67
Figura 60: Código del Discriminador/Critico WGAN-GP.....	68
Figura 61: Resumen del modelo Discriminativo de la Implementación WganGp.....	69
Figura 62: Codigo del generador de Implementación técnica WGAN-GP.....	69
Figura 63: Input del Generador Wgan-Gp y capas Iniciales.....	70
Figura 64: Fragmento de Codigo Generador Wgan-Gp.....	70
Figura 65: Capa se salida y definicion de el Modelo Generador WGAN-GP.....	71
Figura 66: Estructura del Generador WGAN-GP.....	71
Figura 67: Constructor clase WGAN-GP.....	72
Figura 68: Metodo Compile WGAN-GP.....	72
Figura 69: Propiedad Metrics.....	73
Figura 70: Funcion penalización del gradiente.....	74
Figura 71: Código de interpolación de muestras.....	74
Figura 72: Cálculo de la penalidad del gradiente y de los gradientes,.....	75
Figura 73: Train_step Wgan-GP.....	76
Para entenderlo mejor paso a paso, vayamos por partes.....	77
Figura 74: Fragmento 1 de código Train_step WGAN-GP.....	77
Figura 75: Fragmento 2 de código Train_step WGAN-GP.....	78
Figura 76: Fragmento 3 de Train_step WGAN-GP.....	79
Figura 77: Fragmento final Train_step Wgan-GP.....	79
Figura 78: Instanciamos modelo WGAN-GP.....	80
Figura 79: Compilamos WGan-GP.....	80
Figura 80: CallBacks y Checkpoints Implementación Tecnica WGan-Gp.....	81
Figura 81: Función OnEpochEnd WGan-Gp.....	82
Figura 82: Checkpoint Wgan-Gp.....	82
Figura 83: Wgan-GP.fit.....	83
Figura 84: Progreso entrenamiento WGAN-GP.....	83
Figura 85: Últimos Epochs implementación WGAN-GP.....	84
Figura 86: Resultados y comparativa cerebros generados vs reales.....	85
Figura 87: Datos de CelebaHQ.....	86
Figura 88: Estructura código Pro-GAN.....	86
Figura 89: Configuración inicial e hiperparametros.....	87
Figura 90: Código clase WSConv2d.....	88
Figura 91: Metodo Forward WSConv2d.....	89

Figura 92: Normalización de Pixeles código.....	89
Figura 93: Bloque Convolucional código.....	90
Figura 94: Clase Generador implementación Pro-GAN.....	91
Figura 95: Bloque Inicial Generador Pro-GAN.....	92
Figura 96: Aplicación Capa Rgb código.....	92
Figura 97: Bloques Progresivos.....	92
Figura 98: Formula fade_in en el código Pro-GAN.....	93
Figura 99: Función forward del generador en PRO-Gan.....	93
Figura 100: Esquema General del funcionamiento del código del Generador en Pro-GAN... 95	
Figura 101: Código del Discriminador en implementación Pro-GAN.....	95
Figura 102: Constructor Discriminador Pro-GAN.....	96
Figura 103: Cálculo de canales de entrada y salida código Pro-GAN para el discriminador... 96	
Figura 104: Definición de capa inicial RGB y el average pooling en el discriminador.....	97
Figura 105: Instancia del proceso del bloque final para el Discriminador.....	97
Figura 106: Formula fade_in Discriminador.....	97
Figura 107: Código desviación estándar mini batch Pro-GAN.....	98
Figura 108: Método Forward o proceso del Discriminador para implementacion Pro-GAN. 98	
Figura 109: Esquema General código discriminador Pro-GAN.....	99
Figura 110: Test de modelos Pro-GAN.....	99
Figura 111: Resultado Test de los modelos de implementacion Pro-GAN.....	100
Figura 112: Import de librerías en Utils.py.....	100
Figura 113:Funcion para mostrar las gráficas en Tensorboard.....	100
Figura 114:Penalidad del Gradiente código para implementacion Pro-GAN.....	101
Figura 115: Métodos de guarda y carga del checkpoint.....	102
Figura 116: Método para generar las imágenes y almacenarlas en en el path.....	102
Figura 120: Imports para Train.Py implementacion Pro-Gan.....	103
Figura 121: Preprocesamiento y carga de los datos, implementacion Pro-GAN.....	104
Figura 122: Funcionamiento Train Fn.....	105
Figura 123:Fragmento de Código comienzo ciclo entrenamiento.....	106
Figura 124: Código función de perdida del critico Pro-GAN.....	107
Figura 125: Optimización, retropropagación, y actualización del critico.....	107
Figura 126: Optimización y función de perdida del generador, código Pro-GAN.....	108
Figura 127: Cálculo de alpha, código Pro-GAN.....	108
Figura 128. Llamada función plot_to_tensorboard.....	108
Figura 129: Funcion Main implementación Pro-GAN.....	109
Figura 130: Fragmento 1 de código del Main.....	110
Figura 131: Fragmento 2 de código del Main.....	111
Figura 132: Resultados implementación Pro-GAN.....	112

Figura 133: Estructura Dataset Mezclado.....	116
Figura 134: Unzip mix Dataset.....	116
Figura 135: Imports CNN cerebros.....	117
Figura 136: Definición de labels y directorios CNN cerebros.....	118
Figura 137: Actualización de estructura de datos.....	119
Figura 138: Fragmento 1 código preprocesamiento de datos, división test y train.....	119
Figura 139 : Fragmento 2 de código preprocesamiento de datos CNN.....	120
Figura 140: Código plot para visualizar los datos del CNN.....	121
Figura 141: Muestra de Datos del Dataset.....	121
Figura 142: Estructura del Modelo CNN basado en EfficientNet.....	122
Figura 143: Procesos modelo secuencial CNN.....	122
Figura 144: Compilación de modelo CNN, y summary(resumen).....	123
Figura 145: Resumen Estructura modelo CNN Cerebros.....	123
Figura 146: Ejecucion entrenamiento clasificador CNN de cerebros enfermos.....	124
Figura 147: Código de las gráficas para visualización de métricas del entrenamiento CNN..	125
Figura 148: Código para la muestra de las métricas finales.....	126
Figura 149: Comparación de gráficas de pérdida, datos Reales vs datos R+Generados...	126
Figura 150: Comparación de gráficas de precisión, datos Reales vs datos R+Generados	127
Figura 151: Comparación matrices de confusión.....	128
Figura 152: Métricas finales entrenamiento con datos reales.....	129
Figura 153: Métricas finales entrenamiento con datos reales + generados.....	129
Figura 154: Diagrama de Gantt.....	136

## 1. INTRODUCCION

---

El presente Proyecto Fin de Grado se centra en abordar la problemática de la falta de datos en diversas industrias y sectores, lo cual limita la capacidad de desarrollar modelos robustos de Inteligencia Artificial (IA). Para solucionar esta limitación, hemos implementado Redes Generativas Antagónicas (GANs), una técnica avanzada de aprendizaje profundo que permite la generación de datos sintéticos. Estas GANs no solo ayudan a ampliar los conjuntos de datos disponibles, sino que también mejoran la robustez y precisión de los modelos de IA al proporcionar ejemplos adversariales para su entrenamiento. A lo largo de este proyecto, se describen las metodologías empleadas, las arquitecturas de GANs implementadas, y los resultados obtenidos, demostrando la efectividad de este enfoque para generar datos de alta calidad y optimizar modelos de IA en contextos con datos limitados.

## 2. ANTECEDENTES Y OBJETIVOS

---

### 2.1 ANTECEDENTES

El uso de Redes Generativas Antagónicas (GANs) ha revolucionado el campo del aprendizaje profundo, permitiendo la generación de datos sintéticos realistas que pueden utilizarse para enriquecer conjuntos de datos y mejorar la robustez de los modelos de inteligencia artificial (IA). Las GANs consisten en dos modelos que compiten entre sí: un generador que crea datos falsos y un discriminador que intenta distinguir entre datos reales y falsos. Esta dinámica de competencia impulsa a ambos modelos a mejorar continuamente.

En particular, las variantes de GANs como Deep Convolutional GANs (DCGANs) y Wasserstein GANs (WGAN-GP) han mostrado ser efectivas en la generación de imágenes de alta calidad, lo que las hace especialmente útiles en áreas donde la disponibilidad de datos es limitada. Estas redes no solo generan datos adicionales para entrenar otros modelos de IA, sino que también ayudan a mejorar la estabilidad y la eficiencia del proceso de entrenamiento.

El proyecto se enfoca en el uso de GANs para abordar la falta de datos en ciertas industrias, proporcionando ejemplos adversariales generados para aumentar la diversidad del conjunto de datos y mejorar la capacidad de generalización de los modelos de IA. Este enfoque no solo incrementa la robustez de los modelos ante ataques adversariales, sino que también mejora su precisión en tareas de clasificación.

## 2.2 OBJETIVO GENERAL

El objetivo general de este proyecto de fin de grado es investigar y demostrar la efectividad del uso de Redes Generativas Antagónicas (GANs) en la generación de datos sintéticos para mejorar la robustez y precisión de modelos de inteligencia artificial (IA) en contextos donde la disponibilidad de datos es limitada. Esto se logrará a través de la implementación, integración y evaluación de diversas arquitecturas de GANs, así como mediante la aplicación de técnicas de mejora en clasificadores existentes.

## 2.3 OBJETIVOS ESPECÍFICOS

1. Comprensión de la arquitectura de las redes generativas neuronales
2. Implementación de GANs:
  - Desarrollar y entrenar diferentes arquitecturas de GANs, incluyendo Deep Convolutional GANs (DCGANs) y Wasserstein GANs con penalización de gradiente (WGAN-GP).
3. Integración y Evaluación:
  - Integrar los datos sintéticos generados por las GANs en el entrenamiento de modelos clasificadores, como redes neuronales convolucionales (CNN).
  - Evaluar el impacto de los datos generados en términos de precisión y capacidad de generalización de los modelos de IA.
4. Mejora de Modelos Existentes:
  - Evaluar el impacto de los datos generados en términos de precisión y capacidad de generalización de los modelos de IA.
5. Documentación y Divulgación:
  - Documentar detalladamente el proceso de desarrollo, implementación y evaluación de las GANs.
  - Proporcionar un marco teórico y práctico que pueda ser utilizado por la comunidad científica y tecnológica para futuras investigaciones y desarrollos en el campo del aprendizaje profundo y la inteligencia artificial aplicada.

## 3.FUNDAMENTOS TEORICOS

---

### 3.1 INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

Ultimamente la inteligencia artificial(IA) esta en boca de todos, y no es casualidad ya que la inteligencia artificial se ha convertido en una de las disciplina más innovadora y transformadoras de la tecnología moderno cuyo objetivo principal es replicar y ampliar las capacidades cognitivas humanas a través de sistemas computacionales capaces de aprender, razonar y adaptarse. Desde sus raíces en los años 50, con pioneros como Alan Turing y John McCarthy, la IA ha evolucionado enormemente, impulsada por mejoras en la computación y la disponibilidad de grandes volúmenes de datos. La inteligencia artificial se divide principalmente en dos grandes subdisciplinas, machine learning y deep learning.

Machine learning permite a las máquinas aprender de los datos sin ser programadas explícitamente para cada tarea es decir en lugar de seguir instrucciones paso a paso, estas máquinas analizan datos y encuentran patrones, realizando predicciones. Estos modelos pueden trabajar con datos estructurados como no estructurados(imágenes,Videos,Texto...).Esta subdisciplina lleva años en marcha sin embargo otra subdisciplina de la IA está en pleno auge, llevando a cabo una revolución de resultados impresionante, se trata de el Deep Learning.

El Deep Learning utiliza redes neuronales profundas para aprender representaciones complejas de datos no estructurados, resultando ser muy efectivo en generación de imágenes, modelos de procesamiento de lenguaje(Gpt, Gemini etc...). En este proyecto de GANS generamos imágenes y las clasificaremos, por ello nos centraremos en esta disciplina que es el deep learning.

En conclusión, la Inteligencia Artificial, impulsada por machine learning y deep learning, está transformando la manera en que interactuamos con la tecnología y el mundo que nos rodea. Sus aplicaciones son vastas y diversas, abarcando múltiples industrias y mejorando la eficiencia y la efectividad en una amplia gama de actividades humanas. La IA no solo está redefiniendo el presente, sino que también está forjando el futuro. Con cada avance, nos acercamos más a un mundo donde las máquinas no solo ejecutan tareas, sino que también entienden, aprenden y evolucionan. Esta capacidad de adaptación y mejora continua es lo que convierte a la IA en una herramienta tan poderosa y transformadora, y es nuestra responsabilidad asegurar que se utilice de manera que beneficie a toda la humanidad. A continuación comprenderemos los fundamentos teóricos estrictamente necesarios para poder comprender los objetivos, funcionamiento desarrollo, y código de este proyecto.

### **3.2DEEP LEARNING**

Antes de profundizar en la redes generativas adversariales primero hemos de comprender algunos conceptos y conocimientos clave de el aprendizaje profundo (deep learning) y de los modelos generativos y discriminativos

Las GANS son un tipo de aprendizaje profundo, es por esto que antes de entrar en detalle en las gans tenemos que comprender cómo funcionan los algoritmos de deep learning.La definicion solida de el deep learning es “una clase de algoritmos de machine learning que utilizan muchas capas acopladas de unidades de procesamiento para aprender representaciones de alto nivel de datos no estructurados.” [1]Vamos a desglosar esta definición poco a poco para entenderla mejor.

El deep learning está diseñado para el aprendizaje a través de datos no estructurados, los datos no estructurados como su nombre indica no están agrupados y ordenados en ninguna tabla, estos datos pueden ser imágenes, texto o audio.

Los tipos de datos no estructurados son mucho más difíciles de procesar ya que la información de estos datos no es muy informativa, es decir por ejemplo en una imagen o en un audio cogemos un fragmento de este, como puede ser un pixel o 2 segundos de audio o una simple letra de un texto, no nos da tanta información como aquellos datos que si están estructurados.A pesar de esto, los modelos generativos basados en deep learning como pueden ser GANS se especializan en el manejo y generación características de alto nivel (objetos, caras, frases ...)

Los sistemas de aprendizaje profundo incluyendo GANS se componen redes neuronales, una red neuronal es una serie de distintas capas amontonados formadas por unidades de procesamiento (neuronas). Es importante recalcar que las capas estan conectadas entre si mediante las neuronas a través de distintos pesos formando una red profunda(deep network). Hay muchos tipos de diferentes capas, cada tipo de capa ofrece funciones distintas, y la mezcla de estas darán lugar a varios algoritmos distintos. Entraremos más en detalle sobre las capas cuando nos adentramos en los algoritmos de generación de ejemplos adversariales.

En el siguiente esquema podemos apreciar mejor esto:

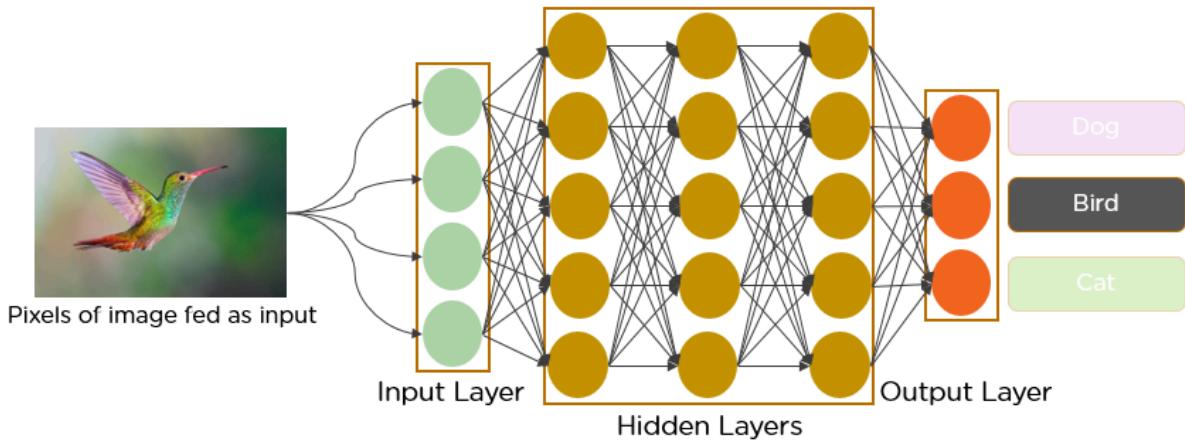


Figura 1: Esquema de una red neuronal profunda[1]

Cada círculo representa una unidad de procesamiento en las capas (layers). Toda deep neural network generativa tiene una capa de input (entrada) y otra de output (salida), en la capa de input se pasan los datos de entrada, las hidden layers se le llaman a todas las capas que se encuentran entre el input y el output, en estas capas se hace todo el procesamiento de los pixeles para lograr un output.

El objetivo o la funcionalidad principal de las deep networks reside en las hidden layers, estas capas están conectadas por un set de pesos, el objetivo es encontrar el set, suma o conjunto de pesos más adecuados que resulten en el output mas preciso, a esto se le llama entrenamiento de el algoritmo (training).

El training se alimenta del *backpropagation* (retropropagacion), función que veremos habitualmente en los códigos de este proyecto, backpropagation se encarga de retroceder el error de predicción por las hidden layers otra vez para corregir el error ajustando los pesos para hacer el algoritmo y el output más precisos.

Las redes neuronales son muy útiles ya que debido a la autosuficiencia del entrenamiento gracias a las capas, backpropagation y al aprendizaje continuo, estos modelos de deep learning son muy hábiles a la hora de detectar características complejas en los datos (como pueden ser objetos, sonrisas etc..), el modelo es autosuficiente ya que ajusta los pesos por cuenta propia con el objetivo final de minimizar el error en sus predicciones.

Normalmente las primeras hidden layers tratan las características simples (low level features), como puede ser la intensidad de color de los pixeles en una zona o el sombreado etc, una vez estas características son tratadas por cada capa el algoritmo va aprendiendo hasta en las ultimas capas ser capaz de distinguir características de alto nivel como caras objetos etc. Las

redes neuronales profundas tienden a tener varias capas ya que cuanto más profundas más aumenta la precisión del algoritmo aunque el tiempo de ejecución será mayor.

Antes de hablar sobre los distintos tipos de layers, y de funciones, vamos a entender antes los fundamentos teóricos de las GANS (Redes generativas adversarias o antagónicas) [1]

## 4.FUNDAMENTOS REDES GENERATIVAS ADVERSARIALES

---

*Basado en [1], [3], [4],[6], [5],[7].*

### 4.1 INTRODUCCIÓN

GANS son las siglas de generative adversarial network (redes generativas antagónicas/adversarias), el nombre de estas ya nos da una gran pista de como funcionan. Como mencionado, estas redes son profundas por lo que funcionan con distintas capas y unidades de procesamiento, además son generativas (generan datos a partir de un training dataset ), esto indica que utiliza un modelo generativo, además de esto las GANS utilizan un modelo discriminatorio que competirá directamente como adversario de el modelo generador.

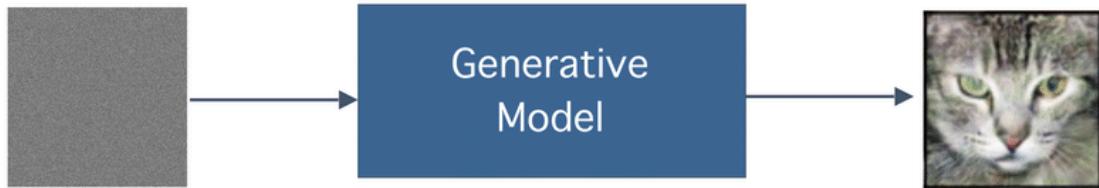
### 4.2 ARQUITECTURA

#### 4.1.1 Modelo Generativo

El modelo generativo a partir de un dataset de datos no estructurados reales, entrena(training) con estos datos con el fin final de generar nuevo datos indistinguibles del dataset general. Por ejemplo, queremos un modelo ia para generar imágenes de animales, este modelo de ia será entrenado con un dataset real de animales para que genere imágenes de IA totalmente nuevas de animales.

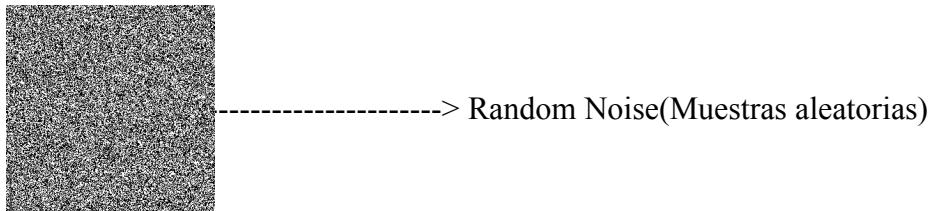
Un modelo generativo se diseña para aprender y entrenarse a partir de un conjunto de datos no estructurados y reales. Su objetivo principal es generar nuevos datos que sean prácticamente indistinguibles del conjunto de datos original. Tomando como ejemplo la creación de un modelo de ia destinado a generar imágenes de animales, dicho modelo se

entrenaría (training) con un conjunto de datos real compuesto por imágenes completamente nuevas y ÚNICAS de animales (añadir esquema).



*Figura 2: Modelo Generativo a partir de vector ruido[16]*

El aprendizaje a partir de datos bien estructurados representa un reto significativo. Sin embargo, gracias a la tecnología de las redes neuronales profundas explicadas anteriormente, es posible alcanzar este objetivo. En las fases iniciales, el modelo generativo comienza generando “random noise”(muestras aleatorias). En el contexto de la generación de imágenes, esto equivale a la creación de píxeles de manera aleatoria.



*Figura 3: Ruido aleatorio*

En este proyecto nos referiremos a él random noise mediante la variable z

A partir de este ruido, el modelo generativo se propone generar datos que emulan los patrones y características de los datos reales de tal forma que los datos sintéticos sean indistinguibles de los auténticos. Esto no solo definirá la calidad y autenticidad de los datos generados sino que también demuestra el potencial de las redes neuronales profundas para interpretar y replicar la complejidad de los conjuntos de datos reales no estructurados. Esto último hace que los modelos generativos se alineen perfectamente con el objetivo de este proyecto donde queremos que las imágenes generadas sean totalmente indistinguibles de los datos utilizados para entrenar.

Por último, es importante destacar que el modelo generativo no opera de manera determinista. En lugar de ello, se fundamenta en principios de distribución probabilística. Esto significa que el proceso no se adhiere a un método fijo para calcular los resultados generados. Por el contrario, siempre se producen salidas diversas, cada una con el objetivo de replicar la distribución de los datos reales. Así, el modelo busca generar nuevos datos que reflejen

fielmente las características y patrones originales. Este enfoque probabilístico subraya la capacidad del modelo generativo para crear variaciones ricas y complejas, demuestra su eficacia y versatilidad en la generación de datos.

#### 4.1.2 Modelo Discriminativo

Los modelos discriminativos constituyen una clase fundamental dentro de los algoritmos de aprendizaje automático, especialmente en tareas de clasificación y regresión. A diferencia de los modelos generativos, que intentan modelar cómo se generan los datos aprendiendo la distribución conjunta de las características de entrada y las etiquetas de salida, los modelos discriminativos se centran directamente en modelar la frontera de decisión entre las clases. Esto se logra aprendiendo la distribución condicional de la etiqueta de salida dado un vector de entrada.

El núcleo de un modelo discriminativo es su capacidad para diferenciar entre diferentes clases o categorías basándose en las características observadas en los datos de entrenamiento. Para ello, el modelo examina los ejemplos de entrenamiento - datos ya etiquetados - y aprende las características que distinguen a cada clase. Utilizando este conocimiento, puede predecir la clase o el valor de nuevas instancias no vistas anteriormente.

Por ejemplo, en una tarea de clasificación de imágenes, donde el objetivo es identificar si una imagen es de un gato o no, un modelo discriminativo aprenderá los atributos visuales clave que caracterizan a los gatos (como las orejas puntiagudas, los bigotes, etc.) a partir de un conjunto de imágenes etiquetadas. Cuando se le presenta una nueva imagen, el modelo evalúa la presencia y la fuerza de estas características para estimar la probabilidad de que la imagen represente a un gato.

Una de las fortalezas de los modelos discriminativos es su capacidad para cuantificar la incertidumbre en sus predicciones a través de probabilidades. Siguiendo el ejemplo anterior, si al modelo se le proporciona una imagen de un gato, no solo predice que la imagen pertenece a la categoría "gato", sino que también puede ofrecer una probabilidad asociada a esa predicción, como un 70%. Esto indica que, según las características observadas en la imagen y lo que ha aprendido durante el entrenamiento, hay un 70% de posibilidades de que la imagen sea de un gato.



Figura 4: Modelo Discriminativo[16]

Los modelos discriminativos son particularmente útiles en situaciones donde el objetivo es realizar predicciones precisas y es posible disponer de grandes cantidades de datos etiquetados para el entrenamiento.

En resumen los modelos discriminativos estiman  $p(y|x)$ , representando la probabilidad de que Y(output) tome un cierto valor dado un conjunto específico de datos de entrada X

#### 4.1.3 Generador vs Discriminador

El modelo generador generara imágenes tratando de engañar a el discriminador, y el discriminador clasificara si la imagen es real o falsa, a medida que compiten varias veces entre ellos van aprendiendo uno de otro gracias a la profundidad de las redes neuronales, el generador cada vez aprende mejor a engañar al discriminador, y el discriminador trata de constantemente aprender a clasificar las imágenes como reales o falsas al mismo tiempo. Mejorando y aprendiendo ambos modelos continuamente

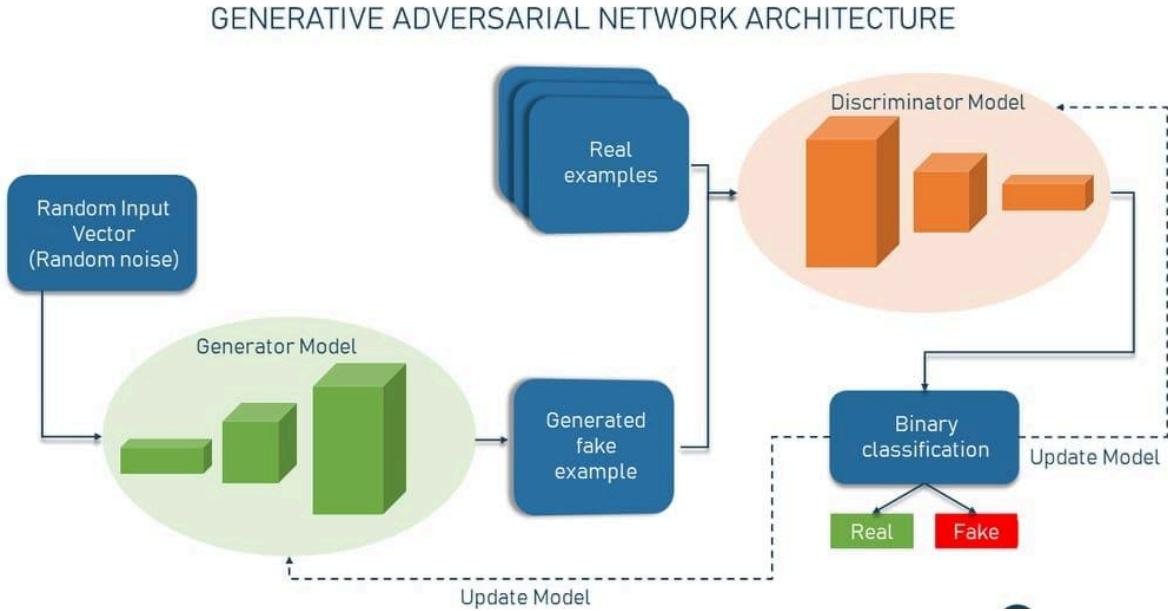


Figura 5: Arquitectura de red generativa adversarial [17]

Consideremos un ejemplo clásico para ilustrar este concepto: Imagine un experto en arte (Discriminador) cuya tarea es diferenciar entre obras de arte auténticas y falsificaciones. Por otro lado, tenemos a un falsificador (Generador) cuyo objetivo es engañar al experto, haciendo que este no pueda distinguir entre sus falsificaciones y las obras originales.

Inicialmente, el experto en arte identifica sin esfuerzo las falsificaciones, pero el falsificador, al percatarse de esto, comienza a perfeccionar sus técnicas para hacer sus imitaciones cada vez más convincentes. Aunque el experto encuentra cada vez más difícil diferenciarlas, esto a su vez le impulsa a mejorar su capacidad de discernimiento. Así, tanto el falsificador como el experto en arte se encuentran en una constante evolución de sus habilidades, mejorando de forma continua.

Esto nos lleva a preguntarnos, "¿Logrará alguna vez el generador engañar completamente al discriminador?" el discriminador, gracias a su entrenamiento constante, siempre busca nuevas maneras de identificar las sutilezas que delatan una falsificación pero las habilidades del generador pueden llegar a ser tan refinadas que sus creaciones sean prácticamente indistinguibles de las originales **para un observador promedio**. Los modelos se apoyan entre sí al aprender constantemente. Técnicamente esto se explica a través de la función de pérdida.

#### 4.1.4 Función de pérdida

Antes de profundizar en la función de pérdida debemos comprender las nociones probabilísticas que la forman,

Empecemos con entender el sample space( espacio muestral ), el espacio muestral son todos los valores que un modelo toma de una observación “x”. En el caso del Generador el espacio muestral se refiere al conjunto de todos los datos posibles que el generador puede producir, es importante anotar que el espacio muestral del generador está limitado según la arquitectura de la red y el tipo de entrenamiento que recibe. Por lo que a el discriminador se refiere como mencionado anteriormente, el sample space consiste en la clasificación binaria ( aprox 0: falsos, aprox 1: real)

Las Redes Generativas Antagónicas (GANs) representan un enfoque poderoso en el aprendizaje profundo para generar datos que imitan la distribución de un conjunto de datos reales. A diferencia de otros modelos que calculan directamente la función de densidad de probabilidad (PDF), las GANs operan mediante un mecanismo de densidad implícita. Esto significa que, aunque las GANs aprenden a producir muestras que se asemejan a los datos reales, no modelan explícitamente la PDF de esta distribución.

Segun David Foster [1] en GANS, consideramos que tenemos un conjunto de datos reales cuya distribución queremos modelar,  $p_{real}(x)$ , y una distribución generada por el generador,  $p_{gen}(x)$ . Normalmente al inicio  $p_{gen}(x)$  sera significativamente diferente de  $p_{real}(x)$  ya que el modelo genera ruido aleatorio inicialmente, pero el objetivo del *training* de GANS es ajustar  $p_{gen}(x)$  para que se parezca lo mas posible a  $p_{real}(x)$ .

Por otro lado El generador  $G$  crea un ruido aleatorio  $z$ , con su propia distribución y lo transforma una salida  $x_{gen} = G(z)$  que intenta imitar la distribución de los datos reales. La tarea del discriminador  $D$  es evaluar  $x$  y estima la probabilidad de que  $x$  venga de los datos reales en vez de los generados.Más adelante nos adentraremos mejor en como funciona y como conceptualizar el *training* de GANS

GANS aprende a traves de el ajuste de parámetros  $\theta$  permitiendo generar datos que sigan la misma distribución de los datos reales.

La función parametrica del Generador es  $G(z;\theta_g)$ ,  $z$  presenta el ruido aleatorio y  $\theta_g$  representa los parámetros en este caso los pesos y sesgos de la red neuronal profunda. El Generador intenta ajustar sus parámetros para que  $G(z;\theta_g)$  genere datos cada vez mas parecidos a los reales

La función paramétrica del discriminador  $D(x; \theta_d)$ ,  $x$  presenta los datos de entrada y  $\theta_d$  representa los parámetros en este caso los pesos y sesgos de la red neuronal profunda. El discriminador intenta ajustar sus parámetros para que  $D(x; \theta_d)$  mejore su capacidad de distinción.

Una vez comprendido esto tenemos que entender técnicamente como funciona el entrenamiento simultáneo del Generador  $G$  y el Discriminador  $D$ , este funciona a través de la **función de pérdida (Loss function)**. La función de pérdida, guia tanto al  $D$  como al  $G$  para a través de una función **minimax**

La función de pérdida de las GANs está diseñada para cumplir dos objetivos principales:

- Discriminador: Maximizar su capacidad para diferenciar entre datos reales y datos generados por el generador.
- Generador: Minimizar la capacidad del discriminador para diferenciar entre datos reales y generados, esencialmente engañándolo para que clasifique los datos generados como reales.

Esta dinámica se captura en la siguiente función de valor

\*Recordatorio: teniendo en cuenta aprox 0 es falso(generado) y aprox 1 es real \*

$$\min_G \max_d V(D, G) = E_{x \sim p_{data(x)}} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

- $x$  son los datos reales
- $z$  es el ruido de entrada al *Generador*
- $G(z)$  son los datos generados a partir de  $z$
- $D(x)$ , probabilidad estimada por el discriminador de que  $x$  sea un dato real
- $D(G(z))$  se refiere a la probabilidad estimada por  $D$  de que los datos generados por el  $G(z)$  son datos reales

$E_{x \sim p_{data(x)}} [\log D(x)]$  : representa la expectativa de que el discriminador clasifique correctamente los datos reales como reales.

Función de pérdida del Generador—> $E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$

$E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$  : representa la expectativa de que el discriminador clasifique los datos generados como falsos.

- El Discriminador busca maximizar  $V(D,G)$  haciendo que  $D(x)$  sea grande (aprox 1 real) para datos reales (maximizando  $\log D(x)$ ) y que  $D(G(z))$  sea pequeño (aproximándose a 0, generado) maximizando así  $\log(1 - D(G(z)))$
- El Generador busca minimizar  $V(D,G)$  haciendo que  $D(G(z))$  sea lo mas aproximado a 1 posible debido a  $\log(1 - D(G(z)))$  si minimizamos este log quedaría un valor aprox a 1

Cabe recalcar un apunte muy importante que pasa por desapercibido en la función de pérdida por lo que respecta al generador, el siguiente;

Tiene sentido minimizar  $E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$ , sin embargo, en el comienzo del entrenamiento de las GANS el discriminador es muy fuerte siempre en las fases iniciales, lo que puede conllevar a gradientes débiles y a un riesgo de convergencia de mínimos, donde el aprendizaje del G se estanque.

Por ello por términos de evitar el riesgo de convergencia de mínimos, es una práctica común es maximizar  $E_{z \sim p_z(z)}[\log(D(G(z)))]$  lo cual produce el mismo efecto, minimizando  $D(G(z))$  aprox 0, pero  $\log(0)$  es aprox. 1. Es lógico que esta estrategia cogerá fuerza en fases iniciales ya que el discriminador identifica mucho mejor los datos falso dando valores de  $D(G(z))$  aprox a 0 (fake) haciendo que  $\log(D(G(z))$  resulte en 1. Por esta razón en los primeros *epochs* solemos ver cambios más notables en los datos generados.

Desde otro punto de vista, en la fase inicial como hemos mencionado  $D(G(z))$  dará valores muy pequeños, haciendo que  $\log(1-D(G(z))$  nos de valores muy pequeños, lo que hará a el modelo engancharse en los valores pequeños y avanzar a un ritmo realmente lento, haciendo que los gradientes (estado del aprendizaje) se convergen con los mínimos.

## 4.2 FUNCIONES DE ACTIVACIÓN

Las funciones de activación son capas (layers) esenciales para la construcción de las redes profundas de GANS, estas son utilizadas para adaptar los datos recibidos para fortalecer el aprendizaje de estos., las funciones de activación mas utilizadas son :

#### 4.2.1 ReLU (Rectified Linear Unit)

La capa *ReLU*, transforma aquellos valores negativos recibidos en el input de la capa, siempre a 0, en caso de que los valores del input sean positivos estos se mantendrán igual, se utiliza en las hidden layers(capas intermedias u ocultas).

Sin embargo el uso de ReLU en GANS conlleva muchos riesgos, como dejar las neuronas(Unidad de procesamiento) muertas, es decir aquellas neuronas que han recibido un input de número negativo transforman este valor a 0 pero este valor no es útil dejando a las neuronas inútiles y sin aprovechar para futuros aprendizaje durante el ***backpropagation***. Una alternativa a RELU es LeakyReLU.

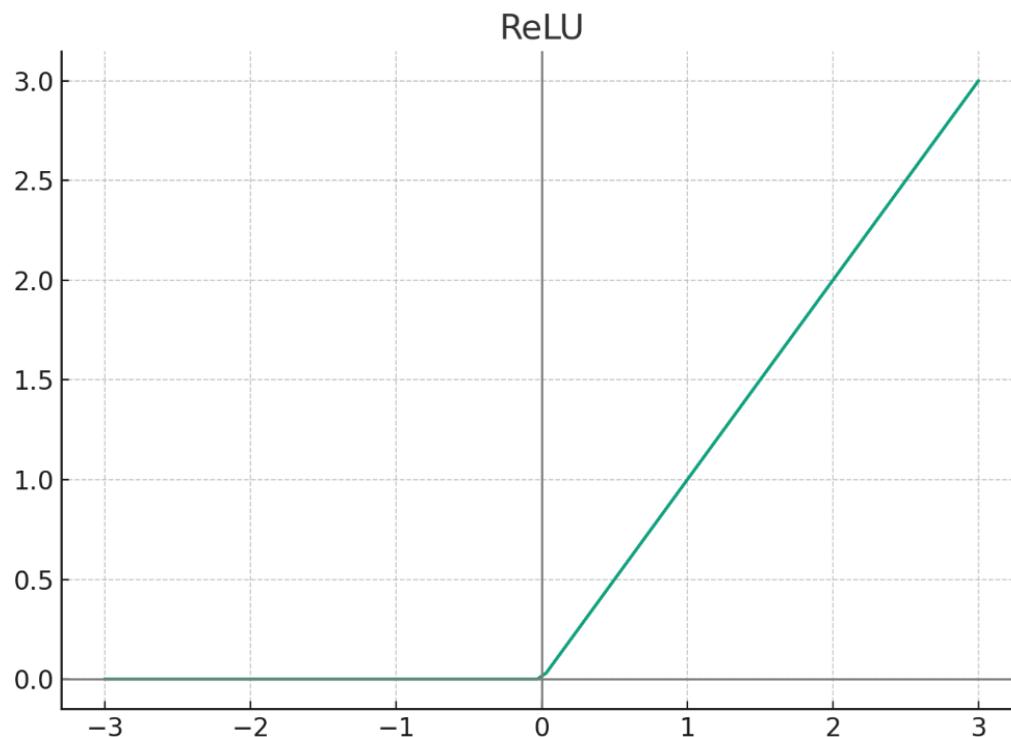


Figura 5: *ReLU*

#### 4.2.2 Leaky ReLU

Leaky ReLU, tiene funcionamiento igual a ReLU para los valores positivos de entrada, sin embargo para los valores negativos, tras aplicar un pequeño coeficiente  $\alpha$  (e.g., 0.01) a estos valores se devuelve un valor de salida muy pequeño proporcional a el valor negativo, solucionando asi el problema del gradiente 0 proporcionando un pequeño número negativo para aprender de esa neurona tras el ***backpropagation*** de la red, evitando neuronas muertas.

fórmula:  $f(x) = \max(\alpha x, x)$

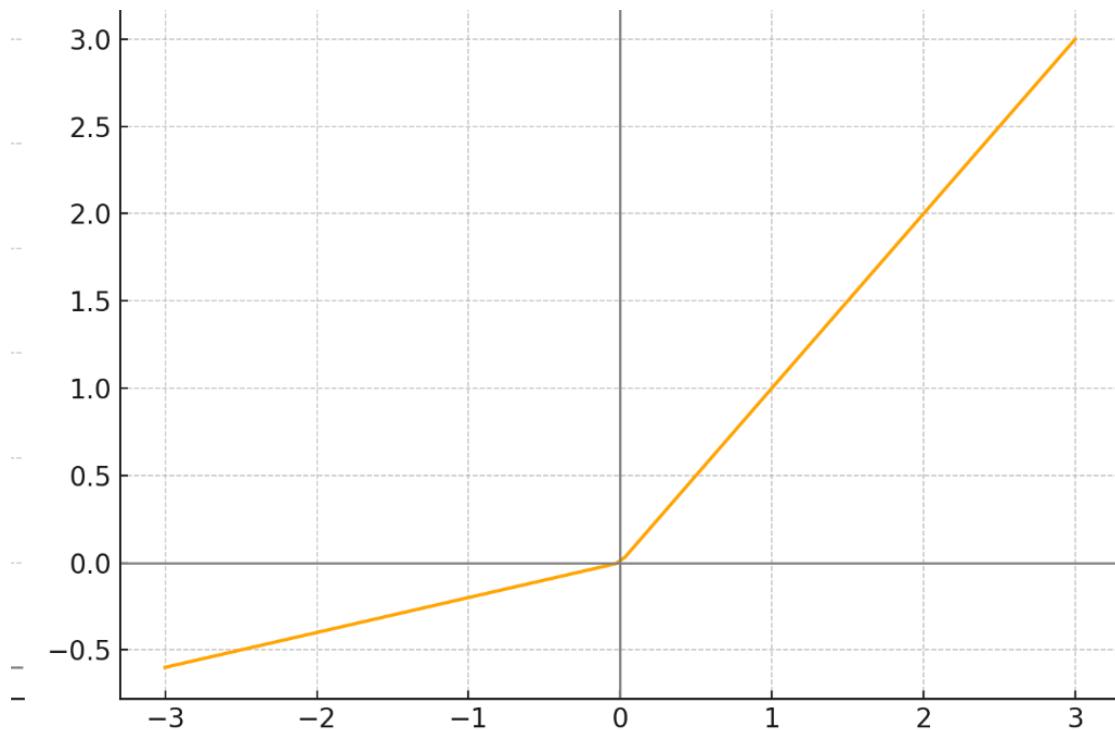


Figura6: LeakyReLU

#### 4.2.3 Tangente Hiperbólica(Tanh)

La tangente hiperbólica consiste en una capa de función activación que agiliza y mejora el aprendizaje de los datos, su función es mantener seguros que los datos de salida el generado se concentren en un rango de valores entre [-1,1]. Los datos reales, como las imágenes, son preprocesados para tener valores en el rango [-1, 1], lo que hace que la salida de Tanh sea ideal para modelar este tipo de datos. Al generar datos que ya están en el rango esperado, se facilita el aprendizaje y la evaluación por parte del discriminador.

$$\text{fórmula: } \text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

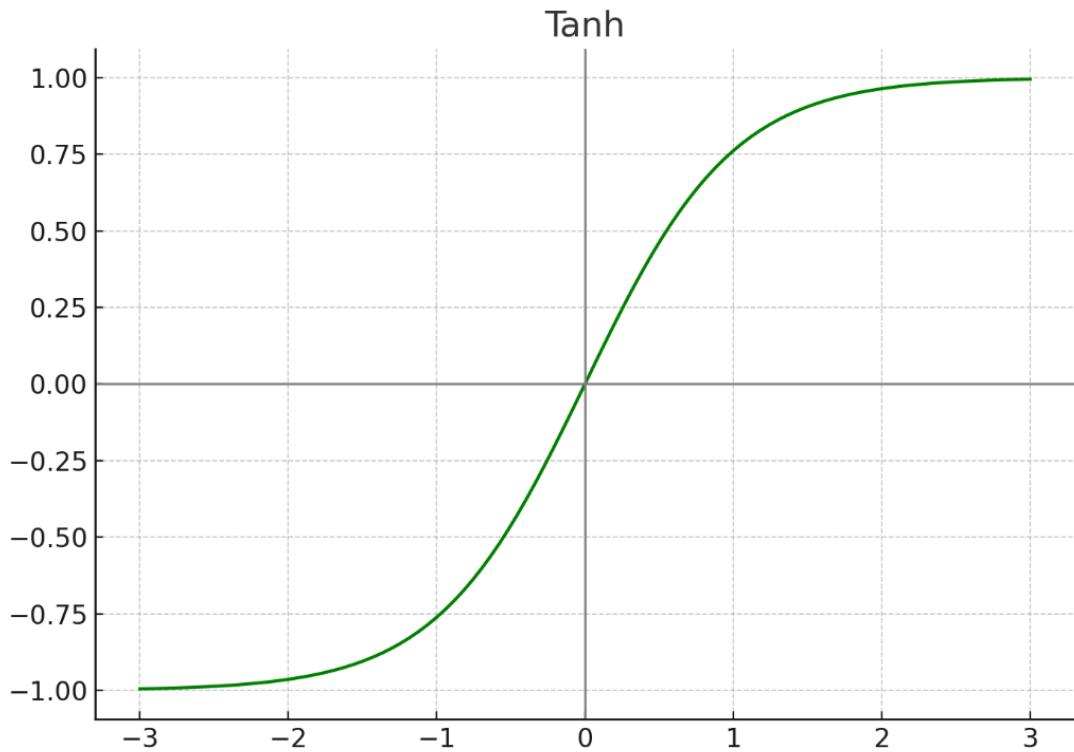


Figura 7: Tangente Hiperbolica

#### 4.2.4 Sigmo ID

La capa Sigmo ID mapeo los valores de entrada entre 0 y 1 en la capa de salida del Discriminador para clasificar las entradas como reales o generadas.

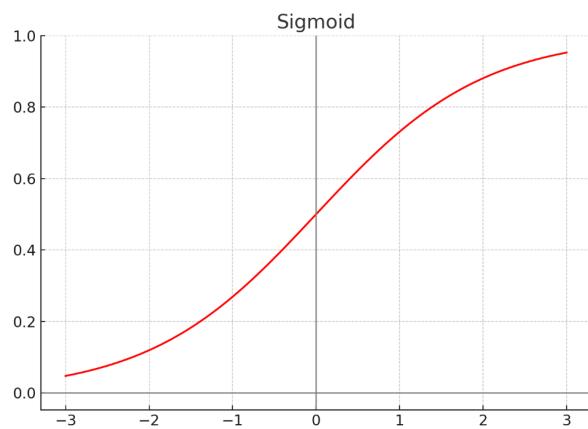


Figura 8: SigmoID

$$\text{fórmula: } f(x) = \frac{1}{1+e^{-x}}$$

[8]. Ya vistas la funciones de activación seguimos con las convoluciones

### 4.3 CONVOLUCIONES

Conv Layers son extremadamente importantes en el aprendizaje profundo especialmente en aquellas redes que trabajan con datos no estructurados, como GANS y CNN(convolutional networks). Antes de profundizar en estas capas, hay que comprender ciertos aspectos fundamentales como las convoluciones, filtros, paddings y strides

#### 4.3.1 Capas Convolucionales

Una convolución es una operación matemática en Deep Learning que procesa dos funciones para producir un resultado. En el caso de las GANS, se utiliza para aplicar filtros a los datos espaciales de las imágenes, extrayendo así gracias a los filtros características como patrones, sombreados, texturas y bordes complejos de las imágenes, las convoluciones no transpuestas se utilizan en el discriminador y las transpuestas en el generador.

El filtro suele ser normalmente una matriz pequeña que se va aplicando y deslizando sobre las porciones de la imagen extrayendo así a través del filtro características esenciales de la imagen, el valor obtenido tras aplicar el filtro se guarda en el mapa de características. Este mapa de características será la salida tras aplicar los filtros a todas las porciones de la imagen.

Primero visualizaremos un ejemplo sencillo y luego uno más complicado para poder entender el funcionamiento mejor

ENTRADA(*Porcion img*)      FILTRO

$$\begin{bmatrix} -1 & -0.5 & 0 \\ 0.5 & 1 & 0.5 \\ 0 & -0.5 & -1 \end{bmatrix} \times \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$M_{1,1} = (-1 \cdot -1) + (-0.5 \cdot 1) + (0.5 \cdot 1) + (1 \cdot -1) = 0$$

$$M_{1,2} = (-0.5 \cdot -1) + (0 \cdot 1) + (1 \cdot 1) + (0.5 \cdot -1) = 1$$

$$M_{2,1} = (0.5 \cdot -1) + (0 \cdot 1) + (0 \cdot 1) + (-0.5 \cdot -1) = 1$$

$$M_{2,2} = (1 \cdot -1) + (0.5 \cdot 1) + (-0.5 \cdot 1) + (-1 \cdot -1) = 0$$

*M : mapa de características es igual a*

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Podemos deducir de esta aplicación de un filtro  $2 \times 2$  a una porción  $3 \times 3$  dimensional de una imagen, obtenemos una porción del futuro mapa de características de la imagen.

Otro ejemplo:(Kernel=Filtro)

Input	Kernel	Output																													
<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>2</td><td>0</td></tr> <tr><td>0</td><td>3</td><td>4</td><td>5</td><td>0</td></tr> <tr><td>0</td><td>6</td><td>7</td><td>8</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	0	0	0	0	0	1	2	0	0	3	4	5	0	0	6	7	8	0	0	0	0	0	0	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>0</td><td>1</td></tr> <tr><td>2</td><td>3</td></tr> </table>	0	1	2	3	$\star$
0	0	0	0	0																											
0	0	1	2	0																											
0	3	4	5	0																											
0	6	7	8	0																											
0	0	0	0	0																											
0	1																														
2	3																														
		$=$																													
		<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>0</td><td>3</td><td>8</td><td>4</td></tr> <tr><td>9</td><td>19</td><td>25</td><td>10</td></tr> <tr><td>21</td><td>37</td><td>43</td><td>16</td></tr> <tr><td>6</td><td>7</td><td>8</td><td>0</td></tr> </table>	0	3	8	4	9	19	25	10	21	37	43	16	6	7	8	0													
0	3	8	4																												
9	19	25	10																												
21	37	43	16																												
6	7	8	0																												

Figura 9 : Ejemplo práctico de aplicación de filtro

Si aplicamos el filtro a todas las porciones de la imagen de izquierda a derecha obtendremos y de arriba a abajo guardando los valores en el mapa de características,obtendremos un mapa completo de características que sigue una característica árticular del input, por ejemplo :



Figura 10: Ejemplo de patrones de Filtros

Las capas convolucionales son simplemente una colección de filtros, donde los **pesos** aprendidos por las redes neuronales son los valores de estos filtros, que van entrenando y aprendiendo y ajustándose cuando encuentran características como combinaciones de colores profundos, bordes etc.

#### 4.3.2 Strides y Padding

El stride determina cuantos pixeles de la imagen se mueve el filtro a través de la entrada, significando que un stride de 1 procesa un píxel cada vez en cada ubicación en la entrada. Sin embargo si el stride es mayor que 1, como 2 por ejemplo querrá decir que se procesa 1 pixel en cada dos ubicaciones de entrada lo que da como resultado una imagen de tamaño menor (en este caso casi a la mitad), esta técnica es muy aplicada en las convoluciones del discriminador.

El padding es un elemento diferenciador en las capas convolucionales del discriminador, el padding introduce 0 ceros en la matriz de los valores de la imagen, para asegurar que la pérdida de información en los bordes sea mínima, al minimizar la imagen aplicando un stride mayor que 1, para compensar el desplazamiento natural por el tamaño del filtro

¿Cómo calculamos el tamaño de salida ?

Tras aplicar una capa convolucional la operación seguirá la siguiente fórmula

$$\text{Tamaño de salida} = \frac{W-F+2P}{S} + 1 \quad [27]$$

W es el tamaño de la entrada F es el tamaño del kernel o filtro, P es el padding y S el stride, apliquemos lo mencionado anteriormente con un ejemplo:

Imaginemos que queremos aplicar una capa convolucional a una imagen de tamaño 64 x64, W=64, y utilizamos un filtro kernel 3x3 con un stride s2 y usando un padding = "same"

Para padding "same", el objetivo es producir una salida con las mismas dimensiones espaciales que la entrada cuando el stride es 1. Sin embargo, con un stride de 2, necesitamos calcular cuánto padding es necesario para mantener el tamaño de la salida lo más cerca posible al tamaño de la entrada dividido por el stride. Para simplificar, asumimos que el padding necesario para mantener el tamaño "same" es justo suficiente para compensar el tamaño del filtro.

En este caso sería 1, para que el filtro se ajuste exactamente al borde de la imagen original con ceros añadidos.

Si aplicamos la fórmula:

$$\text{Salida} = \left( \frac{64 - 3 + 2 \times 1}{2} \right) + 1 = 32$$

Esto nos produce una imagen de salida de tamaño 32x32, si aplicamos otra capa convolucional el resultado será 16x16 y así sucesivamente.

El objetivo del discriminador como veremos en DCGANS sera reducir la imagen de entrada (imagen generada), extrayendo así las características de la imagen hasta un tamaño de 1x1, que indicara un valor aprox a 0 falso o 1 verdadero

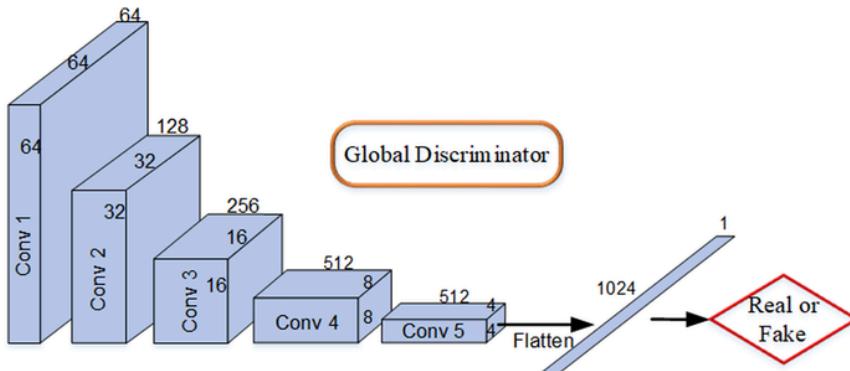


Figura 11: Ejemplo Discriminador [19]

#### 4.3.3 Capas Convolucionales Transpuestas

Las convoluciones transpuestas realizan el proceso inverso a las convoluciones standard del discriminador, las traspuestas se utilizan en el generador para que a partir del vector ruido o latente ( $z$ ) se genere una imagen de las mismas dimensiones y tamaño que las reales proporcionadas por el dataset, aumentando la resolución de esta imagen.

Como hemos mencionado, estas convoluciones aumentan el tamaño de las entradas, esto se consigue mediante la dilatación de la entrada con ceros, es decir mediante la inserción de ceros en el mapa de características de la imagen

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad M \text{ aumentada} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 0 & 0 \\ 3 & 0 & 4 \end{bmatrix}$$

Esto es un ejemplo el resultado de la salida dependerá del tamaño del filtro y el stride

la fórmula para calcular el tamaño de salida es:

$$O = S \cdot (I - 1) + F - 2P$$

O es la salida, I la entrada, S el stride, F el tamaño del filtro, y P el padding

Podemos observar que esta vez el stride funciona de manera diferente ya que contribuye a el crecimiento de las dimensiones de la imagen. Para las convoluciones transpuestas, un stride de más de 1 indica el espaciado entre los valores originales de la entrada en la matriz dilatada. Es decir indica la cantidad de ceros insertados entre los valores de el mapa de entrada por ejemplo un stride de 2 quiere decir que se insertara un cero entre cada valor.

El padding ahora servirá para la preservación de dimensiones, imaginemos este ejemplo:

queremos generar una imagen de  $6 \times 6$ , bien a partir del vector ruido aplicamos una capa convolucional transpuesta de stride 2 y filtro 3 tal que así

$$O = 2 \cdot (2 - 1) + 3 - 2 \cdot 0 = 5$$

Sin embargo esto resulta en una salida de  $5 \times 5$  es aquí donde entra el padding, el padding rellenara el borde de esta matriz de 0s para ajustar el pequeño error convirtiéndola en una imagen  $6 \times 6$ , en este caso ajustaríamos el padding a 0.5

$$O = 2 \cdot (2 - 1) + 3 - 2 \cdot 0.5 = 6$$

Ejemplo Grafico

de  $1 \times 1$  a  $64 \times 64$  mediante convoluciones transpuestas

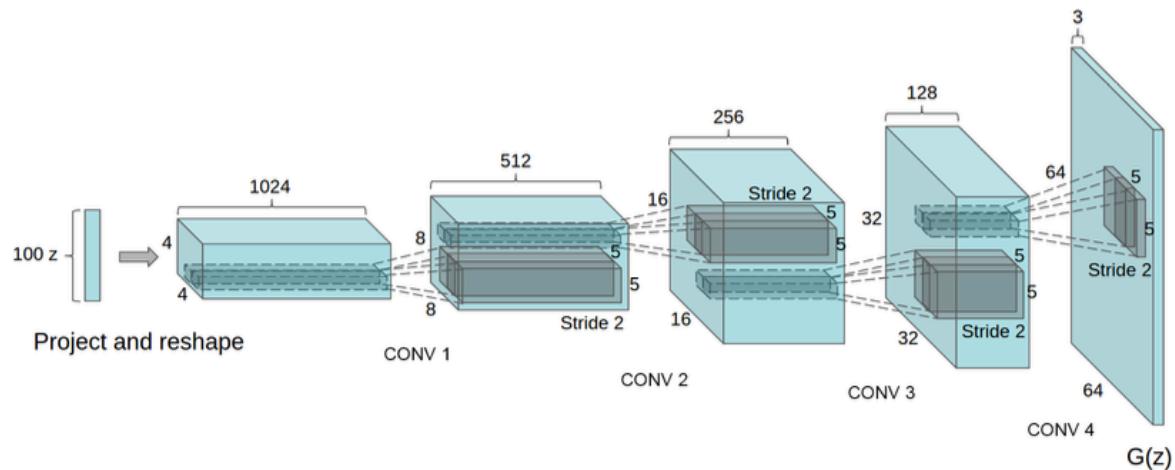


Figura 12: Proceso de Generador con Convoluciones Transpuestas [28]

## 4.4 LAYERS/CAPAS

### 4.4.1 Batch Normalization

Normalización de lotes en castellano, es una capa que no es estrictamente necesaria aplicarla en una red neuronal de GANS, pero si se aplica, juegan un papel fundamental al estabilizar y acelerar el entrenamiento de las redes neuronales mediante la normalización de las salidas de las capas de las funciones de activación, esto se hace ajustando las funciones de activación para que tengan una media cercana a 0 y una desviación estándar cercana a 1 [20].

Esta normalización se aplica por lotes de datos, y se aplica por separado a cada característica específica del *batch(lote)*, para una característica específica en un mini-batch, BN normaliza las salidas de la capa anterior según la siguiente fórmula:

$$\mu_{\beta} = \frac{1}{m} \sum_{i=1}^m x_i \quad \sigma_{\beta}^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\beta})^2$$

Calculamos la media  $\mu_{\beta}$  del lote de datos y  $\sigma_{\beta}^2$  es la varianza,  $x_i$  son los datos

Una vez extraída la media y la varianza normalizamos los datos :

$$\hat{x}_i = \frac{x_i - \mu_{\beta}}{\sqrt{\sigma_{\beta}^2 + \epsilon}} \quad \epsilon \text{ es una constante muy pequeña para evitar la división por 0}$$

Acto seguido escalamos y desplazamos los datos de las activaciones ya normalizados

$$y_i = \gamma \hat{x}_i + \beta$$

$y_i$  son los datos de salida tras la normalización de los datos la escalación a partir del parámetro  $\gamma$  y del desplazamiento  $\beta$ , ambos parámetros se ajustan y son aprendidos durante el entrenamiento.

Consideremos una GAN entrenada para generar imágenes. Sin BN(batch normalization), las capas del generador pueden empezar a producir activaciones con rangos muy variados, complicando el aprendizaje del discriminador y viceversa, al aplicar BN, las activaciones se estabilizan.

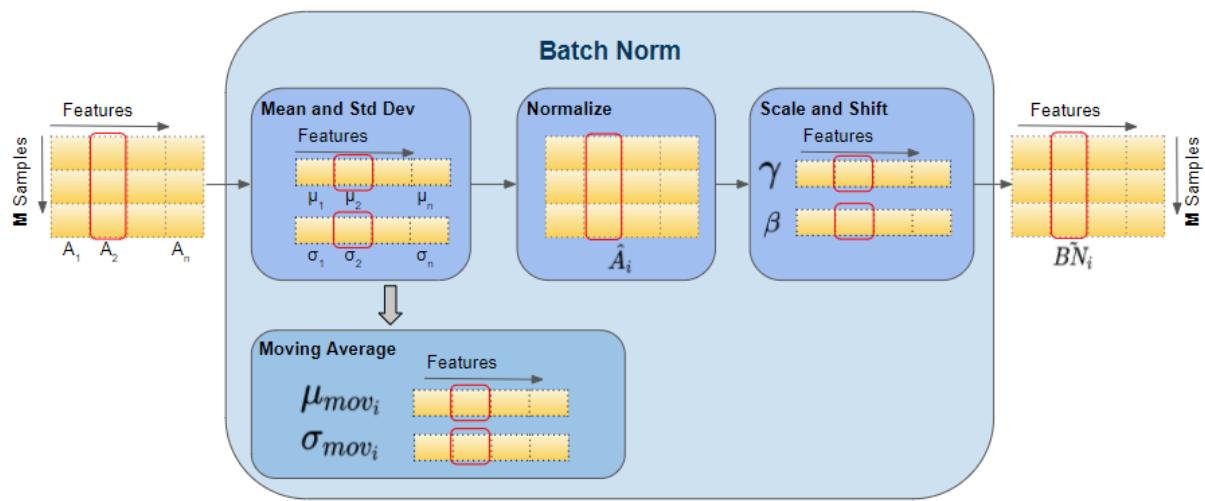


Figura 13: Funcionamiento Batch Normalization

#### 4.4.2 Dense Layer

La *capa densa* o *dense layer* es fundamental en muchas arquitecturas de deep learning como en las *redes generativas antagónicas*, esta capa conecta todas y cada una de las neuronas de la capa anterior con todas y cada una de la capa siguiente facilitando el aprendizaje al facilitar la modelación de relaciones de las características (*features*) de los datos de entrada

Los datos de entrada requieren forma de vector

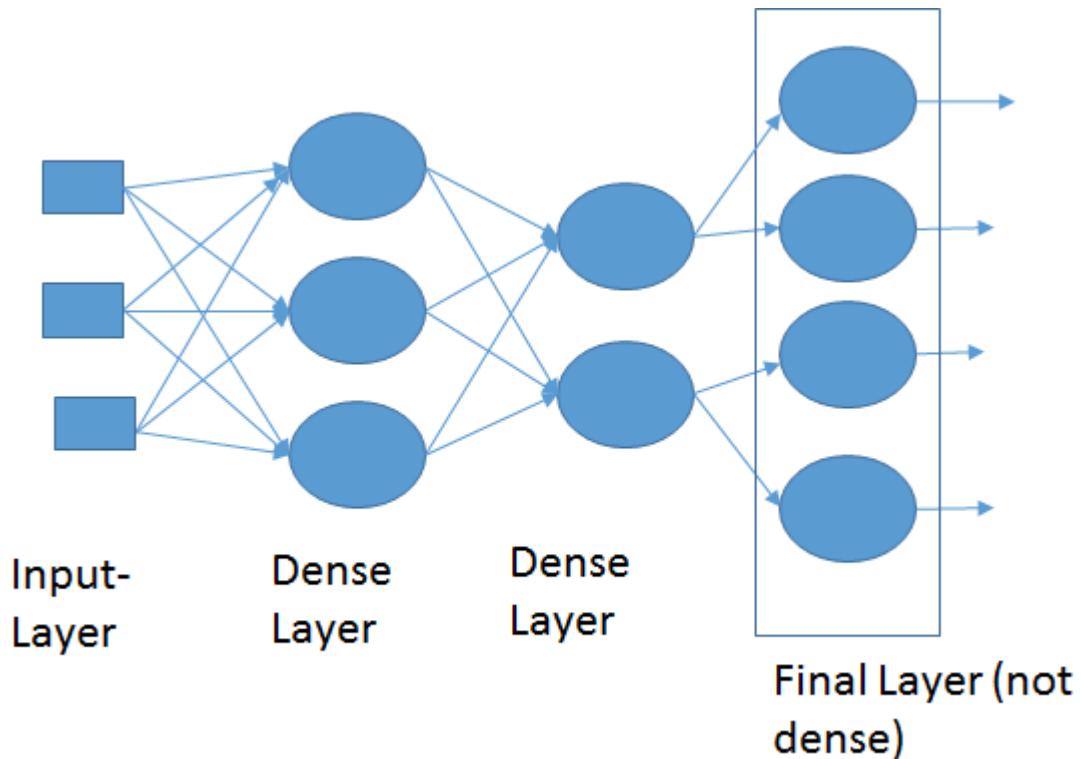


Figura 14: Capa Densa

En el esquema anterior podemos fijarnos en como las unidades de procesamiento(neuronas) dense layers estan completamente conectadas entre si.

Una capa densa funciona a través de la siguiente fórmula

$$y = f(Wx + b)$$

$x$  es el vector de entrada,  $W$  es la matriz de pesos de la capa,  $f$  es la función de activación aplicada a los datos de salida de la capa,  $y$  es el vector de salida y  $b$  son los sesgos.

*Los sesgos son otro conjunto de parámetros en los modelos de redes neuronales que se suman a la suma ponderada de las entradas y los pesos antes de aplicar la función de activación. Hay un sesgo asociado con cada neurona en la capa siguiente. Estas permiten que la neurona tenga una flexibilidad adicional evitando una salida nula activando así la neurona*

- **Pesos:**  $W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$
- **Sesgos:**  $b = [b_1, b_2]$
- **Salida:**  $y = Wx + b$

La operación completa sería:

$$\begin{aligned} y &= \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [b_1, b_2] \\ y &= \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix} \end{aligned}$$

Las capas densas suelen encontrarse tanto en el generador como en el discriminador. En el generador, las capas densas pueden utilizarse para transformar un vector de ruido aleatorio de entrada en una representación más compleja y dimensionalmente mayor, que eventualmente se moldea en los datos generados (por ejemplo, una imagen) y en el discriminador, las capas densas ayudan a analizar y clasificar las entradas como reales o sintéticas basándose en las características aprendidas durante el entrenamiento.

Como ejemplo para visualizarlo mejor imagina una GAN simple diseñada para generar imágenes a partir de un vector de ruido. El generador podría empezar con una capa densa que toma el vector de ruido (digamos, de 100 dimensiones) y lo transforma en una representación más grande (por ejemplo, 1024 dimensiones), seguida de más transformaciones hasta alcanzar la forma deseada de la imagen. El discriminador tomaría una imagen como entrada, la aplataría si es necesario, y luego utilizaría capas densas para extraer características y finalmente producir una clasificación

#### 4.4.3 Flatten Layer

La Capa flatten juega un papel muy importante en el discriminador, es la capa responsable de transformar una entrada multidimensional en un vector unidimensional, para conectar este vector unidimensional con las capas densas(solo admiten vectores).

Hasta ahora hemos tomado en cuenta la altura y anchura de las imágenes como dimensiones espaciales de la capa convolucional del discriminador sin embargo, no nos hemos centrado en los canales (channels), los canales sin la cantidad de filtros que se han aplicado en una capa convolucional, digamos que si hemos aplicado 8 filtros a una capa convolucional quiere decir que tendremos 8 mapas de características.

El filtro transforma la entrada multidimensional tal que así  $W(\text{ancho}) \times H(\text{altura}) \times C$  (canales), quiere decir por ejemplo que si tenemos una capa convolucional  $4 \times 4 \times 8$ , esto resultará en un vector resultante de 128 elementos.

Una manera de visualizar la operación de Flatten es imaginar un cubo de lego donde cada bloque representa un elemento en el tensor de entrada. La capa Flatten desmonta este cubo y alinea todos los bloques en una única fila. Como vemos en el siguiente ejemplo:

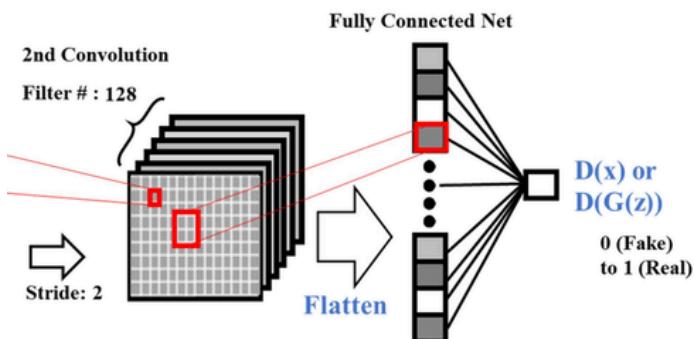


Figura 15: Capa Flatten

## 4.5 VARIABLES FUNDAMENTALES

### 4.5.1 Gradientes

Los gradientes son la base del aprendizaje en las redes neuronales como GANS, un gradiente es la derivada de la función de pérdida respecto a los parámetros de la función(pesos y sesgos) :  $\Delta_{\theta} J(\theta)$ , también la dirección en la que se tienen que ajustar los pesos y sesgos para maximizar o minimizar la función de pérdida

En el caso de GANS, los gradientes indican en el generador como o hacia qué dirección tiene que actualizar sus parámetros para producir datos más realistas o creíbles y para el discriminador como debe ajustar sus parámetros para mejorar su habilidad en la clasificación

Hay dos problemas muy comunes en GANS que involucran los gradientes

- Explosión de Gradientes:

Cuando el LR o los hiperparámetros son demasiados grandes, se causan amplias oscilaciones que llevan a una falta de convergencia del modelo

- Desvanecimiento de Gradientes: Cuando el discriminador es muy bueno, los gradientes son muy pequeños y los pesos y sesgos apenas se actualizan, dejando a el generador obsoleto.

#### **4.5.2 Learning Rate/ Tasa de Aprendizaje**

La tasa de aprendizaje (Learning Rate ) controla el tamaño de los pasos que se dan para actualizar los pesos del modelo en respuesta al gradiente de la función de pérdida y es esencial para la fase de entrenamiento tanto para el generador como para el discriminador.

El valor de la tasa de aprendizaje es muy crítica para el modelo un pequeño ajuste a esta y el algoritmo puede cambiar drásticamente. haciendo que el discriminador domine al generador, cayendo el gradiente en un riesgo de convergencia de mínimos

Cuando la tasa de aprendizaje es muy alta los pesos se actualizan en exceso y puede que no se extraigan características esenciales de los datos, mientras que si es demasiado baja puede ser muy lenta o converger con los mínimos locales.

La actualización de los pesos en el entrenamiento de redes neuronales, incluidas las GANs, se realiza generalmente de acuerdo con la fórmula:

$$\theta = \theta - LR \cdot \Delta_{\theta} J(\theta)$$

$\theta$  son los pesos,  $\Delta_{\theta} J(\theta)$  es el gradiente respecto a la función de pérdida del discriminador dados los pesos  $\theta$  o del generador.

El learning Rate es un parámetro muy sensible, debido a que es directamente proporcional a el gradiente, que se decide antes del entrenamiento por el diseñador o programador del modelo.

Supongamos que estás entrenando una GAN para generar imágenes. Si eliges un LR de 0.01 para el generador y de 0.001 para el discriminador, este desequilibrio podría hacer que el discriminador aprenda más lentamente que el generador. Como resultado, el generador podría encontrar trucos para "engaños" al discriminador sin aprender a generar imágenes de alta calidad.

Es por esto que la elección y ajuste de el LR es fundamental para el entrenamiento de las GANS, para el ajuste de el LR se utiliza el optimizador ADAM.

#### **4.5.3 Optimizador ADAM**

El optimizador Adam(Adaptive Moment Estimation) es un algoritmo para el ajuste de la tasa de aprendizaje LR de cada parámetro de manera adaptativa, el algoritmo adam mezcla bases de el algoritmo Momentum y RMSprop(root mean square) [9] [1].

Adam consiste en dos momentos que estiman respectivamente el promedio móvil del gradiente y la media móvil del cuadrado del gradiente, primero actualizamos los momentos, después corregimos los momentos y actualizamos los parámetros respecto a el LR

Media Móvil del Gradiente:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t \quad [9] [29]$$

Media Móvil del Cuadrado del Gradiente

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2 \quad [9] [29]$$

- $g_t$ : gradiente en el tiempo t

- $\beta_1$ y  $\beta_2$  son los coeficientes de decaimiento exponencial para estimar los momentos o medias, generalmente son 0.9 y 0.999 aunque son modificables

Las medias móviles pueden estar sesgadas a 0 al inicio del entrenamiento sobretodo cuando se inicializan como ceros, por esta razón se aplica la siguiente corrección de sesgos

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t} \quad [30]$$

$$\widehat{v}_t = \frac{v_t}{1-\beta_2^t} \quad [30]$$

Ahora se actualizan los pesos

$$\theta_{t+1} = \theta_t - \frac{LR}{\sqrt{\widehat{v}_t + \epsilon}} \cdot \widehat{m}_t \quad [30]$$

- $\theta_t$  son los pesos en el tiempo t

-LR: tasa de aprendizaje

$-\epsilon$  : número minusculo para evitar la division entre 0

Como podemos observar adam ajusta adaptativamente los learning rate por cada parametro en el tiempo y momentos de los gradientes.

Adam proporciona un balance entre eficiencia computacional y estabilidad del entrenamiento, facilitando el desarrollo de modelos generativos potentes y versátiles. Sin embargo, la elección del optimizador y sus hiperparámetros siempre debe ser considerada cuidadosamente y adaptada a las necesidades específicas del modelo y del conjunto de datos.

## 5.TIPOS DE REDES GENERATIVAS ADVERSARIALES

---

### 5.1DEEP CONVOLUTIONAL GANS

Las deeps convolutional Gans o convoluciones redes generativas adversariales profundas funcionan exactamente como explicado desde el punto *4.1 al 4.5*

### 5.2 WASSERSTEIN GRADIENT PENALTY GANS

WGAN (wasserstein GAN), es un tipo de red generativa adversarial un tanto diferente a lo comentado previamente, Este tipo de GAN ha sido probada y aplicada en varios estudios [10] comprobando y afirmando que su estabilidad es mucho mejor.

El principal cambio que aplica WGAN a las redes generativas adversariales tradicionales vistas anteriormente, es el cambio de la función de pérdida para el discriminador como para el generador resultando en una convergencia estable para ambas.

#### 5.2.1 función de pérdida

Esta nueva función de pérdida ya no utiliza BCE (binary cross entropy) entropía de cruce binaria, lo que quiere decir que los valores no estarán comprendidos entre 0 y 1, sino entre -1(fake) y 1(real). **La salida del discriminador ya no será una probabilidad sino un puntaje/distancia.** En wasserstein a partir de ahora nos referiremos a el discriminador como crítico

Es decir, la función de pérdida de wasserstein mide la distancia o diferencia entre la distribución generada de la real, $\max_d(E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)])$  [10].

Como observamos la función de pérdida del crítico o discriminador trata de maximizar la diferencia o “distancia” entre las predicciones del generador como las del discriminador

EL generador va tratar de generar las imágenes lo menos diferentes del valor de  $D(x)$  para generar menos diferencia.

A diferencia de las Gans Tradicionales es mucho más efectivo indicarle al generador la distancia de mejora para parecerse a los datos reales que un label de 0 o 1 como en las gans tradicionales.

### 5.2.2 Gradient Penalty

Sin embargo al no seguir una entropía binaria, es muy probable que la generación de funciones de pérdida sin restricciones pueda generar funciones de pérdida muy altas para el generador o para el discriminador, generando o sobreexploración de gradientes o el desvanecimiento de estos.

Para solucionar este problema debemos intentar que nuestro modelo WGAN siga la función continua 1-Lipschitz

La función dice lo siguiente:

La división entre la diferencia absoluta de las predicciones y la diferencia absoluta de la media de los pixeles tiene que ser igual o menor que 1 (será la constante Lipschitz)

$$\frac{|D(x_1) - D(x_2)|}{|x_1 - x_2|} \leq 1 \quad [10]$$

Seguir esta restricción o condición nos asegura que la evolución será continua evitando valores infinitos positivos como negativos y así evitando que el gradiente no sobreexplote ni se desvanezca

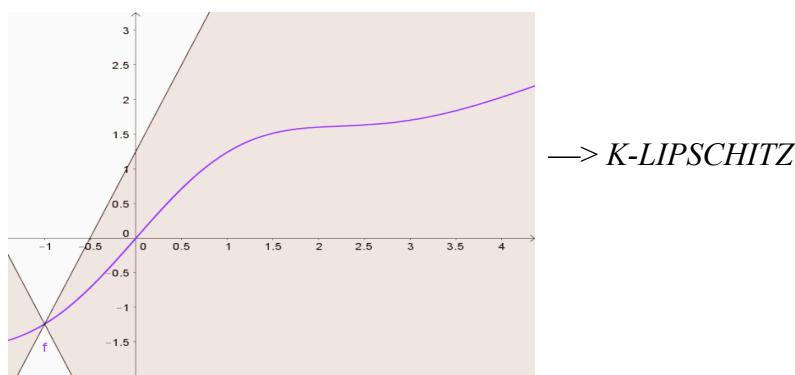


Figura 16: Constante Lipschitz

Para cumplir esta condición y por ende evitar los problemas de los gradientes, la primera solución que se trato fue realizada por Martin Arjovsky en el paper inicial de las WGAN [], mediante el ajuste de pesos o “*weight clipping*”. Se sugirió limitar los pesos del crítico(discriminador) a un rango pequeño de valores como puede ser [-0.001,0.001], sin embargo esta técnica desperdicia el discriminador y lo debilita haciendo que el generador gane demasiada fuerza, haciendo al discriminador muy débil. Algunos pueden pensar que esto es bueno pero recordemos que si el discriminador no es lo bastante fuerte el generador no puede aprender de este y viceversa como mencionamos en el ejemplo de los ladrillos por lo tanto no es una buena solución para seguir

Sin embargo Ishaan Gulrajani, propone una técnica para seguir la condición de Lipschitz, mediante la penalización del gradiente ***gradient penalty***, penalizando las desviaciones del gradiente respecto a la constante lipsitch en nuestro caso 1. Esto permite seguir la condición **Lipschitz** proporcionando una mayor estabilidad al modelo.

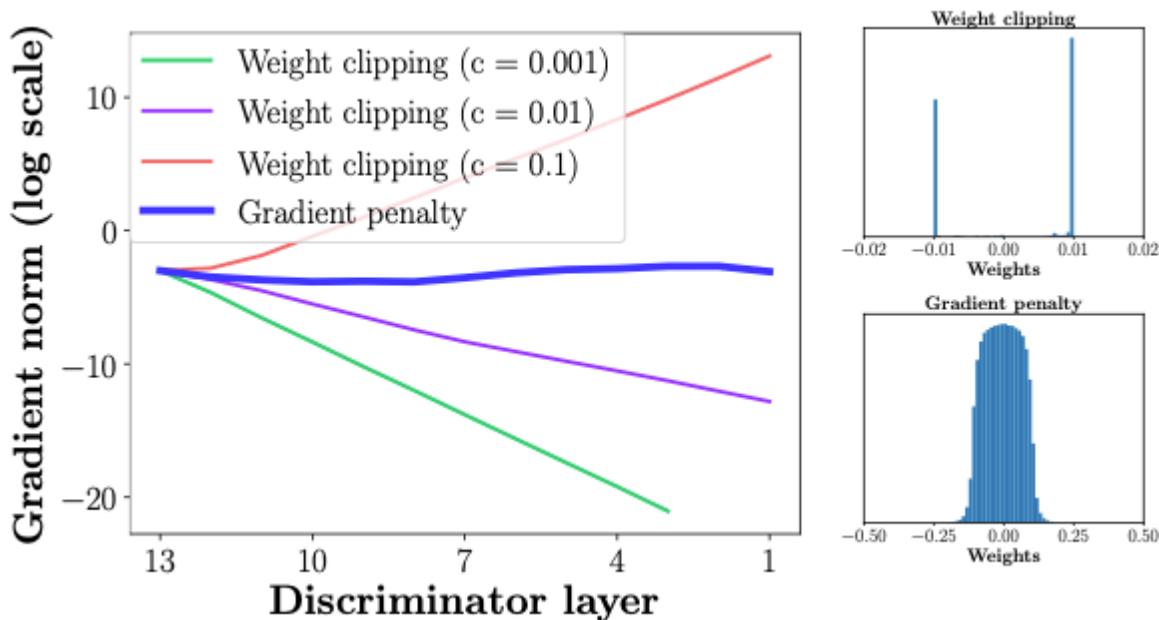


Figura 17: Ajuste de pesos y penalización de gradientes [33]

Podemos observar la diferencia entre el ajuste de pesos y la penalización del gradiente en la anterior gráfica, observamos que la penalización del gradiente se mantiene entre 0 y 1

Antes de aplicar la penalización a la función de pérdida del crítico, necesitamos de muestras interpoladas, estas juegan un papel importante para que se cumpla la restricción 1- Lipschitz. Las muestras (imágenes) interpoladas son una combinación lineal entre muestras:

$$\bar{x} = G(z)$$

$$\hat{x} = \alpha x + (1 - \alpha) \bar{x}$$

$x$  es una muestra de la distribución de los datos reales,  $\bar{x}$  de los datos generador y  $\alpha$  es un parámetro aleatorio comprendido entre [0 - 1]

Pongamos un ejemplo para unas muestras reales con valor 0  $x = 0$ ,  $\bar{x} = 5$ ,  $\alpha = 0.5$

$$\hat{x} = 0.5 \cdot 0 + (1 - 0.5) \cdot 5 = 2.5$$

Nos da una muestra interpolada de 2.5 una distribución media de ambas muestras la generada y la real como podemos ver en el siguiente esquema la mezcla de distintas muestras reales con generadas

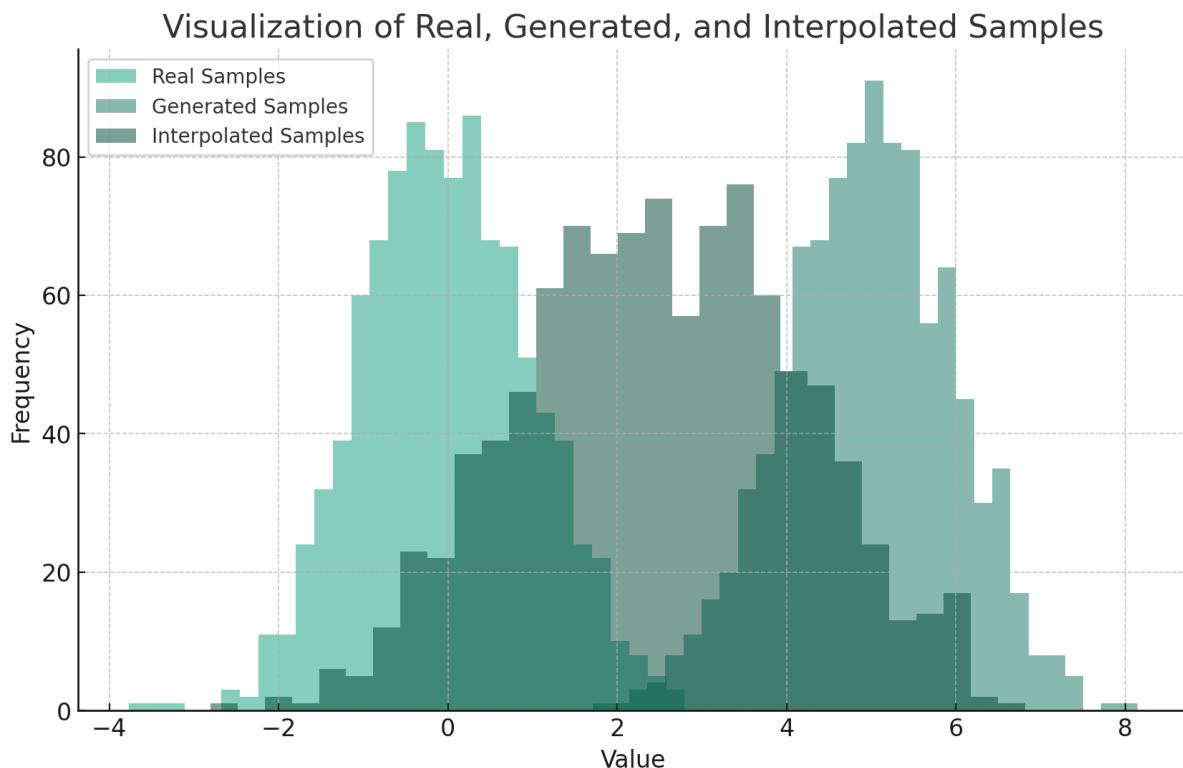


Figura 18: Gráfica de funcionamiento de muestras interpoladas

Una vez comprendidas y realizadas las muestras interpoladas aplicamos la penalización del gradiente:

$$\lambda \cdot (\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2 [10]$$

- $\lambda$  es el hiperparametro(coeficiente) que determina la influencia del GP en la fórmula

- $\Delta_x D(\hat{x})$  es el gradiente respecto a las muestras interpoladas

- $\left\| \Delta_x D(\hat{x}) \right\|_2$  es la norma L2 o norma euclíadiana respecto a el gradiente.

---

Norma euclíadiana:

$$\|\nabla_{\hat{x}} D(\hat{x})\|_2 = \sqrt{\left(\frac{\partial D}{\partial x_1}\right)^2 + \left(\frac{\partial D}{\partial x_2}\right)^2 + \dots + \left(\frac{\partial D}{\partial x_n}\right)^2}$$

$\left\| \Delta_x D(\hat{x}) \right\|_2$  indica la longitud del vector gradiente en función del discriminador evaluado con las muestras interpoladas  $\hat{x}$

---

Se resta 1 ya que si la norma  $\left\| \Delta_x D(\hat{x}) \right\|_2$  es mayor o menor que 1 la penalización sea positiva afectando así la función de pérdida del crítico para seguir cumpliendo la restricción de Lipschitz.

La penalización del gradiente se aplica a la función de pérdida del crítico(discriminador):

$$\max_d (E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)] + \lambda E_{\hat{x} \sim p_{\hat{x}}}(\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2)$$

Así funciona una red generativa adversarial wasserstein :

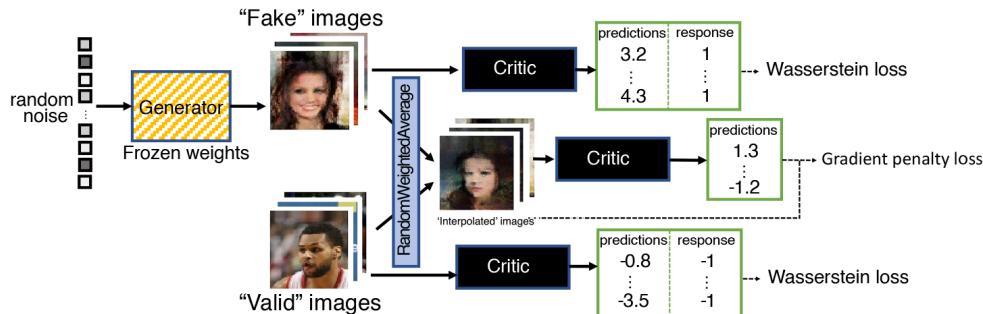


Figura 19: Ejemplo Red Generativa Adversarial Wasserstein

Explicación basada en el estudio [10].

## 5.3 PROGRESSIVE GANS

### 5.3.1 Introducción

Esta red Generativa adversarial demostró ser un cambio revolucionario en las GANS, consiguiendo generar unos datos generados de alta calidad y realistas como estos [22]



Figura 20: Imágenes generadas por ProGan

A diferencia de las anteriores GANS podemos ver que esta es realmente más avanzada que el resto de redes generativas adversariales, pero ¿cómo funciona?

Los principales cambios que trajo esta arquitectura diseñada por [nombre de paper de nvidia] de NVIDIA fueron los siguientes:

### 5.3.2 Crecimiento Progresivo

Las PRO-GAN progressive generative adversarial networks, como su propio nombre indica son redes generativas progresivas, las entrenamos progresivamente.

En vez de actuar directamente con las resoluciones y con las imágenes reales, bajamos la resolución de las imágenes, haciendo que el generador no trate de generar directamente las imágenes de alta calidad sino que trate de generar primero imágenes indistinguibles de baja

resolución a los datos reales a baja resolución, una vez el generador y el crítico se igualen en el entrenamiento de dicha resolución, cambiamos de resolución y realizamos el mismo proceso, así hasta llegar a la resolución final deseada.

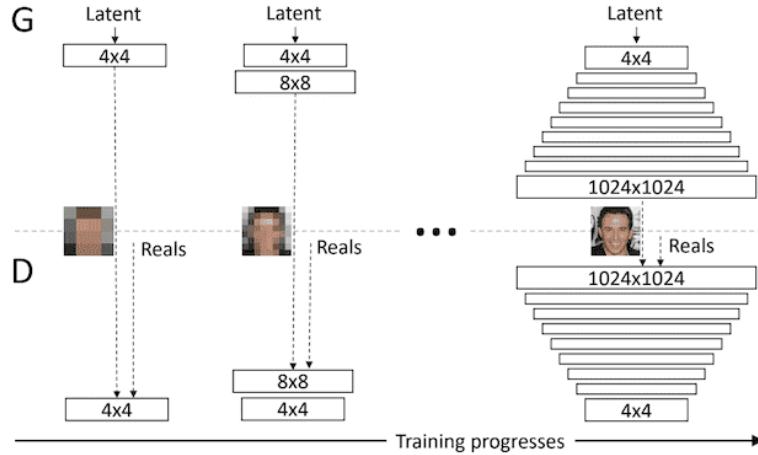


Figura 21: Generador y Discriminador en ProGAN [31]

Como vemos en la imagen primero entrenamos con la imagen de resolución 4x4 y vamos escalando cuando el generador sea indistinguible por el discriminador.

### 5.3.3 Transicion

Para evitar saltos abruptos entre resoluciones, se realiza tanto en el generador como en el discriminador, un proceso de transición(fade in) donde se mezclan las capas previas de la anterior resolución con las nuevas capas de nueva resolución resultantes de los filtros. Este proceso se lleva a cabo a través de la interpolación mediante el parámetro  $\alpha$ , el cual va aumentando mediante el entrenamiento progresivamente de 0 a 1, este “fade-in”o interpolación se lleva a cabo mediante la siguiente operación:

$$\text{Interpolated output} = \text{OldLayer} * (1-\alpha) + \text{NewLayer} * \alpha$$

Incrementamos 0 a 1 gradualmente  $\alpha$  para permitir progresivamente durante la interpolación mas capas nuevas salida de to/from RGB y menos capas to/fromRGB antiguas

Continuamos realizando el mismo proceso de transición hasta que todas las capas de resolución anterior a la nueva se hayan interpolado con las nuevas:

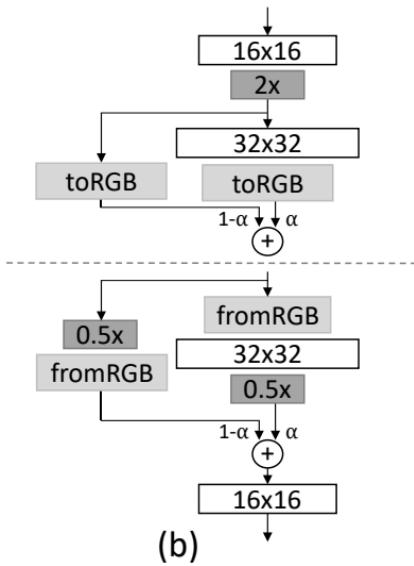


Figura 22: Transicion [32]

Una vez realizado ya el proceso de transición, estabilizamos tanto el generador como el discriminador:

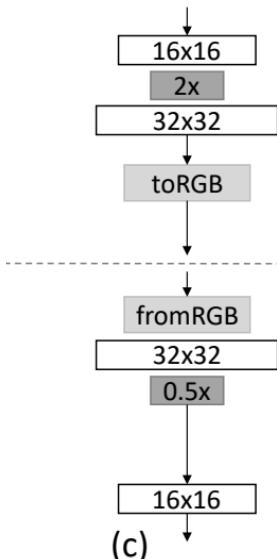


Figura 23: Estabilización [32]

El proceso de estabilización asegura por ultimo que el modelo pueda generar y discernir eficazmente imágenes en la nueva resolución antes de avanzar a la siguiente resolución.

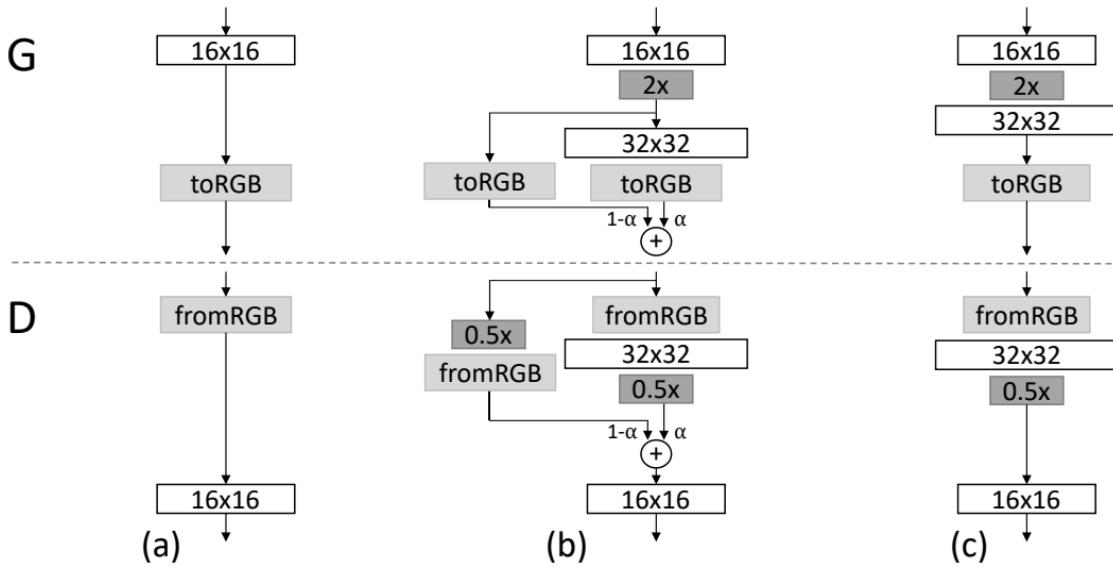


Figura 24: Esquema de Generador y Discriminador ProGan [32]

A parte del entrenamiento progresivo, se introducen nuevos conceptos como la normalización por pixel, mini batch std (desviación típica de pequeños lotes), tasa de aprendizaje ecualizada,

### 5.3.4 MiniBatchSTD

Para resolver el problema de variación o modo colapso en las imágenes del Generador, se aplica una capa minibatchSTD a el discriminador. Esta capa añade la desviación estándar de los valores de los mapas de características de la media de todos los píxeles. para aumentar la variabilidad entre las imágenes generadas dentro de un mini lote, haciendo que sea más difícil para el discriminador rechazar las imágenes generadas basándose en la falta de diversidad lo que conlleva a que el generador genere imágenes más variadas.

### 5.3.5 PixelWise Normalization

En vez de utilizar normalización por lotes de datos en el generador utilizaremos normalización por píxeles. Este tipo de Normalización normaliza cada valor de activación de cada pixel en un mapa de características/ canal evitando que la escala de los valores de activación escale sin control.

La fórmula se calcula:

$$b_{xy} = \frac{a_{xy}}{\sqrt{\frac{1}{N} \sum_{j=0}^{N-1} (a_{x,y}^j)^2 + \epsilon}}$$

Media de los pixeles de todos los canales (mapas de características)=  $\sqrt{\frac{1}{N} \sum_{j=0}^{N-1} (a_{x,y}^j)^2 + \epsilon}$

Valor de los pixeles=  $a_{x,y}^j$

número de canales/mapas de características = N

Pixel resultante en posición x,y del canal=  $b_{xy}$

$\epsilon$  valor añadido para evitar división por 0

Esta fórmula ajusta la activación de cada píxel de manera que la norma promedio de las activaciones a través de los canales sea 1. Este proceso se repite para cada píxel en cada mapa de características generado por el generador. Esto nos ayuda a mantener el entrenamiento bajo control y mejorar la estabilidad de este.

### 5.3.6 EQLR / Tasa de Aprendizaje Ecualizada

La tasa de aprendizaje igualada o ecualizada, trata de mantener uniforme la magnitud de las actualizaciones de los pesos en todas las capas de la red generativa adversarial.

La EQLR ajusta los pesos de la red basando en la desviación estándar de los pesos en la capa consiguiendo así que la actualización de los pesos sea independiente al tamaño de la capa, esto es extremadamente útil en progresos ya que vamos añadiendo capas progresivamente

$$\text{Peso ajustado} = \text{Peso} \cdot \sqrt{\frac{2}{k*k}}$$

$k*k$  = Tamaño del kernel/filtro

$$\sqrt{\frac{2}{k^*k}} = \text{Desviación estándar de los pesos}$$

Dicha fórmula nos asegura que todas las capas de la red aprenderán con una tasa de aprendizaje igualada independientemente del tamaño de entrada de las capas.

Explicación basada en [11],[12], [23]

*Podemos Observar que a diferencia de otros tipos de GANS mencionamos la función de pérdida esto es porque es de libre uso, se puede utilizar cualquiera que consideremos efectiva, en mi opinión siempre recomendaría aplicar la función de pérdida de WGAN-GP ya que hemos visto que es la mas precisa en el desarrollo*

## 6.ENTORNOS

Antes de empezar a programar hemos de configurar el entorno, para ello utilizaremos Pytorch, Keras, Cuda, Visual Studio Code, Google Collab.

### 6.1 PYTORCH

Antes de empezar el código tenemos que configurar nuestro entorno virtual para poder trabajar con pytorch, librerías de pytorch y con la GPU.

Pytorch, es una de las bibliotecas mas utilizadas en código de IA, fue desarrollada por facebook, esta biblioteca esta basada en un software que se utiliza para desarrollar redes neuronales, mezclando una API de alto nivel basada en python con la biblioteca Torch de machine Learning. Un claro ejemplo de su potencial son las redes generativas adversariales

En esta implementación técnica de una deep convolutional gan, utilizamos la versión de pytorch 2.2.1

## 6.2 CUDA

Para ejecutar los entrenamientos localmente en la GPU de nuestro ordenador, y así evitar tiempos de entrenamiento neuronal más que exagerados, es necesario que si tenemos una gráfica nvidia instalamos, el driver CUDA.

CUDA permite a nuestra gpu una ejecución paralela, organizando una serie de núcleos de procesamiento paralelo que permiten ejecutar miles de hilos simultáneamente, potenciando el rendimiento de nuestra tarjeta grafica para cálculos complejo. Esto nos viene como anillo al dedo para entrenar nuestros modelos, que requieren una eficiencia computacional muy alta.

Pytorch ofrece soporte nativo para CUDA, dándonos una perfecta combinación de software para ejecutar redes neuronales profundas.

En este trabajo utilizaremos pytorch +Cuda +Vscode para la implementación técnica de la red generativa adversarial convolucionalmente profunda (DCGAN) y la implementación de redes generativas adversariales progresivas (PRO-GANS). La versión que utilizaremos son las siguientes

```
>>> import torch
>>> torch.__version__
'2.2.1+cu121'
```

*Figura 25: Version Cuda*

Pytorch 2.2.1 y cuda12.1.

## 6.3 VS CODE / GOOGLE COLAB

El editor de código que emplearemos para la implementación de la DCGAN así como la PROGAN, será vscode. La razón por la que elegido Vs Code es por la costumbre de realizar código de python en él, y el soporte de python en Vscode, funciona considerablemente bien. Vscode se ejecutara en local en nuestra computadora.

Sin embargo, también utilizaremos GoogleCollab, google collab esta específicamente diseñado para programas algoritmos de machine learning, deep learning, trabajar y manejar

grandes volúmenes de datos. La gran principal ventaja respecto a VScode es que todo el código que ejecutemos se ejecutará en la nube, en la gpu de un servidor de google remoto, despreocupandonos así de nuestros recursos. En este trabajo utilizaremos la gpu A100 de nvidia disponible en google collab para la implementación técnica de la red generativa adversarial Wasserstein con penalización de gradiente(WGAN-GP) y mas adelante para la mejora de modelos CNN.

## 6.4 KERAS +TENSORFLOW

En muchos modelos programados durante este proyecto, utilizaremos el entorno de Keras +Tensorflow + Google Colab, esto nos va asegurar un entorno robusto y eficiente para desarrollar los modelos

Tensorflow al igual que torch es una librería de machine learning desarrollada por google para satisfacer las necesidades a partir de redes neuronales artificiales. Tensorflow facilita la creación e implementación de modelos de aprendizaje automático permitiéndonos automatizar cientos de procesos. Más adelante en el código veremos una de las principales ventajas que nos trae consigo tensorflow, que son los tensores.

Keras es una API basada en tensorflow, esta API hace mucho mas productivo el diseño de la arquitectura de modelos de aprendizaje profundo, olvidándonos así de operaciones de bajo nivel no tan necesarias. keras cuenta con módulos de gran ayuda como las capas, distintos optimizadores como los vistos anteriormente así como distintas funciones de pérdida. Esto lo hace perfecto para la investigación avanzada como la realizada en este proyecto

## 6.5 TENSORBOARD

TensorBoard es una herramienta de visualización integrada para tensorflow, es una herramienta crucial y lo usaremos en todos los modelos de generación de ejemplos adversariales (GANS), tensorboard nos permitirá analizar y observar como está evolucionando nuestro modelo durante el entrenamiento dándonos gráficas, histogramas, distribuciones muy útiles como la pérdida del generador o la pérdida del discriminador

durante el entrenamiento, así como proyecciones de la actualización de lotes de las imágenes mientras se van generando en cada época, etc. Tensorboard es muy flexible y somos nosotros quienes indicamos qué queremos ver durante el entrenamiento y cómo nos lo va a enseñar. Tensorboard un ejemplo de un tensorboard puede ser el siguiente.



Figura 26: Ejemplo Visualización Tensorboard

Tensorboard siempre accederá a los logs de nuestro código para acceder a los datos del entrenamiento a tiempo real. Para iniciar lo siempre será en consola: ***tensorboard --logdir= #directorio donde se encuentren nuestros logs***

Por defecto se nos abrirá la visualización en el puerto 6006 de nuestro PC

## 7. IMPLEMENTACIÓN TÉCNICA DE LAS GANS

Antes de comenzar subrayar que los conjuntos de datos utilizados para la demostración técnica de cómo funcionan las gans no serán los mismos, que los conjuntos de datos que usaremos posteriormente para los casos de mejora de modelos de IA.

### 7.1 IMPLEMENTACIÓN TÉCNICA DCGAN

Como indicado anteriormente este es el tipo de GAN más básico y por tanto el menos preciso en generación de imágenes, sin embargo necesitamos conocer técnicamente cómo funciona el código para conocer los siguientes tipos de redes como WGAN-GP que PRO-GAN que son mucho más sofisticados. El DCGan que vamos a desarrollar como ejemplo lo hemos basado en un entorno Vscode+ pytorch +Cuda +Tensorboard

### 7.1.1 Dataset

Al ser un GAN básico utilizaremos, un dataset de imágenes pequeñas y simples, para ver su potencial, para ello elegiremos el dataset MNIST [24]. Este dataset es conocido mundialmente y si has trasteado con IA seguro que en tus primeros pasos has utilizado este dataset.

El conjunto de datos consiste en 60.000 imágenes de dígitos comprendidos entre 0-10 escritos a mano. Las imágenes son de 28x28 y cada una tiene solo un canal de una escala de gris, es decir blanco y negro.

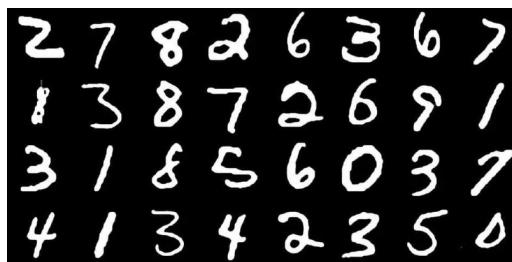


Figura 27: Mnist Dataset

Este conjunto de datos está organizado en 4 archivos

- train-images-idx3-ubyte.gz: conjunto de imágenes de entrenamiento (9912422 bytes)
- train-labels-idx1-ubyte.gz: etiquetas de entrenamiento (28881 bytes)
- t10k-images-idx3-ubyte.gz: conjunto de imágenes para el testeo (1648877 bytes)
- t10k-labels-idx1-ubyte.gz: etiquetas para el testeo (4542 bytes)

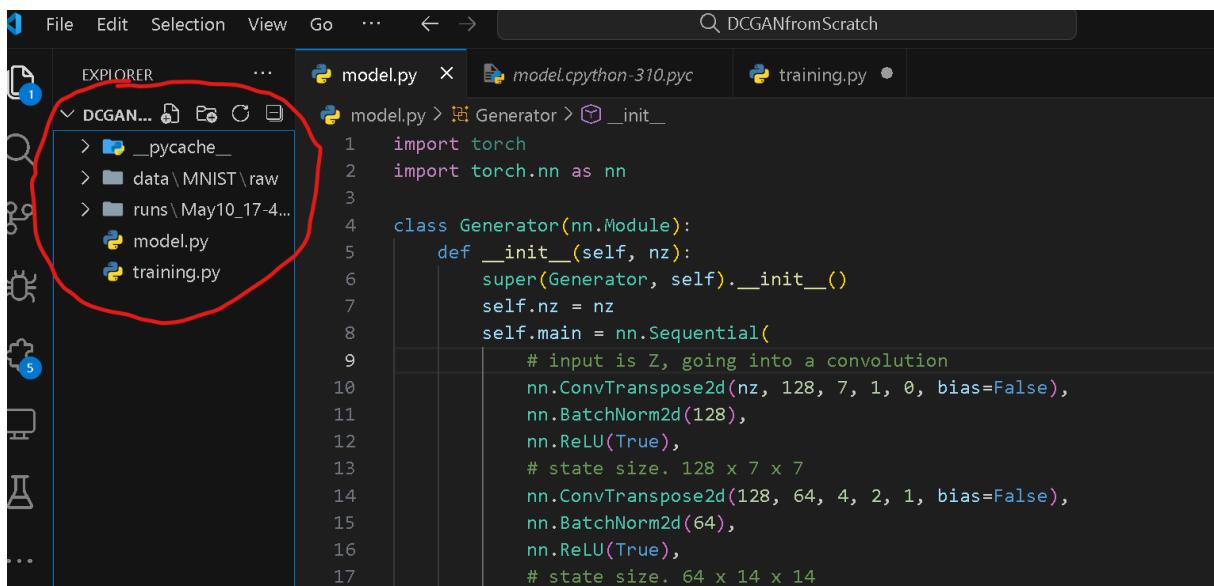
En nuestro caso no utilizaremos las etiquetas ni los conjuntos de datos de testeos, estos son más utilizados en modelos CNN que veremos más adelante cuando tengamos que mejorar estos modelos.

La red generativa convolucionalmente profunda (DCGAN) utilizada en este caso ha sido entrenada con el conjunto de imágenes de entrenamiento.

Dicho esto empecemos con el código, mientras transcurre la memoria ire explicando paso a paso cómo funciona este código.

### 7.1.2 Estructura General

Bien Hagamos una captura general del código que hemos desarrollado previamente para este proyecto.



```

File Edit Selection View Go ... < > Q DCGANfromScratch
EXPLORER ...
DCGANfromScratch ...
model.py X model.cpython-310.pyc
model.py Generator > __init__
1 import torch
2 import torch.nn as nn
3
4 class Generator(nn.Module):
5     def __init__(self, nz):
6         super(Generator, self).__init__()
7         self.nz = nz
8         self.main = nn.Sequential(
9             # input is Z, going into a convolution
10            nn.ConvTranspose2d(nz, 128, 7, 1, 0, bias=False),
11            nn.BatchNorm2d(128),
12            nn.ReLU(True),
13            # state size. 128 x 7 x 7
14            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
15            nn.BatchNorm2d(64),
16            nn.ReLU(True),
17            # state size. 64 x 14 x 14

```

Figura 28: Estructura General Código DcGan

Vemos que el entorno está formado por la carpeta data, la carpeta runs, los modelos en *model.py*, y el código de entrenamiento en *training.py*.

Esta estructura es y va a ser común en siguientes implementaciones en este proyecto con algún que otro cambio. La carpeta data, como su nombre indica almacenará todos los datos mnist que hemos especificado antes, en la carpeta runs se guardarán los eventos que guardaran los datos de los logs mientras se ejecuta *training.py*, gracias a Tensorboard

### 7.1.3 Generador

En los modelos, *model.py*, escribimos la arquitectura y funcionamiento que tendrán nuestro modelo generativo (Generador) y nuestro modelo discriminativo (Discriminador).

Empecemos por los imports, luego iremos al generador, y finalmente a el discriminador

```
import torch
import torch.nn as nn
```

Figura 29: Import Generador DcGan

Importamos *torch* y *torch.nn*. Torch.nn es esencial, ya que contiene las definiciones para las capas convolucionales, funciones de activación, funciones de pérdida, en resumen todas las variables de las que necesita un modelo generativo como un discriminativo, discutidas previamente. A continuación el Generador

```
class Generator(nn.Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(nz, 128, 7, 1, 0, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            # state size. 128 x 7 x 7
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            # state size. 64 x 14 x 14
            nn.ConvTranspose2d(64, 1, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. 1 x 28 x 28
        )

    def forward(self, input):
        return self.main(input)
```

Figura 30: Código Generador DcGan.

Antes de nada, recordemos las imágenes a tratar son mnist y tienen un tamaño de 28x28 y solamente 1 canal ya que son en una sola escala de grises (blanco y negro).

Creamos la clase, programamos el constructor de nuestro generador:

`__init__(self,nz)` En este método inicializo el generador con el parámetro nz, en este caso nz es la dimensión del espacio latente, o otras palabras más familiares el tamaño del vector ruido aleatorio que se alimenta a todo generado (1x1)

`super(Generator,self).__init__()` Para que el generador funcione como un módulo de red neuronal heredamos de nn.module y por tanto habrá que llamar a el constructor de la superclase

`self.main=nn.Sequential` aquí indicamos el tipo de módulo que vamos a utilizar, en este caso *sequential*, es decir todas las convoluciones funciones de activación etc, se van a ejecutar de modo secuencial en el código.

Volviendo a la teoría previa, simplemente leyendo el código que hemos desarrollado ya cualquiera se puede imaginar lo que está haciendo, sin embargo a continuación lo voy a clarificar en caso de duda.

El generador aplica secuencialmente capas de deconvolución o convolución transpuesta. Por ejemplo, la primera deconvolución, `nn.ConvTranspose2d(nz, 128, 7, 1, 0, bias=False)`, toma el vector de ruido `nz` y lo transforma en un tensor de dimensiones `[128, 7, 7]`. Aquí, `7x7` corresponde al tamaño del filtro (kernel), con un stride de `1` y sin padding. Esto significa que la representación inicial en este punto tiene una dimensión de `7x7`.

Continuando con el proceso, la siguiente capa toma la imagen generada y aumenta la resolución del tensor a `[64, 14, 14]` (es decir, `14x14` para cada uno de los 64 canales), utilizando un filtro de `'4x4'`, un stride de 2 y un padding de 1. Finalmente, la última capa transforma la salida para producir un único `1` canal (blanco y negro) de una imagen generada de tamaño `28x28`.

Una pregunta que podría surgir es: ¿Por qué utilizamos tantos canales en las capas intermedias de la red, desde la primera hasta la última capa convolucional? Como se explicó en la sección 2.1, "Fundamentos Teóricos", en las redes neuronales es importante destacar que los "nodos", en este caso los canales, se expandan en número a través de las capas sucesivas. Esto es importante porque al aumentar el número de canales, la red puede capturar y procesar una mayor cantidad de información y características de los datos de entrada. A su vez, esta riqueza en la representación de datos facilita una retroalimentación más efectiva durante el proceso de backpropagation, permitiendo así que la red aprenda patrones más complejos y

sutilezas de los datos de entrenamiento. Esto no es solo aplicable a GANS sino a casi todos los tipos de deep learning(aprendizaje profundo)

La estabilización del entrenamiento es muy importante, así que analizaremos estas capas entre deconvoluciones.

***nn.BatchNorm2d()*** y ***nn.ReLU(True)***, en batchnorm recibimos los canales y los valores dentro del tensor son normalizados y posteriormente transformados por ReLU para introducir no linealidades

Por último la activación final ***nn.Tanh()***, escalando los valores del tensor o matriz entre [-1,1]

#### 7.1.4 Discriminador

Hemos analizado el generador, el discriminador es simple es un efecto espejo del generador, así que ahora es mas sencillo comprenderlo, vemos que ahora en vez de usar las capas deconvolucionales, utilizó las convolucionales ya que queremos una salida 1x1x1 discriminativa para que el discriminador clasifica la imagen según sus características extraídas (Esto es muy similar a como funcionan los modelos CNN clasificadores en general )

Vemos como la imagen sea generada o no pasara de ***1x28x28, (28/2 por el stride=2) → 64x14x14 → 128x1 x1→ 1x1x1***. Utilizamos por ultimo la activación ***Sigmoid()***, para convertir el valor escalar entre un valor de 0 y 1, para poder clasificar la imagen como 0 fake o 1 real

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            #1x28x28
            nn.Conv2d(1, 64, 4, 2, 1, bias=False),#
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(64, 128, 4, 2, 1, bias=False),#64x14x14
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 1, 7, 1, 0, bias=False),#128x7x7
            nn.Sigmoid()#1x1x1
        )

        def forward(self, input):
            return self.main(input).view(-1, 1).squeeze(1)

```

Figura 31: Código Discriminador DcGan

**nn.LeakyRelu(0.2, inplace=true)**, aplica un pequeño gradiente 0.2 a aquellas neuronas que no produzcan ninguna salida “neuronas muertas” y **BatchNorm()**, normalizando los valores regulando y acelerando el entrenamiento.

Si eres observador/ra, no utilizo BatchNorm después de la primera capa esto es porque después de la primera capa no se suelen haber extraído suficientes características tras pasar el filtro, por tanto puede que normalizar estas pocas caracerísticas puede afectar a la estabilidad de el entrenamiento del discriminador.

### 7.1.5 Hiperparámetros e Imports

En training.py tenemos el código de como vamos a entrenar ambos modelos discriminador y generador.

Primero importamos de la libreria torch los módulos que vamos a utilizar

```

py training.py > ...
1 import torch
2 import torch.optim as optim
3 from torch.utils.data import DataLoader
4 from torchvision import datasets, transforms
5 from torchvision.utils import make_grid
▶ Launch TensorBoard Session
6 from torch.utils.tensorboard import SummaryWriter
7 from model import Generator, Discriminator
8 import torch.nn as nn

```

Figura 32: Import De módulos de Torch DcGan

Importamos ***torch***, importamos ***torch.optim*** que son los optimizadores, en este caso hemos utilizado ADAM,. ***DataLoader***, es una clase que utilizamos para manejar el conjunto de datos o “subir el conjunto de datos”.***Datasets***, es un modulo que tiene torch, donde proporciona acceso a los datasets mas famosos utilizados para entrenar IA como en nuestro caso MNIST. ***Transforms*** lo utilizamos para transformar los datos y normalizarlos, lo veremos mas adelante en el codigo.

***make\_grid*** como su nombre indica toma un conjunto de imágenes y las almacena en un “cuadro” visualmente, ***make\_grid*** lo utilizaremos para visualizar las imágenes generadas mientras se van entrenando con ***tensorboard***

***summary\_writer***, lo utilizamos para escribir los datos a tensorboard y podamos visualizarlos en tensorboard. Por ultimo, importamos de ***model.py*** el generador y el discriminador y ***nn*** como mencionado anteriormente en los modelos. Adelante analizaremos los hiperparámetros.

```

# Hyperparametros
batch_size = 128
epochs = 50
lr = 0.0002
beta1 = 0.5
nz = 100 # Tamaño del vector sonido

```

*Figura 33: Valores de HiperParametros DcGan*

Los hiperparámetros, es la pieza donde se sostiene el entrenamiento y el resultado que vamos a tener. La precisión de las imágenes generadas por GANS, y rendimiento del entrenamiento se ven afectadas severamente por los valores que introduzcamos en los hiperparametros, tienen una importancia significativa. Si ejecutamos una GAN y al cabo del tiempo no conseguimos los resultados esperados, primero habrá que ajustar los hiperparámetros para ver cómo cambia el entrenamiento de los modelos.

A continuación , se van a analizar:

**batch\_size :** Batch en castellano, lote, en el tamaño de lote indicamos la cantidad de datos o ejemplos se van a procesar antes de que el modelo generador o discriminador actualice los parámetros. Cuanto mayor sea el batch size más estables serán los gradientes, sin embargo consumirá mucha más memoria y tiempo. Si tenemos un batch size pequeño evita potencialmente el riesgo de convergencia de mínimos, mencionado en el punto **3.2.3**, sin embargo el rendimiento de los resultados y entrenamiento sera mucho menos preciso y estable.. Los tamaños de lote, suelen ser de 64 en e es decir **64 o de 128, o 192, 256 etc**. En este caso he elegido un tamaño de lote **128**.

**epochs:** Epochs, o épocas en castellano m son fundamentales, indica cuántas veces va a ser entrenado todo el conjunto de datos, cuanta más épocas más características se extraen y mejores resultados, pero claro mas uso de memoria y tiempo. Si el dataset que utilizamos no tiene muchos datos o imágenes, no es lo suficientemente grande, no es necesario introducir un número exagerado de epochs. En este caso he utilizado 50 que son mas que necesarios para obtener resultados precisos con MNIST.

**LR :** La tasa de aprendizaje explicada en el punto **3.7.2**, escogemos una de **0.0002** asegurándonos que no es ni muy baja ni muy alta.

**Beta1:** Es el parámetro de  $\beta_1$  para el optimizador ADAM, la importancia de este parámetro la podemos ver en el punto 3.7.3, controla la tasa de decaimiento de la media de los gradientes.

**nz:** la dimensión del ruido

El objetivo de las Gans es el equilibrio entre el generador y el discriminador, siendo muy delicado, los hiperparámetros tienen un impacto significativo en la estabilidad del entrenamiento

Antes de continuar y llegar a el bucle/loop de entrenamiento explicaremos mas el codigo

### 7.1.6 Preparacion de datos y Definición de Componentes

```

17 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
18
19 # Cargamos MNIST dataset
20 transform = transforms.Compose([
21     transforms.ToTensor(),
22     transforms.Normalize((0.5,), (0.5,))
23 ])
24 train_data = datasets.MNIST(root='./data', train=True, download=True, transform=transfo
25 train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
26
27 # Inicializamos los modelos
28 netG = Generator(nz).to(device)
29 netD = Discriminator().to(device)
30
31 # Optimizadores
32 optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
33 optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
34
35 # Loss function
36 criterion = nn.BCELoss()
37

```

Figura 34: Preparación de datos y definición de componentes codigo DcGan

En la linea 17 indicamos que el código utilize la gpu para entrenar los modelos mediante CUDA, si este no esta instalado entonces mediante el procesador

De la linea 19 a la 25 cargamos y normalizamos los datos, convertimos las imágenes en tensores para Pytorch, y normalizamos con *Normalize* una media de 0.5 y una desviación estándar de 0.5, cambiando los valores de cada píxel de [0,1] a [-1,1 ]. Con el módulo ***datasets*** nos descargamos y extraemos la data en el directorio data. Por último pero no menos importante, con el módulo ***DataLoader***, organizamos los datos en lotes y los mezclamos(*shuffle=true*) para una mejor generalización de los datos.

De la linea 26 a la 27, inicializamos creando instancias del generador y el discriminador (***ng,nz***)definidos en model.py, los mandamos a la GPU(*device*). En la línea 32 y 33 inicializamos los optimizadores dl discriminador como del generador, con la tasa de

aprendizaje, y los parámetros  $\beta_1$  definido anteriormente y  $\beta_2 = 0.999$ . La línea 36 es muy importante, el criterio(criterio), aquí definimos qué función de pérdida van a usar nuestros modelos, como estamos desarrollando una DCGAN básica, la función de pérdida será a ***Binary Cross Entropy***, (entropía de cruce binario). Esta fórmula funciona tal especificada en el punto **3.2.3**

Ya hemos cubierto la configuración del entorno, la preparación de datos, los hiper parámetros y la definición de componentes esenciales para el modelo y su optimización. Ahora profundizaremos en cómo funcionará el bucle de entrenamiento

### 7.1.7 Entrenamiento

El bucle de entrenamiento es muy extenso por esta razón lo vamos a desglosar en dos partes en este documento.

```

42 ✓ for epoch in range(epochs):
43   ✓   for i, (real_images, _) in enumerate(train_loader):
44     # Update/Actualizamos discriminador
45     netD.zero_grad()
46     real_images = real_images.to(device)
47     b_size = real_images.size(0)
48     label = torch.full((b_size,), 1, dtype=torch.float, device=device)
49     output = netD(real_images)
50     errD_real = criterion(output, label)
51     errD_real.backward()
52     D_x = output.mean().item()

53     # Generate/Generamos las imágenes "falsas"
54     noise = torch.randn(b_size, nz, 1, 1, device=device)
55     fake_images = netG(noise)
56     label.fill_(0)
57     output = netD(fake_images.detach())
58     errD_fake = criterion(output, label)
59     errD_fake.backward()
60     D_G_z1 = output.mean().item()
61     errD = errD_real + errD_fake
62     optimizerD.step()

63     # Actualizamos generador
64     netG.zero_grad()
65     label.fill_(1)
66     output = netD(fake_images)
67     errG = criterion(output, label)
68     errG.backward()
69     D_G_z2 = output.mean().item()
70     optimizerG.step()
71
72
73

```

Figura 35:Código de entrenamiento DcGan

Iniciamos el ciclo de entrenamiento en en la línea 42 iteramos sobre el número de épocas, y en cada época con el for anidado iteramos sobre el conjunto de imágenes reales definido previamente por lotes,en este caso **128**,en el ***train\_loader***.

### Actualización del Discriminador

```

42   for epoch in range(epochs):
43       for i, (real_images, _) in enumerate(train_loader):
44           # Update/Actualizamos discriminador
45           netD.zero_grad()
46           real_images = real_images.to(device)
47           b_size = real_images.size(0)
48           label = torch.full((b_size,), 1, dtype=torch.float, device=device)
49           output = netD(real_images)
50           errD_real = criterion(output, label)
51           errD_real.backward()
52           D_x = output.mean().item()
53

```

Figura 36: Actualización del Discriminador código DcGan

De la línea 45 a la 52 actualizaremos los parámetros y gradientes del discriminador, **net.zero\_grad()** reinicia los gradientes del discriminador. con **real\_images.to(device)** movemos las imágenes a la GPU, ajustamos la variable **b size** con el tamaño del lote de imágenes.

En la línea 48 creamos un tensor **label** con unos (1) indicando que las imágenes son reales, recordemos 1 real, 0 fake. Mandamos las imágenes reales a el modelo del discriminador para que las evalúe **output=and(real images)**. Si lo relacionamos con la teoría explicada durante el proyecto **nerD(real images)** sería equivalente a **(D(x))**.

**Con error real = criterio (output,label)** calculamos la pérdida mediante la función de pérdida BCE comparando **D(x)(salida del discriminador evaluando datos reales )** con las etiquetas genuinas las verdaderas.

En la línea 51 En caso de error propagamos el error hacia atrás mediante la función **backward()**, esto se le llama **backpropagation** mencionado en los fundamentos del deep learning punto 2.2

Calculamos la media de las salidas/evaluaciones del discriminado **output.mean(),item()** para las imágenes reales y lo guardamos en **D\_X**. D

### Generación de imágenes

```

54     # Generate/Generamos las imágenes "falsas"
55     noise = torch.randn(b_size, nz, 1, 1, device=device)
56     fake_images = netG(noise)
57     label.fill_(0)
58     output = netD(fake_images.detach())
59     errD_fake = criterion(output, label)
60     errD_fake.backward()
61     D_G_z1 = output.mean().item()
62     errD = errD_real + errD_fake
63     optimizerD.step()

```

Generamos un tensor del ruido aleatorio ***noise*** nz=100, de 1 canal y lo mandamos a la GPU, pasamos el ruido a el generador de ***model.py G(z)***,en mi código ***netG(noise)***,para que genere imágenes falsas a partir del ruido.Guardamos las imágenes ***fake images***

En la línea 57 ***label.fill\_(0)*** Rellenamos el tensor label previamente instalado de ceros indicando que las imágenes son falsas. **Importante comprender, el tensor label lo utilizamos solamente para comparar el output del discriminador respecto a el input de este(imágenes Generadas o Reales).**

En la línea 58 guardamos en el output, las predicciones o evaluaciones del Discriminador respecto las imágenes del generador, esto sería en las fórmulas que hemos visto anteriormente ***D(G(z))***,para comprenderlo mejor en caso de no entenderlo sería ***netD(netG(noise).detach())***..

Utilizamos ***detach*** porque detach asegura que los pesos del Generador no se ven afectados cada vez que el discriminador se actualiza

Ahora como anteriormente para las imágenes reales, calculamos en la línea 59 el error de las predicciones del discriminador respecto a las imágenes generadas, y en la posterior línea realizas el proceso de backpropagation. propagando el error hacia atrás.

Calculamos otra vez la media las salidas y lo almacenamos en ***D(G(z))***. En la línea 62 combinamos ambas pérdidas, ***err real*** para las predicciones del discriminador sobre datos y ***errD\_fake*** para las predicciones de discriminador sobre las imágenes generadas, dando lugar a la función de pérdida del discriminador ***errD***.

Por último aplicamos el optimizador ADAM para el discriminador, previamente instalado en la línea 32, para optimizar la capacidad de distinción del Discriminador.

### Actualizar Generador

```

65     # Actualizamos generador
66     netG.zero_grad()
67     label.fill_(1)
68     output = netD(fake_images)
69     errG = criterion(output, label)
70     errG.backward()
71     D_G_z2 = output.mean().item()
72     optimizerG.step()
73

```

Figura 37: Actualización Generador código DcGan

Parecido del discriminador limpiamos la información de los gradientes cada vez que lo actualizamos **netG.zero\_grad()**

Aquí cambian las cosas, el generador como es bien sabido, trata engañar al discriminador como sea posible, por esto vamos a guardar unos (1 real) en el tensor de la etiqueta, para posteriormente en la línea 69, al comparar el output en **criterion** de las predicciones del discriminador sobre las imágenes generadas, **D(G(z))**, si la salida del discriminador coincide con el label, indica que el discriminador se ha equivocado, lo que sera un exito para el generador, en caso contrario almacenaremos el error y lo prepararemos similar a el Discriminador. Realizas la media de las salidas, y optimizamos el generador.

Cada actualización será realizada al completar cada iteración del conjunto de datos, “segundo for anidado”.

### Visualización y registro de estadísticas

Ya establecido el entrenamiento, necesitamos algún código para visualizar las métricas de pérdida del generador y discriminador, así como las imágenes generadas en cada iteración durante el entrenamiento. Para ello introducimos la herramienta **Tensor Board** (punto 5.6)

Volvamos a el código:

```

75     if i % 100 == 0:
76         print(f'{epoch}/{epochs} [{i}/{len(train_loader)}] Loss_D: {errD.item():.4f} Loss_G: {errG.item():.4f} D(x): {D_x:.4f} D(G(z)): {D_G_z1:.4f} / {D_G_z2:.4f}')
77         writer.add_scalar('Loss/Discriminator', errD.item(), epoch * len(train_loader) + i)
78         writer.add_scalar('Loss/Generator', errG.item(), epoch * len(train_loader) + i)
79
80     # Log images
81     with torch.no_grad():
82         fake_images = netG(noise).detach().cpu()
83         img_grid_real = make_grid(real_images[:32], normalize=True)
84         img_grid_fake = make_grid(fake_images[:32], normalize=True)
85         writer.add_image('Real Images', img_grid_real, global_step=epoch * len(train_loader) + i)
86         writer.add_image('Generated Images', img_grid_fake, global_step=epoch * len(train_loader) + i)
87
88     # Cerramos TensorBoard writer
89     writer.close()

```

Figura 38: Código para el registro de las estadísticas DcGan

Seguimos anidados en el foro, vamos a visualizar las estadísticas y métricas cada 100 iteraciones de entrenamiento interno. ***if i %100==0***

Imprimimos en consola los epochs, la pérdida del discriminador (Loss D), la pérdida del generador (Loss G), y las evaluaciones del discriminador sobre imágenes reales ( $D(x)$ ) y falsas ( $D(G(z))$ ).

```
print(f'{epoch}/{epochs}][{i}/{len(train_loader)}]
```

```
Loss_D:{errD.item():.4f}
```

```
Loss_G:{errG.item():.4f}
```

```
D(x): {D_x:.4f}
```

```
D(G(z)): {D_G_z1:.4f} / {D_G_z2:.4f}'
```

En la línea 77 y 79, llamamos a la función de tensor board, **writer.addscalar** para registrar la pérdida del Discriminador y la del Generador.

En el registro de imágenes en TensorBoard

```
80
81     # Log images
82     with torch.no_grad():
83         fake_images = netG(noise).detach().cpu()
84         img_grid_real = make_grid(real_images[:32], normalize=True)
85         img_grid_fake = make_grid(fake_images[:32], normalize=True)
86         writer.add_image('Real Images', img_grid_real, global_step=epoch * len(train_loader) + i)
87         writer.add_image('Generated Images', img_grid_fake, global_step=epoch * len(train_loader) + i)
```

Figura 39: Código de logueo de imágenes en Tensorboard para DcGan

Primeramente realizamos el cálculo de gradientes, con la función ***torch.no\_grad()***, porque no necesitamos cálculo de ningún tipo de gradientes para visualizar imágenes.

En la línea 82 instanciamos las imágenes generadas para enviarlas a tensorboard, el uso de detach y cpu es bien importante, ya que queremos que la visualización no afecta para nada a los parámetros del entrenamiento. Lo mandamos a la cpu porque la GPU esta ocupada con los cálculos durante el entrenamiento y asi interferimos lo menos posible. Hacemos un gridlayout para mostrar las imágenes reales y las que se van generando. Hecho esto añadimos

a el escritor / writer para visualizar los gráficos e imágenes, estarán indexadas según global step=epoch \* len(train\_loader) + i)

Por último cerramos tensorboard una vez se hayan completado todas las épocas `writer.close()`.

### 7.1.8 Ejecucion

Ya todo listo ejecutaremos el código

Es importante cada vez que vayamos ejecutar el código borrar la carpeta runs, ya que contiene logs de anteriores entrenamientos, que pueden dificultar a tensorboard a la hora de visualizar la información

Ejecutamos Training.py

al ejecutar training.py vemos como se está ejecutando el entrenamiento mostrandonos el progreso en cada iteración

```
[0/50][ 200/469] Loss_D: 0.7666 Loss_G: 1.0685 D(x): 0.6711 D(G(z)): 0.2956 / 0.3515
[0/50][ 300/469] Loss_D: 0.7254 Loss_G: 2.0960 D(x): 0.8738 D(G(z)): 0.4323 / 0.1321
[0/50][ 400/469] Loss_D: 0.6052 Loss_G: 1.2789 D(x): 0.7083 D(G(z)): 0.2093 / 0.2925
[1/50][ 0/469] Loss_D: 0.6585 Loss_G: 2.0533 D(x): 0.8484 D(G(z)): 0.3733 / 0.1419
```

Figura 40: Progreso en cada iteración DcGan

Podemos ver como se enfrentan simultáneamente el generador y el discriminador potenciándose y aprendiendo mas y mas entre ellos.

Ahora vamos a visualizar Tensorboard, para ello ejecutamos en **otra terminal**

```
tensorboard --logdir runs
```

Figura 41: Tensorboard -logdir runs

Se nos abrirá el puerto 6006 en localhost y podremos visualizar el entrenamiento gráficamente a tiempo real

```
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.8.0 at http://localhost:6006/ (Press CTRL+C to quit)
■
```

Figura 42: Ejecucion servidor Tensorboard en localhost

### Evolución de las funciones de pérdida

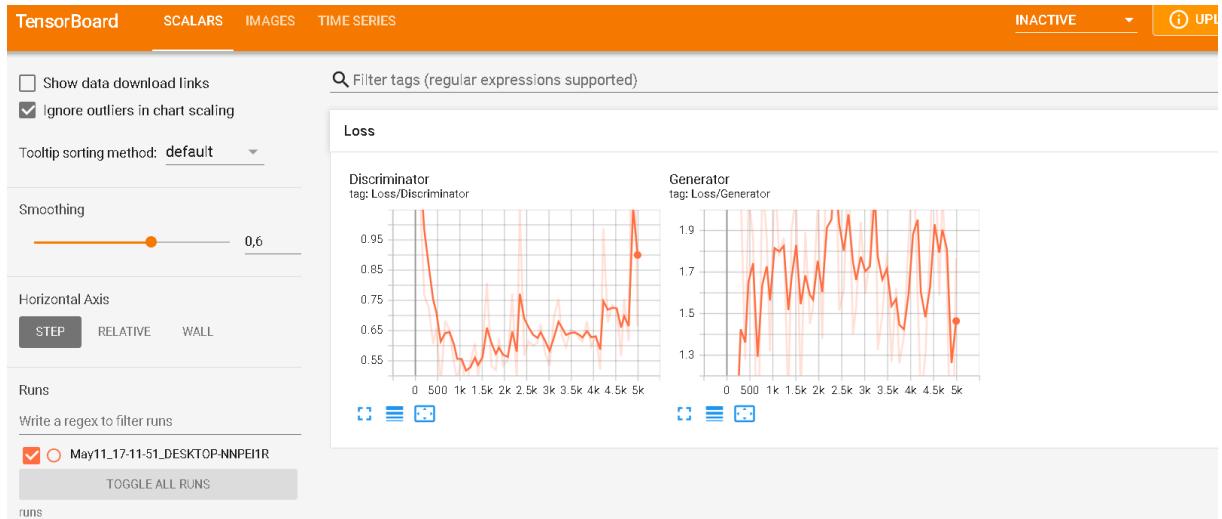


Figura 43: Evolución funciones de pérdida implementación técnica DcGan

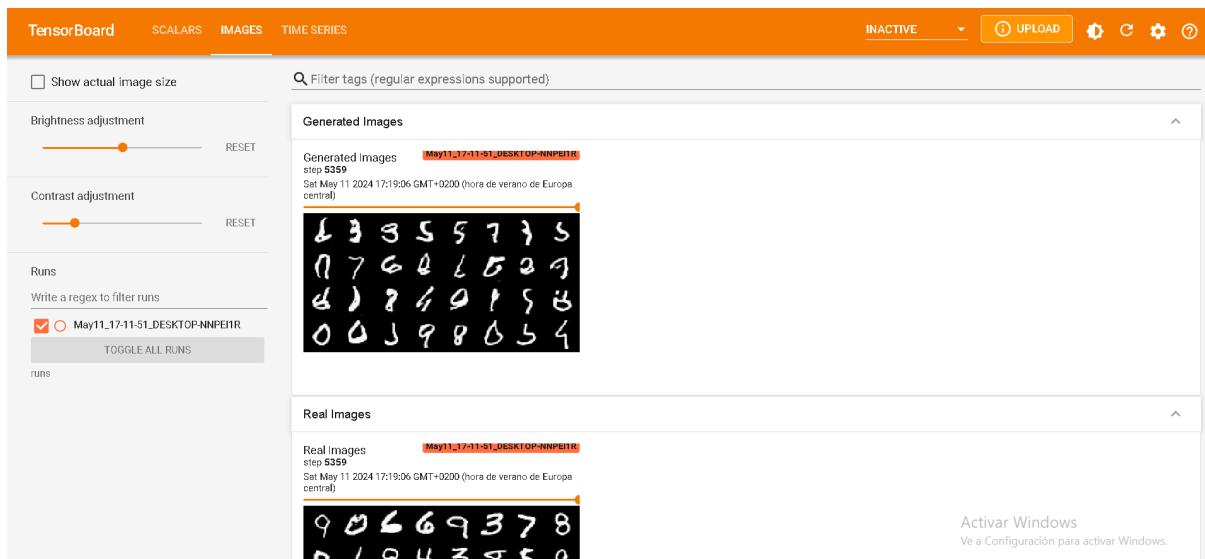


Figura 44: Resultados en tensor board de la implementación Técnica DcGan

Si le damos a reload es la esquina superior derecha se actualizarán las visualizaciones según el entrenamiento

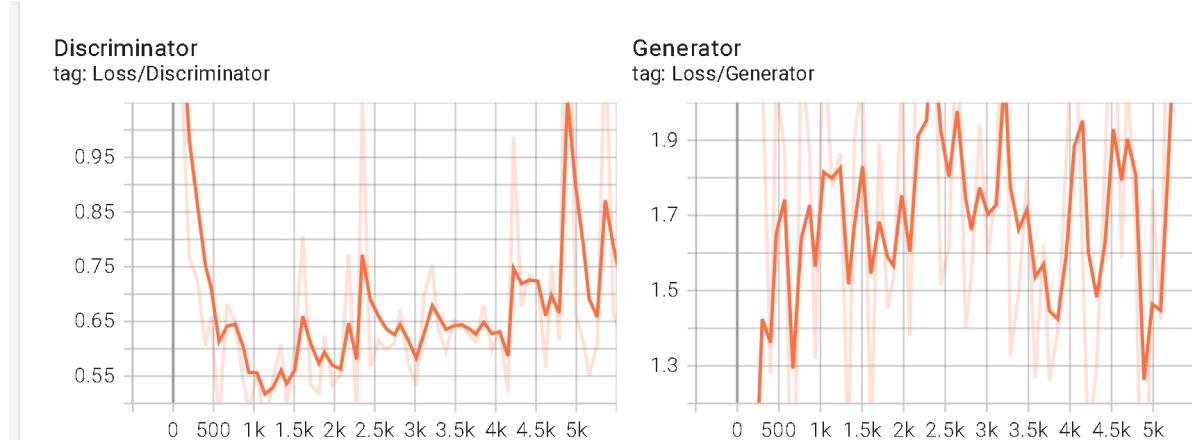
### 6.1.3 Análisis Resultados

Ya finalizado el entrenamiento obtenemos los siguientes resultados

GENERADAS	REALES
3 1 7 1 9 0 1 3	1 0 2 1 0 3 0 2
9 1 4 9 7 7 3 0	0 9 2 2 1 9 2 7
8 0 7 7 4 6 8	9 4 5 7 7 3 3 4
4 8 0 7 8 2 9 8	2 2 8 4 3 9 2 3

Figura 45: Comparación ejemplos Mnist generados vs reales

Podemos observar que los resultados son bastante similares a los reales, sin embargo hay algún fallo. Hay que mencionar que las DCGAN son las redes generativas antagónicas o adversariales más sencillas, ya que la función de pérdida se basa en la BCEloss. Siempre se pueden mejorar los resultados ajustando y encontrando los hyperparameters que más se ajusten con tu entrenamiento



*Figura 46: Gráfica evolución Función de Pérdida del Discriminador como del Generador; en la implementación técnica DcGan*

```
[47/50][0/469] Loss_D: 0.4604 Loss_G: 2.3620 D(x): 0.7945 D(G(z)): 0.1432 / 0.1668
[47/50][100/469] Loss_D: 0.3125 Loss_G: 2.5407 D(x): 0.8089 D(G(z)): 0.0665 / 0.1353
[47/50][200/469] Loss_D: 0.4733 Loss_G: 2.6422 D(x): 0.8115 D(G(z)): 0.1735 / 0.1154
[47/50][300/469] Loss_D: 0.3927 Loss_G: 2.6463 D(x): 0.8730 D(G(z)): 0.1817 / 0.1139
.8347 D(G(z)): 0.1280 / 0.1167
[49/50][0/469] Loss_D: 0.3536 Loss_G: 2.9303 D(x): 0.8973 D(G(z)): 0.1784 / 0.0919
[49/50][100/469] Loss_D: 0.4652 Loss_G: 1.6171 D(x): 0.7730 D(G(z)): 0.1454 / 0.2759
[49/50][200/469] Loss_D: 0.5535 Loss_G: 3.4565 D(x): 0.9384 D(G(z)): 0.2973 / 0.0671
[49/50][300/469] Loss_D: 0.5457 Loss_G: 1.4674 D(x): 0.7238 D(G(z)): 0.1270 / 0.3070
[49/50][400/469] Loss_D: 0.5487 Loss_G: 3.2298 D(x): 0.9660 D(G(z)): 0.3314 / 0.0719
```

*Figura 47: Métricas Durante el entrenamiento de la DcGAN*

Aquí podemos observar las funciones de pérdida del discriminador y del generador durante el entrenamiento. Como notamos, y continuaremos viendo en distintas GANs, el discriminador tiende a perder eficacia rápidamente en las primeras iteraciones de los entrenamientos de redes generativas adversariales. Sin embargo, se fortalece progresivamente. La gráfica del generador muestra fluctuaciones significativas, lo cual es habitual en la implementación de una DCGAN. A pesar de ello, ha logrado aprender las características básicas del dataset a partir de puro ruido. Dado que se trata de un dataset simple, si aplicáramos datasets con características más complejas observaremos que este tipo de GAN no es tan eficiente como las WGAN-GP PRO-GAN, con las cuales se pueden conseguir resultados más impactantes por su arquitectura más compleja comentada en el punto **4.2 y 4.3**.

## 8.2 IMPLEMENTACIÓN TÉCNICA WGAN-GP

La red generativa antagónica Wasserstein con gradient penalty, se distingue principalmente por su función de pérdida y la aplicación de la penalización de gradiente. Esta implementación técnica es esencial comprenderla porque para completar el objetivo principal del proyecto, mejora de los modelos mediante ejemplos generados por GANS (redes generativas adversariales/antagónicas) utilizaremos imágenes generadas por este tipo de red. El código de a continuación lo he usado posteriormente para la mejora de modelos.

### 8.2.1 Dataset

Vamos a utilizar en este caso el conjunto de datos ***Brain MRI Images for Brain Tumor Detection*** [25], con el cual generamos cerebros con tumores como sin tumores. El dataset está compuesto por 253 imágenes, 98 cerebros sin tumor y 155 cerebros con tumor. Generamos ambos con la misma implementación. Las imágenes no tienen todas la misma resolución de primeras, en el código las redefinimos de tamaño todas a 128x128 un tamaño muy efectivo para las redes neuronales degenerativas antagónicas.



Figuras 48: Dataset Cerebros

La estructura del dataset será la siguiente,

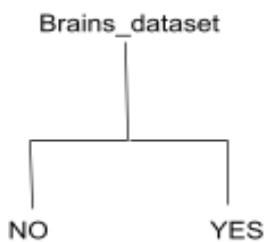


Figura 49: Estructura simple Dataset cerebros

Muy simple, accederemos al directorio de No si queremos generar cerebros sin tumor, y al contrario. Esto nos facilita para la generación pero nos dificulta cuando clasifiquemos las imágenes en la mejora de modelos.

### 8.2.2 Entorno

El entorno que he escogido para este ejemplo, ha sido **Google Collab** utilizando la tarjeta gráfica A100, ya que es significativamente más rápida y eficaz en algoritmos de IA que mi GPU local. Utilizaremos **tensor flow** y **keras**, keras nos va a permitir construir los modelos

de una manera mucho más intuitivo. Por último, **Tensor Board** para visualizar los logs. Importante mencionar google collab funciona con notebooks como júpiter.

[https://colab.research.google.com/drive/1Gq1VgCZIKgC4y9pV\\_miZycVDh0wJWtIB#scrollTo=ROs0uPD7kGu8](https://colab.research.google.com/drive/1Gq1VgCZIKgC4y9pV_miZycVDh0wJWtIB#scrollTo=ROs0uPD7kGu8)



Figura 50: Gpu para el entrenamiento WGAN-GP

Una vez subido el zip de los cerebros a generar lo descomprimimos tal que así:

```
!unzip Brains.zip
```

```
inflating: brain_tumor_dataset/no/17_no.jpg
inflating: brain_tumor_dataset/no/18_no.jpg
inflating: brain_tumor_dataset/no/19_no.jpg
inflating: brain_tumor_dataset/no/2_no.jpeg
inflating: brain_tumor_dataset/no/20_no.jpg
```

Figura 51: Unzip Brains.zip

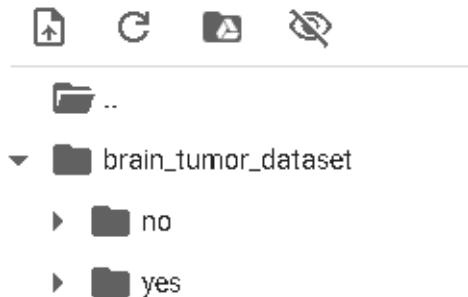


Figura 52: Estructura del Dataset Brains en Collab

Realizó los imports necesarios :

```

> import numpy as np

import tensorflow as tf
from tensorflow.keras import (
    layers,
    models,
    callbacks,
    utils,
    metrics,
    optimizers,
)

```

Figura 53: Imports necesarios para el código de la implementación WGAN-GP

Realizamos los imports necesarios, modelos, capas, optimizadores, métricas, callbacks para ejecutar funciones durante el entrenamiento etc...

### 8.2.3 Hiperparámetros

```

▶ IMAGE_SIZE = 128
CHANNELS = 3
BATCH_SIZE = 128
Z_DIM = 128
LEARNING_RATE = 0.0002
ADAM_BETA_1 = 0.5
EPOCHS = 20
CRITIC_STEPS = 5
GP_WEIGHT = 10.0
LOAD_MODEL = False

ADAM_BETA_2 = 0.9

```

Figura 54: Hiperparametros implementación Wgan-GP

Enfatizó que los hiper parámetros son cruciales así que los valores que establecemos aquí serán determinantes. Primero establecer el tamaño de las imágenes a 128x128, ya que vamos a diseñar la arquitectura tanto del discriminador como la del generador para trabajar con este tamaño de entrada y salida. Indico 3 canales, aunque las imágenes están solo en una escala de

grises, he considerado que se tengan en cuenta los canales rgb en caso de alguna anomalía. Elijo un tamaño de lote (**batch\_size**) a 128, un valor común equilibrando el uso de memoria y la velocidad de convergencia durante el entrenamiento, si queremos usar menos memoria podremos usar un **batch size** de 64, pero los resultados serán más pobres. La dimensión del vector ruido la establecemos a 128 como en las deep convolutional GANS.

El learning rate o tasa de aprendizaje más efectiva que he trasteado para esta implementación ha sido de 0.0002 algo pequeña pero es necesaria para asegurarse que las actualizaciones de los pesos sean suaves.

Los parámetros de Adam los establezco a  $\beta_1 = 0.5$  y  $\beta_2 = 0.9$ . El valor de beta 1 común suele ser 0.9 sin embargo he notado una notable mejora con un valor de 0.5 ya que el momento 3.7.3. 20 épocas han sido suficientes.

El parámetro **CRITICAL STEPS** y diferencia a la WGAN-GP de las DIGAN. **Este parámetro indica cuántas veces el crítico/discriminador se actualiza respecto por cada actualización del generador**, como todo hiper parámetro es muy sensible, pocos pasos del crítico harían a el generador demasiado fuerte no permitiendo aprender a el crítico simultáneamente parejos, y demasiadas actualizaciones del crítico por actualización del generador pueden hacerlo demasiado fuerte desembocando en una convergencia de mínimos, debido a la debilidad del generador. En este caso 5 actualizaciones del crítico por actualización del generador es estable.

Volvamos a el punto 4.2.2 a la fórmula de la penalidad de Gradiente :

$$\lambda \cdot (\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2$$

Bien el hiperparametro **GP\_WEIGHT** seria  $\lambda$  En este caso he encontrado el valor 10.0 estable, cumpliendo la condición de Lipshitz.

Load\_Model establecido a false, es una variable que actuaría como flag para un callback que veremos más adelante, en caso de que queramos cargar un modelo que previamente hayamos entrenado cambiamos el parámetro a False

En resumen, todo trata de ajustar los valores para lograr un balance entre el discriminador y el generador que no predomine uno o otro.

#### 8.2.4 Preprocesamiento de Datos

El dataset de cerebros no viene ya preparado para su aprendizaje, por ello deberemos realizar un preprocesamiento de datos.

```

import tensorflow as tf
import os
import matplotlib.pyplot as plt

# Variables Constantes
IMG_HEIGHT, IMG_WIDTH = 128, 128
BUFFER_SIZE = 60000
BATCH_SIZE = 128

def load_image(img_path):
    img = tf.io.read_file(img_path)
    img = tf.image.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
    img = (img - 127.5) / 127.5 # Normalizamos a [-1, 1]
    return img

```

Figura 55: Preprocesamiento de Datos

En la función **load image**, leemos las imágenes de el dataset, las hacemos decode es decir transformamos la imagen a datos para un tensor y normalizamos los valores a -1y 1.

```

# Crear dataset a partir del path
def create_dataset_from_directory(directory_path):
    # Listamos las imágenes del directorio y comprobamos que el directorio existe
    if not os.path.isdir(directory_path):
        raise ValueError(f"Provided path '{directory_path}' is not a directory.")

    all_image_paths = [os.path.join(directory_path, fname) for fname in os.listdir(directory_path) if os.path.isfile(os.path.join(directory_path, fname))]

    # Convertimos cada imagen del directorio a datos del tensor mediante la función load_image()
    path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
    image_ds = path_ds.map(load_image)
    image_ds = image_ds.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

    return image_ds

```

Figura 56: Creación del dataset a partir del Path.

Creamos un dataset de las imágenes del directorio transformadas a tensores para el entrenamiento después de llamar a load image, mezclamos las imágenes y las separamos en lotes/batches de 128.

```

#Cargar Datos entrenamiento
train_dir = '/content/yes' # Ajustar cuando queramos
train_dataset = create_dataset_from_directory(train_dir)

```

Figura 57: Cargar datos para el entrenamiento de el WGan-GP

Cargamos el dataset en la variable train dataset, train\_dir el directorio de las imágenes que vamos a generar, en este caso aquellas con tumor cerebral.

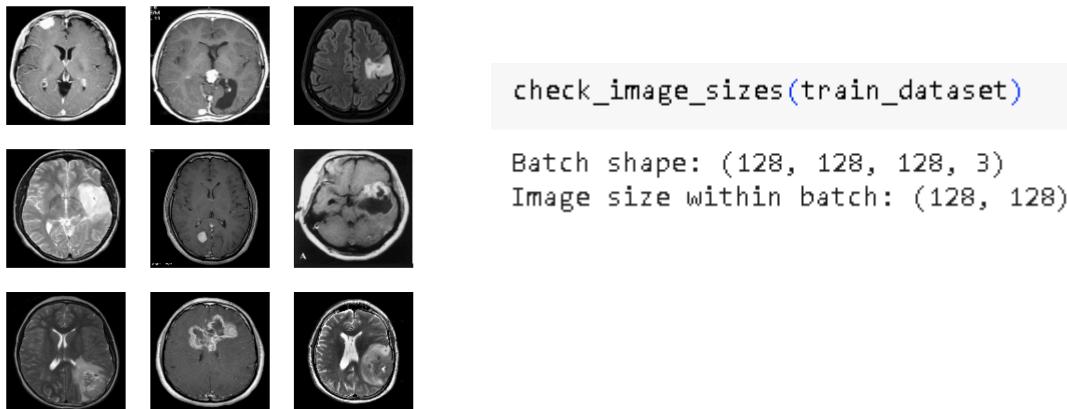
Para hacernos una idea de cómo son las imágenes reales de cerebros con tumor vamos a visualizar 9 y asegurarnos que su tamaño se ha redimensionado a 128x128 para los modelos crítico y generativo. Realizamos estas dos funciones :

```
# Funcion para enseñar las images
def show_images(dataset, n_images):
    plt.figure(figsize=(10, 10))
    for images in dataset.take(1):
        for i in range(min(n_images, BATCH_SIZE)): # Asegurarnos not to exceed batch size
            ax = plt.subplot(3, 3, i + 1)
            plt.imshow((images[i].numpy() + 1) / 2) # Rescale to [0, 1] for displaying
            plt.axis('off')
    plt.show()
def check_image_sizes(dataset):
    for images in dataset.take(1): # Cogemos un lote de 128
        print("Batch shape:", images.shape)
        print("Image size within batch:", images.shape[1:3]) # prints (height, width)
        break # Hacemos break porque para ver las dimensiones no necesitaremos mas que un lote (batch)

# Display images
show_images(train_dataset, 9) # Show 9
```

*Figura 58: Mostrar ejemplos de Cerebros del Dataset*

Básicamente usamos la librería matplotlib para mostrar las imágenes ordenadas, y reescalar las imágenes normalizadas a [0,1] solamente y repito solamente para la visualización de los datos.



*Figura 59: Ejemplos de cerebros del Dataset y sus propiedades.*

## 8.2.5 Discriminador/crítico

```

from tensorflow.keras import layers, models

IMAGE_SIZE = 128
CHANNELS = 3

# Input layer para el critic
critic_input = layers.Input(shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS))

# Building the critic architecture
x = layers.Conv2D(64, kernel_size=4, strides=2, padding="same")(critic_input) # Output shape: ~64x64x64
x = layers.LeakyReLU(0.2)(x)
x = layers.Conv2D(128, kernel_size=4, strides=2, padding="same")(x) # Output shape: ~32x32x128
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(256, kernel_size=4, strides=2, padding="same")(x) # Output shape: ~16x16x256
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(512, kernel_size=4, strides=2, padding="same")(x) # Output shape: ~8x8x512
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)
x = layers.Conv2D(1024, kernel_size=4, strides=2, padding="same")(x) # Output shape: ~4x4x1024
x = layers.LeakyReLU(0.2)(x)
x = layers.Dropout(0.3)(x)

# Flatten y output layer
x = layers.Flatten()(x) # Flatten the tensor para el output layer
x = layers.Dense(1)(x) # Output del critic

# Construimos el modelo
critic = models.Model(critic_input, x)
critic.summary()

```

Figura 60: Código del Discriminador/Critico WGAN-GP

Recibimos la imagen de tamaño 128x128 y 3 canales

En la primera capa, ajustó 64 filtros con un tamaño de kernel/filtro de 4x4, un stride de 2 y un padding para asegurar que la salida tenga las mismas dimensiones que la entrada dividida por el stride. Reducimos la dimensión espacial a la mitad. Aplicamos la función de activación **Leaky Relu** con un ajuste de pendiente pequeño de 0.2 para el flujo de gradiente y desactivamos aleatoriamente el 30 % de las neuronas para mejorar la variedad y generación del modelo con **dropout(0.3)(x)**, cada vez que llamamos a (x) estamos haciendo la operación sobre el crítico. Realizamos el mismo procedimiento o proceso hasta reducir el tamaño a 4x4 x 1024 características/canales. En la Capa final con **layers.Flatten()**(x), transformamos el tensor a un vector para poder calcular la distancia en una única salida mediante la capa **Dense(1)**. Recordemos, en WGAN-GP la salida del discriminador no es un clasificador binario sino una puntuación para calcular la distancia entre los datos reales y los generados.

La función `.summary()` de keras nos da un esquema bien definido sobre el proceso de nuestro modelo.

Layer (type)	Output Shape	Param #
<hr/>		
input_2 (InputLayer)	[(None, 128, 128, 3)]	0
conv2d_5 (Conv2D)	(None, 64, 64, 64)	3136
leaky_re_lu_5 (LeakyReLU)	(None, 64, 64, 64)	0
conv2d_6 (Conv2D)	(None, 32, 32, 128)	131200
leaky_re_lu_6 (LeakyReLU)	(None, 32, 32, 128)	0
dropout_4 (Dropout)	(None, 32, 32, 128)	0
conv2d_7 (Conv2D)	(None, 16, 16, 256)	524544
leaky_re_lu_7 (LeakyReLU)	(None, 16, 16, 256)	0
dropout_5 (Dropout)	(None, 16, 16, 256)	0
conv2d_8 (Conv2D)	(None, 8, 8, 512)	2097664
leaky_re_lu_8 (LeakyReLU)	(None, 8, 8, 512)	0
dropout_6 (Dropout)	(None, 8, 8, 512)	0
conv2d_9 (Conv2D)	(None, 4, 4, 1024)	8389632
leaky_re_lu_9 (LeakyReLU)	(None, 4, 4, 1024)	0
dropout_7 (Dropout)	(None, 4, 4, 1024)	0
flatten_1 (Flatten)	(None, 16384)	0
dense_1 (Dense)	(None, 1)	16385
<hr/>		
Total params:	11162561	(42.58 MB)
Trainable params:	11162561	(42.58 MB)
Non-trainable params:	0	(0.00 Byte)

Figura 61: Resumen del modelo Discriminativo de la Implementación WganGp

Los **parámetros**(parámetros) nos indican la cantidad de pesos y sesgos que la red aprende durante el entrenamiento. Se calculan tal que multiplicando las dimensiones del tensor altura anchura y mapa de características/canales.

## 8.2.6 Generador

```

from tensorflow.keras import layers, models

# Hyperparameters
Z_DIM = 128 # Dimension del vector ruido
GENERATOR_INITIAL_SIZE = 8
CHANNELS = 3

# Generator input: latent vector
generator_input = layers.Input(shape=(Z_DIM,))

# Empezamos con una dense layer y la redimensionamos a un tensor 3D : 8x8x1024
x = layers.Dense(GENERATOR_INITIAL_SIZE * GENERATOR_INITIAL_SIZE * 1024)(generator_input)
x = layers.LeakyReLU(0.2)(x)
x = layers.Reshape((GENERATOR_INITIAL_SIZE, GENERATOR_INITIAL_SIZE, 1024))(x)

# Conv2DTranspose para aumentar el tamaño : 16x16x512
x = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Conv2DTranspose 32x32x256
x = layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Conv2DTranspose 64x64x128
x = layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Conv2DTranspose 128x128x64
x = layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Final Conv2DTranspose para asegurarnos que el tamaño del output es el correcto : 128x128x3
x = layers.Conv2DTranspose(CHANNELS, kernel_size=3, strides=1, padding="same", activation="tanh")(x)

# Generator model
generator = models.Model(generator_input, x)
generator.summary()

```

Figura 62: Código del generador de Implementación técnica WGAN-GP

En caso del generador, como usualmente utilizaremos las convoluciones transpuestas para transformar el vector ruido en una imagen generada de las mismas dimensiones que recibe el crítico como input 128x128 x3.

En la input Layer del generador como de costumbre introducimos las dimensiones del vector ruido, **128**.

```
# Generator input: latent vector
generator_input = layers.Input(shape=(Z_DIM,))

# Empezamos con una dense layer y la redimensionamos a un tensor 3D : 8x8x1024
x = layers.Dense(GENERATOR_INITIAL_SIZE * GENERATOR_INITIAL_SIZE * 1024)(generator_input)
x = layers.LeakyReLU(0.2)(x)
x = layers.Reshape((GENERATOR_INITIAL_SIZE, GENERATOR_INITIAL_SIZE, 1024))(x)
```

Figura 63: Input del Generador Wgan-Gp y capas Iniciales

Definimos con las dimensiones iniciales del generador a 8x8x1024 en la capa **Dense()**, transformamos el vector ruido a un vector 8x8x1024, que en total son unas 65536 neuronas en la capa densa, cada una conectada completamente a la otra.

Al igual que el crítico, activamos la función con un coeficiente de pendiente de -0.2 para mantener el flujo del gradiente. Acto seguido realizamos **layers.Reshape**, esto transforma el vector 8x8x1024 de la capa densa a un tensor 3D 8x8x1024, como en las redes convolucionales tratamos con tensores este paso es necesario para que el generador con las cada convolución inversa o traspuesta aprenda y aumente la resolución de la img.

```
# Conv2DTranspose para aumentar el tamaño : 16x16x512
x = layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Conv2DTranspose 32x32x256
x = layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Conv2DTranspose 64x64x128
x = layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Conv2DTranspose 128x128x64
x = layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same")(x)
x = layers.BatchNormalization(momentum=0.8)(x)
x = layers.LeakyReLU(0.2)(x)

# Final Conv2DTranspose para asegurarnos que el tamaño del output es el correcto : 128x128x3
x = layers.Conv2DTranspose(CHANNELS, kernel_size=3, strides=1, padding="same", activation="tanh")(x)

# Generator model
generator = models.Model(generator_input, x)
generator.summary()
```

Figura 64: Fragmento de Código Generador Wgan-Gp

A partir de la input layer, en las **CONV2DTRANSPOSE**(convoluciones traspuestas), el generador disminuye los canales y aumenta la resolución hasta llegar al tamaño deseado.

16x16 to 32x32 (256 channels)

32x32 to 64x64 (128 channels)

64x64 to 128x128 (64 channels)

Es importante realizar la normalización por lote en cada paso por la capa de convolución transpuesta excepto en la ultima, **layers.Batch Normalization(momentum=0.8)**, momentum es el parámetro de desplazamiento, $\beta$  , visto en el punto **3.6.1.**

$$y_i = \hat{\gamma}x_i + \beta$$

Llegamos a la última capa la de salida del generador:

```
# Final Conv2DTranspose para asegurarnos que el tamaño del output es el correcto : 128x128x3
x = layers.Conv2DTranspose(CHANNELS, kernel_size=3, strides=1, padding="same", activation="tanh")(x)

# Generator model
generator = models.Model(generator_input, x)
generator.summary()
```

Figura 65: Capa se salida y definicion de el Modelo Generador WGAN-GP

Nos aseguramos de que la imagen es 128x128 x3 canales, y utilizamos la función de activación tanh para escalar los valores entre [-1,1].models.Model(generator input,x), creamos el modelo generador con el input y el output y utilizamos la función de summary de keras que nos proporciona el siguiente resumen de el funcionamiento del generador:

Model: "model_2"	Layer (type)	Output Shape	Param #
	=====	=====	=====
	input_3 (InputLayer)	[(None, 128)]	0
	dense_2 (Dense)	(None, 65536)	8454144
	leaky_re_lu_10 (LeakyReLU)	(None, 65536)	0
	reshape (Reshape)	(None, 8, 8, 1024)	0
	conv2d_transpose (Conv2DTranpose)	(None, 16, 16, 512)	8389120
	batch_normalization (BatchNormalization)	(None, 16, 16, 512)	2048
	leaky_re_lu_11 (LeakyReLU)	(None, 16, 16, 512)	0
	conv2d_transpose_1 (Conv2DTranspose)	(None, 32, 32, 256)	2097408
	batch_normalization_1 (BatchNormalization)	(None, 32, 32, 256)	1024
	leaky_re_lu_12 (LeakyReLU)	(None, 32, 32, 256)	0
	conv2d_transpose_2 (Conv2DTranspose)	(None, 64, 64, 128)	524416
	batch_normalization_2 (BatchNormalization)	(None, 64, 64, 128)	512
	leaky_re_lu_13 (LeakyReLU)	(None, 64, 64, 128)	0
	conv2d_transpose_3 (Conv2DTranspose)	(None, 128, 128, 64)	1311136
	batch_normalization_3 (BatchNormalization)	(None, 128, 128, 64)	256
	leaky_re_lu_14 (LeakyReLU)	(None, 128, 128, 64)	0
	conv2d_transpose_4 (Conv2DTranspose)	(None, 128, 128, 3)	1731

Figura 66: Estructura del Generador WGAN-GP

### 8.2.7 Definiendo el Entrenamiento(Clase WGAN-GP)

El entrenamiento de WGAN-GP es distinto a las redes adversariales comunes, el mayor cambio reside en el entrenamiento, concretamente en las funciones de pérdida y el añadido de la penalidad del gradiente así como la introducción de la distancia.

Primeramente iniciamos la clase WGAN-GP heredando de la clase models de keras

```
class WGANGP(models.Model):
    def __init__(self, critic, generator, latent_dim, critic_steps, gp_weight):
        super(WGANGP, self).__init__()
        self.critic = critic
        self.generator = generator
        self.latent_dim = latent_dim
        self.critic_steps = critic_steps
        self.gp_weight = gp_weight
```

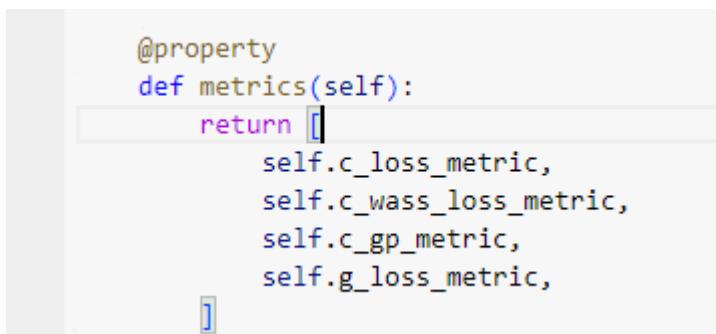
Figura 67: Constructor clase WGAN-GP

A continuación iniciamos el constructor del entrenamiento, inicializamos el crítico desarrollado anteriormente, el generador, y los hiper parámetros latent dim(dimensiones de vector ruido), critical steps (actualizaciones del crítico respecto a cada actualización del generador) y por ultimo la “fuerza” o el peso de la penalidad del gradiente que afectará a la función de pérdida, tal como funciona en la teoría 4.2.

```
def compile(self, c_optimizer, g_optimizer):
    super(WGANGP, self).compile()
    self.c_optimizer = c_optimizer
    self.g_optimizer = g_optimizer
    self.c_wass_loss_metric = metrics.Mean(name="c_wass_loss")
    self.c_gp_metric = metrics.Mean(name="c_gp")
    self.c_loss_metric = metrics.Mean(name="c_loss")
    self.g_loss_metric = metrics.Mean(name="g_loss")
```

Figura 68: Metodo Compile WGAN-GP

A continuación, dentro de la clase configuramos el método de compilación del modelo de entrenamiento, inicializamos los optimizadores para el crítico como para el generador. Acto seguido definimos las métricas principales **c\_gp** (**critic gradient penalty**), **c\_loss**(**critic loss**), **c\_wass\_loss**(critic wasserstein loss ), **g\_loss**(generator loss), para trackear/monitorizar el entrenamiento.



```

@property
def metrics(self):
    return [
        self.c_loss_metric,
        self.c_wass_loss_metric,
        self.c_gp_metric,
        self.g_loss_metric,
    ]

```

Figura 69: Propiedad Metrics

Definimos la propiedad `metrics` que devuelve la siguiente lista de métricas muy importantes para supervisar y evaluar el entrenamiento:

**closs metric:** Esta métrica mide la pérdida total del crítico (closs). La pérdida total del crítico incluye tanto la pérdida de Wasserstein como la penalización de gradiente. Esta métrica es esencial para entender cómo el crítico está aprendiendo a distinguir entre imágenes reales y generadas.

$$\max_d (E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)] + \lambda E_{\hat{x} \sim p_{\hat{X}}}(\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2)$$

**c\_wass\_loss\_metric:** Recogemos las métricas de la función de pérdida wasserstein, recordemos es la distancia entre la media de las puntuaciones del crítico sobre las imágenes reales y las generadas por el generador.

$$\max_d (E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)])$$

**c\_gp\_metric:** Con esta métrica medimos la penalización del gradiente para que se cumpla la condición Lipschitz

$$\lambda \cdot (\left\| \Delta_x \hat{D}(\hat{x}) \right\|_2 - 1)^2$$

**gloss metric:** Medimos la pérdida del generador, la pérdida del generador es la media negativa de las predicciones del crítico sobre las imágenes generadas. Una pérdida más baja para el generador indica que las imágenes generadas son más convincentes para el crítico.

Definimos la función de la penalidad del gradiente,

```

J
def gradient_penalty(self, batch_size, real_images, fake_images):
    alpha = tf.random.normal([batch_size, 1, 1, 1], 0.0, 1.0, dtype=tf.float32)
    real_images = tf.cast(real_images, tf.float32)
    fake_images = tf.cast(fake_images, tf.float32)

    diff = fake_images - real_images
    interpolated = real_images + alpha * diff

    with tf.GradientTape() as gp_tape:
        gp_tape.watch(interpolated)
        pred = self.critic(interpolated, training=True)

    grads = gp_tape.gradient(pred, [interpolated])[0]
    norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
    gp = tf.reduce_mean((norm - 1.0) ** 2)
    return gp

```

Figura 70: Funcion penalización del gradiente

Aplicando la teoría del **4.2.2**, definiremos la penalidad del gradiente, recibimos como parámetros, el modelo de entrenamiento, el tamaño de lote, las imágenes reales, las imágenes generadas(*fake images*)

Generamos alpha **alpha = tf.random.normal([batch\_size, 1, 1, 1], 0.0, 1.0, dtype=tf.float32)**, alpha es una distribución con media 0.0 y desviación estándar 1.0, en **[batch\_size, 1, 1, 1]** indicamos que se generará un **valor aleatorio** de alpha para cada muestra en el lote, pero el mismo valor se aplicará a todos los píxeles de una imagen.

```

diff = fake_images - real_images
interpolated = real_images + alpha * diff

```

Figura 71: Código de interpolación de muestras

En este segmento del código realizamos la interpolación, con **diff= fake images(generadas) - real images**, calculamos la diferencia que nos da la distancia pixel por pixel para calcular la diferencia entre ambas imágenes que nos ayudará posteriormente para el cálculo de la penalidad de los gradientes.

A continuación realizamos el **proceso de interpolación**, generamos las muestras interpoladas sumando/mezclando las imágenes reales más el producto de la diferencia (**diff**) por alpha para asegurar que en la interpolación se realiza en distintas porciones de la imagen.

Por último analizaremos el cálculo de los gradientes y el cálculo la penalidad del gradiente:

```

with tf.GradientTape() as gp_tape:
    gp_tape.watch(interpolated)
    pred = self.critic(interpolated, training=True)

grads = gp_tape.gradient(pred, [interpolated])[0]
norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
gp = tf.reduce_mean((norm - 1.0) ** 2)
return gp

```

Figura 72: Cálculo de la penalidad del gradiente y de los gradientes,

**tf.GradientType()**, es una librería de tensorflow que la vamos a utilizar para capturar y calcular los gradientes de los tensores interpolados.

Mediante **pred = self.critic (interpolate, true)**, acto seguido guardamos las predicciones del crítico sobre las imágenes interpoladas, posteriormente calculamos los gradientes de las predicciones sobre las imágenes interpoladas **grads = gp\_tape.gradient(pred, [interpolated])[0]**.

$$\text{grads} = \Delta_{\hat{x}} D(\hat{x})$$

$$\text{Calculamos la norma } \text{norm} = \|\nabla_{\hat{x}} D(\hat{x})\|_2 = \sqrt{\left(\frac{\partial D}{\partial x_1}\right)^2 + \left(\frac{\partial D}{\partial x_2}\right)^2 + \dots + \left(\frac{\partial D}{\partial x_n}\right)^2}$$

mediante **norm = tf.sqrt(tf.reduce\_sum(tf.square(grads),axis=[1,2,3]))**, con la norma medimos los cambios en las predicciones del crítico con respecto a las perturbaciones minúsculas de las imágenes interpoladas.

En `gp = tf.reduce_mean((norm - 1.0) ** 2)`, en este fragmento de código calculamos la penalidad del gradiente, forzamos a los gradientes a tener una norma cercana a 1, reduce mean es para calcular la media de estos valores, esto es equivalente a  $(\|\Delta_x D(\hat{x})\|_2 - 1)^2$  visto en el punto 4.2.2, y mediante el return gp devolvemos ya la penalización de gradiente de la función.

## Train\_Step

En el train step vamos a definir las actualizaciones del generador y del discriminador, métricas, y calcular las funciones de pérdida.

### Vista General

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]

    for i in range(self.critic_steps):
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )

        with tf.GradientTape() as tape:
            fake_images = self.generator([
                random_latent_vectors, training=True
            ])
            fake_predictions = self.critic(fake_images, training=True)
            real_predictions = self.critic(real_images, training=True)

            c_wass_loss = tf.reduce_mean(fake_predictions) - tf.reduce_mean(
                real_predictions
            )
            c_gp = self.gradient_penalty(
                batch_size, real_images, fake_images
            )
            c_loss = c_wass_loss + c_gp * self.gp_weight

            c_gradient = tape.gradient(c_loss, self.critic.trainable_variables)
            self.c_optimizer.apply_gradients(
                zip(c_gradient, self.critic.trainable_variables)
            )

        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )
        with tf.GradientTape() as tape:
            fake_images = self.generator(random_latent_vectors, training=True)
            fake_predictions = self.critic(fake_images, training=True)
            g_loss = -tf.reduce_mean(fake_predictions)

            gen_gradient = tape.gradient(g_loss, self.generator.trainable_variables)
            self.g_optimizer.apply_gradients(
                zip(gen_gradient, self.generator.trainable_variables)
            )

        self.c_loss_metric.update_state(c_loss)
        self.c_wass_loss_metric.update_state(c_wass_loss)
        self.c_gp_metric.update_state(c_gp)
        self.g_loss_metric.update_state(g_loss)

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

Figura 73: Train\_step Wgan-GP

Para entenderlo mejor paso a paso, vayamos por partes

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]

    for i in range(self.critic_steps):
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim)
        )

        with tf.GradientTape() as tape:
            fake_images = self.generator(
                random_latent_vectors, training=True
            )
            fake_predictions = self.critic(fake_images, training=True)
            real_predictions = self.critic(real_images, training=True)

            c_wass_loss = tf.reduce_mean(fake_predictions) - tf.reduce_mean(
                real_predictions
            )
            c_gp = self.gradient_penalty(
                batch_size, real_images, fake_images
            )
            c_loss = c_wass_loss + c_gp * self.gp_weight

            c_gradient = tape.gradient(c_loss, self.critic.trainable_variables)
            self.c_optimizer.apply_gradients(
                zip(c_gradient, self.critic.trainable_variables)
            )
```

Figura 74: Fragmento 1 de código Train\_step WGAN-GP

Obtenemos el tamaño de lote de imágenes reales (`batch_size`), entramos en el `for` en este loop como podemos observar recorriendo las actualizaciones del crítico ya que vamos a actualizar el crítico 5 veces antes de actualizar el generador.

Generamos vectores de ruido aleatorio para mandar a el generador para generar las imágenes falsas, los guardamos en la variable **random latent vectors**, pasamos posteriormente los vectores de ruido a el generador para que se entrene y genere las imágenes, las imágenes generadas las guardamos en la variable **fake images**:

```

        )

    with tf.GradientTape() as tape:
        fake_images = self.generator(
            random_latent_vectors, training=True
        )
        fake_predictions = self.critic(fake_images, training=True)
        real_predictions = self.critic(real_images, training=True)

        c_wass_loss = tf.reduce_mean(fake_predictions) - tf.reduce_mean(
            real_predictions
        )
        c_gp = self.gradient_penalty(
            batch_size, real_images, fake_images
        )
        c_loss = c_wass_loss + c_gp * self.gp_weight

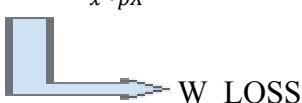
        c_gradient = tape.gradient(c_loss, self.critic.trainable_variables)
        self.c_optimizer.apply_gradients(
            zip(c_gradient, self.critic.trainable_variables)
        )
    
```

Figura 75: Fragmento 2 de código Train\_step WGAN-GP

Acto seguido como podemos ver en el código, muy similar al caso DCAN Mnist, guardamos las predicciones del crítico respecto a las imágenes generadas :**fake predictions**= D(G(z)) y las predicciones respecto a las imágenes reales **real predictions** = D(x).

Calculamos la función de pérdida Wasserstein en la métrica **c wass loss** aplicando la siguiente fórmula ( $E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)]$ ), como observamos en la teoría en el punto **4.2.1**

Llamamos al método anterior **gradient penalty** para calcular la penalidad del gradiente, y calculamos la pérdida total del crítico en la métrica **closs** tal que así:

$$\text{c\_loss} = (E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)]) + \lambda \cdot (\left\| \Delta_x^{\hat{D}}(\hat{x}) \right\|_2 - 1)^2$$


El parametro  $\lambda$  es el hiper parametro que ajustamos a el principio del código que ajusta el peso de la penalización del gradiente

Seguimos en el train step, por último en loop de las actualizaciones del crítico, recogemos con la librería tape.gradientes de tensorflow los gradientes de la pérdida del crítico(**closs**) y mediante **self.c\_optimizer.apply\_gradients**, aplicamos los gradientes para actualizar los pesos del crítico.

Una vez realizadas las 5 actualizaciones del crítico tenemos que salir del loop y **actualizar el generador**:

```
random_latent_vectors = tf.random.normal(
    shape=(batch_size, self.latent_dim)
)
with tf.GradientTape() as tape:
    fake_images = self.generator(random_latent_vectors, training=True)
    fake_predictions = self.critic(fake_images, training=True)
    g_loss = -tf.reduce_mean(fake_predictions)

    gen_gradient = tape.gradient(g_loss, self.generator.trainable_variables)
    self.g_optimizer.apply_gradients(
        zip(gen_gradient, self.generator.trainable_variables)
)
```

Figura 76: Fragmento 3 de Train\_step WGAN-GP

Otra vez generamos vectores de ruido aleatorio, entrenamos el generador con estos vectores generamos imágenes y las guardamos en **fake images**, a continuación como antes hacemos las predicciones del crítico sobre las imágenes generadas **D(G(z))**

Ahora calculamos la función de pérdida del generador **gloss** que será la media negativa de las predicciones del crítico sobre las imágenes generadas o “falsas”.  $Gloss = -D(G(z))$ . Finalmente al igual que el crítico calculamos los gradientes de la pérdida del generador respecto a sus variables y los aplicamos para actualizar los pesos del generador.

Seguimos en el método train step, por ultimo habra que actualizar todas las métricas definidas gracias a el método de keras `update_status` y hacer un return de las métricas a tiempo real para poder monitorearlas en pleno entrenamiento

```
self.c_loss_metric.update_state(c_loss)
self.c_wass_loss_metric.update_state(c_wass_loss)
self.c_gp_metric.update_state(c_gp)
self.g_loss_metric.update_state(g_loss)

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

Figura 77: Fragmento final

Con esto ya tenemos la class WAGON-GP en nuestro código estructurada, ahora tenemos que crearla, compilarla y ejecutarla.

### 8.2.8 Instanciamos y Compilamos

Creamos la clase anteriormente construida con los respectivos hiper parámetros: y el modelo del crítico como el del generador:

```
# Instanciamos / Creamos la clase GAN
wgangp = WGANGP(
    critic=critic,
    generator=generator,
    latent_dim=Z_DIM,
    critic_steps=CRITIC_STEPS,
    gp_weight=GP_WEIGHT,
```

Figura 78: Instanciamos modelo WGAN-GP

Ahora acto seguido compilamos el modelo WGan-GP y especificamos los optimizadores del crítico y el generador con sus respectivos hiper parámetros definidos anteriormente.

```
[ ] # Compile the GAN
from tensorflow.keras import optimizers
from tensorflow.keras import metrics
wgangp.compile(
    c_optimizer=optimizers.Adam(
        learning_rate=LEARNING_RATE, beta_1=ADAM_BETA_1, beta_2=ADAM_BETA_2
    ),
    g_optimizer=optimizers.Adam(
        learning_rate=LEARNING_RATE, beta_1=ADAM_BETA_1, beta_2=ADAM_BETA_2
    ),
)
```

Figura 79: Compilamos WGan-GP

## 8.2.9 CallBacks y Checkpoint

Para poder llamar a un método mientras se entrena el modelo, debemos definir callbacks

Inicialmente, he realizado la clase Image Save Callback que como su nombre indica, llamaremos a esta clase para guardar las imágenes generadas en el directorio que indicamos, al tener esta posibilidad **este método es muy util para mezclar las imágenes generadas con las reales para posteriormente cargarlas en otro modelo IA y mejorarlo**. Vamos a Analizar como funciona el código:

```

class ImageSaverCallback(tf.keras.callbacks.Callback):
    def __init__(self, output_dir, num_images=18, latent_dim=128, img_shape=(128, 128)):
        self.output_dir = output_dir
        self.num_images = num_images
        self.latent_dim = latent_dim
        self.img_shape = img_shape
        # Asegura que el directorio existe sino lo crea
        os.makedirs(output_dir, exist_ok=True)

    def on_epoch_end(self, epoch, logs=None):
        # Revisamos si la epoca es la 14 o a partir de la 14
        if epoch >= 14:
            # Generamos el vector ruido
            random_latent_vectors = tf.random.normal(shape=(self.num_images, self.latent_dim))
            # Generamos Imagenes a partir del ruido
            generated_images = self.model.generator(random_latent_vectors, training=False)
            # Reescalamos los vectores [-1, 1] a [0, 255]
            generated_images = (generated_images * 0.5 + 0.5) * 255
            generated_images = tf.cast(generated_images, tf.uint8)

            # Guardamos las imagenes generadas en cada epoch
            for i, img in enumerate(generated_images):
                img_path = os.path.join(self.output_dir, f"epoch_{epoch+1}_image_{i+1}.png")
                img = tf.image.resize(img, self.img_shape) # Hacemos resize para asegurarnos que las imagenes se guardan
                tf.keras.preprocessing.image.save_img(img_path, img)

    #Llamamos a la clase y la guardamos en image_saver_callback
image saver callback = ImageSaverCallback(output dir="/content/yes resized", num images=18, latent dim=Z DIM)

```

Figura 80: CallBacks y Checkpoints Implementación Tecnica WGan-Gp

Inicializamos la clase en definit indicamos el número de imágenes que queremos que se generen, el dimensión Z=128 y las dimensiones de la imágenes

Luego en def **on\_epoch\_end** :

```

def on_epoch_end(self, epoch, logs=None):
    # Revisamos si la epoca es la 14 o a partir de la 14
    if epoch >= 14:
        # Generamos el vector ruido
        random_latent_vectors = tf.random.normal(shape=(self.num_images, self.latent_dim))
        # Generamos Imagenes a partir del ruido
        generated_images = self.model.generator(random_latent_vectors, training=False)
        # Reescalamos los vectores [-1, 1] a [0, 255]
        generated_images = (generated_images * 0.5 + 0.5) * 255
        generated_images = tf.cast(generated_images, tf.uint8)

        # Guardamos las imagenes generadas en cada epoch
        for i, img in enumerate(generated_images):
            img_path = os.path.join(self.output_dir, f"epoch_{epoch+1}_image_{i+1}.png")
            img = tf.image.resize(img, self.img_shape) # Hacemos resize para asegurarnos q
            tf.keras.preprocessing.image.save_img(img_path, img)

```

Figura 81: Función OnEpochEnd WGan-Gp

Por ultimo:

```
#Llamamos a la clase y la guardamos en image_saver_callback
image_saver_callback = ImageSaverCallback(output_dir="/content/yes_resized", num_images=18, latent_dim=Z_DIM)
```

Este código es muy reutilizable porque podemos cambiar el número de imágenes generadas así como el directorio donde se van a guardar, este callback lo reutilizarse posteriormente para generar otros ejemplos de imágenes posteriormente a el igual que la class **WAGON-GP** y utilizarlos en la mejora de modelos. Este **CallBack** además nos sirve para monitorear la calidad de las imágenes generadas a partir del epoch que especifiquemos en el if.

Para guardar checkpoints (puntos de guardado ) para nuestro entrenamiento en caso de que se interrumpa debido a cualquier imprevisto

```

# Creamos un checkpoint
from tensorflow.keras import optimizers
from tensorflow.keras import callbacks
model_checkpoint_callback = callbacks.ModelCheckpoint(
    filepath="/content/checkpoint/checkpoint.ckpt",
    save_weights_only=True,
    save_freq="epoch",
    verbose=0,
)
#Llamamos a los logs de tensorboard mediante un callback para v
tensorboard_callback = callbacks.TensorBoard(log_dir="./logs")

```

Figura 82: Checkpoint

Indicamos el path donde queremos que se guarde y la información que queremos que guarde que son los pesos al final de cada época. E verbose=0 indicamos que no se imprima nada tras cargar cada checkpoint

### 8.2.10 Ejecutamos el Entrenamiento

```
wgangp.fit(
    train_dataset,
    epochs=EPOCHS,
    steps_per_epoch=400,
    callbacks=[
        model_checkpoint_callback,
        tensorboard_callback,
        image_saver_callback
    ],
)
```

Figura 83: Wgan-GP.fit

Llamando a `.fit` comenzamos/ejecutamos el entrenamiento de **WGAN-GP**, pasamos el dataset, los pasos por epoch y los epochs, en este caso he decidido **400** pasos por época ya que daba mejores resultados pero este hiperpara otro, siempre es reajustable a otros datasets etc con objetivo de mejorar la precisión de las imágenes generadas. **Ejecutamos el código y empezamos el entrenamiento**

```
Epoch 1/20
400/400 [=====] - 147s 255ms/step - c_loss: -78.8709 - c_wass_loss: -118.1967 - c_gp: 3.9326 - g_loss: 22.7077
Epoch 2/20
400/400 [=====] - 97s 241ms/step - c_loss: -23.8443 - c_wass_loss: -30.0047 - c_gp: 0.6160 - g_loss: 19.2792
Epoch 3/20
400/400 [=====] - 96s 241ms/step - c_loss: -15.1309 - c_wass_loss: -19.0543 - c_gp: 0.3923 - g_loss: 15.2931
Epoch 4/20
400/400 [=====] - 96s 241ms/step - c_loss: -18.5992 - c_wass_loss: -21.4905 - c_gp: 0.2891 - g_loss: 4.4763
Epoch 5/20
400/400 [=====] - 97s 241ms/step - c_loss: -10.7582 - c_wass_loss: -13.1718 - c_gp: 0.2414 - g_loss: 3.0096
Epoch 6/20
400/400 [=====] - 97s 241ms/step - c_loss: -11.2247 - c_wass_loss: -13.2627 - c_gp: 0.2038 - g_loss: -110.1490
Epoch 7/20
400/400 [=====] - 97s 241ms/step - c_loss: -17.4188 - c_wass_loss: -20.4436 - c_gp: 0.3025 - g_loss: 64.8246
Epoch 8/20
400/400 [=====] - 97s 241ms/step - c_loss: 14.7029 - c_wass_loss: 11.4009 - c_gp: 0.3302 - g_loss: -841.0105
Epoch 9/20
400/400 [=====] - 96s 241ms/step - c_loss: -5.2945 - c_wass_loss: -7.4996 - c_gp: 0.2205 - g_loss: -546.7908
Epoch 10/20
400/400 [=====] - 96s 241ms/step - c_loss: -2.3037 - c_wass_loss: -3.3766 - c_gp: 0.1073 - g_loss: -448.6449
Epoch 11/20
```

Figura 84: Progreso entrenamiento WGAN-GP

### 8.2.11 Análisis Resultados

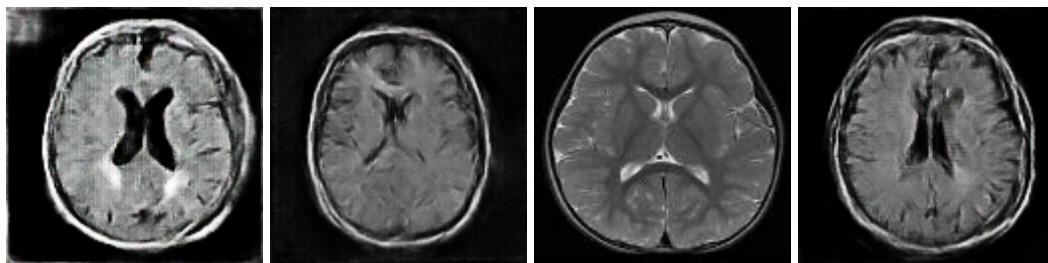
Analizemos los resultados tras 20 épocas:

```
-- epoch 18/20 -- 400/400 [=====] - 97s 241ms/step - c_loss: 12.0047 - c_wass_loss: 10.0144 - c_gp: 0.1990 - g_loss: 374.3512  
Epoch 19/20  
400/400 [=====] - 97s 241ms/step - c_loss: -28.9739 - c_wass_loss: -31.4116 - c_gp: 0.2438 - g_loss: 841.8892  
Epoch 20/20  
400/400 [=====] - 97s 241ms/step - c_loss: 8.7772 - c_wass_loss: 6.9371 - c_gp: 0.1840 - g_loss: -1468.3656  
<keras.src.callbacks.History at 0x79e94c334730>
```

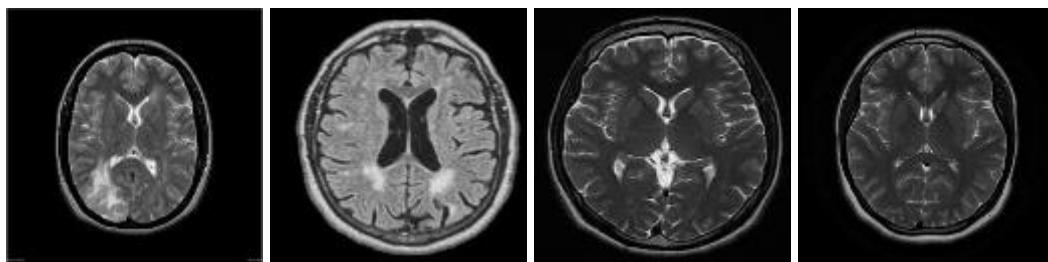
Figura 85: Últimos Epochs implementación WGAN-GP

El entrenamiento tardó alrededor de 40 minutos utilizando el procesador A 100 de nvidia en google Collab. Veamos los resultados de las imágenes generadas:

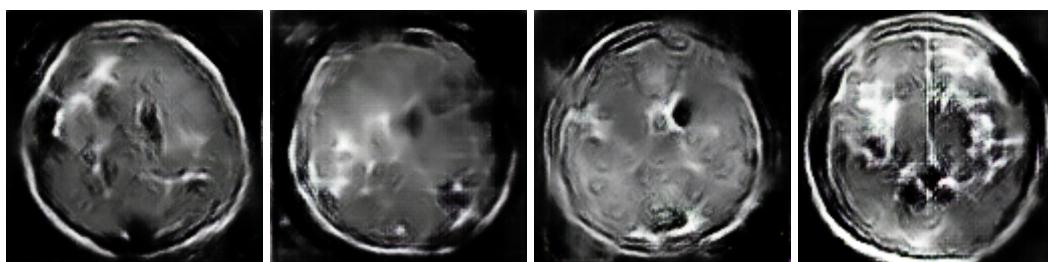
**Generadas sin tumor**



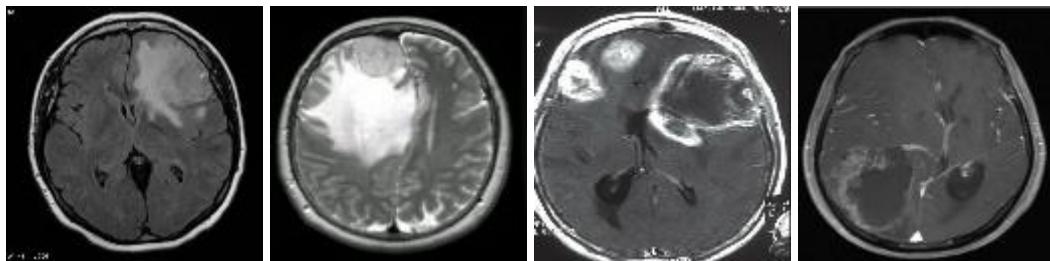
**Reales sin Tumor**



**Generadas Con tumor**



## Reales con Tumor



*Figura 86: Resultados y comparativa cerebros generados vs reales*

Los resultados de WGAN-GP son impactantes respecto a DCGAN sobre todo para los cerebros sin tumor, el hecho de añadir penalidad de gradiente y cambiar la función de pérdida, que impacta directamente en el entrenamiento, mejora considerablemente el modelo, haciéndolo indistinguible para el ojo humano entre las imágenes generadas como las reales.

Por último es importante subrayar como dicho al inicio de la implementación, que para generar cerebros sin tumor o con tumor simplemente cambiar el directorio del train dataset.

### 9.3 IMPLEMENTACIÓN TÉCNICA PROGAN

Una vez ya analizado e implementado el wasserstein Gan, procedemos a implementar Progressive Gan, ta gan es la más precisa pero también la más costosa de implementar en todos los niveles memoria, código y recursos. El código refleja prácticamente la teoría anterior estudiada. Es importante subrayar que esta implementación técnica está basada en [Paper de nvidia progan] y utilizaremos la función de pérdida vista en WGAN-GP

#### 9.3.1 Dataset

Para esta implementación utilizaremos el dataset celebaHQ, consiste de 30.000 imágenes de de resolución 1024 x 1024, he elegido este dataset para esta implementación porque son datos muy difíciles de generar ya que cada cara es diferente, para poder ver los impresionantes resultados de las redes generativas adversariales progresivas.

El dataset proviene de [26] y su estructura es la siguiente:

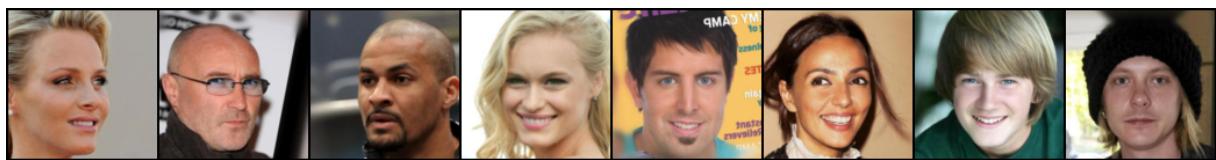


Figura 87: Datos de CelebaHQ

### 9.3.2 Entorno

El entorno que he escogido para la implementación de las redes degenerativas progresivas, es lenguaje de programación como de costumbre python, utilizaremos la librería **Mytorch** y la de **Tensorboard** de **Tensorflow**.

En esta implementación utilizaré **VS CODE**, ejecutaré y entrenaré el modelo en mi propia máquina utilizando mi GPU NVIDIA GTX 2060 gracias a la librería **CUDA**. En este caso he elegido este entorno debido a que aunque la nube google collab me daria mas rendimiento con la gráfica A 100, el presupuesto del proyecto excedería los límites, debido a la enorme cantidad de recursos que consume un entrenamiento tan complejo como lo son las redes generativas adversariales progresivas (**PROGAN**).

### 9.3.3 Estructura Del Código

La Estructura del código sera la siguiente ; el dataset, los logs para tensorboard, model.py(modelos generador y discriminador), train.py (Entrenamiento) y util.spy donde almacenar sobre todo funciones para visualizacion de las métricas, checkpoints para los models y la función de cálculo de la penalidad de gradiente y por ultimo config.py donde guardaremos los hiperparamteros para el entrenamiento.

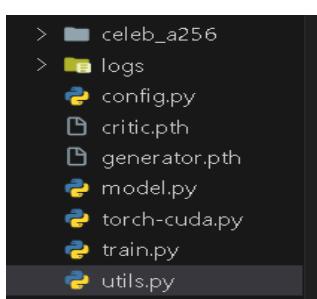


Figura 88: Estructura código Pro-GAN

### 9.3.4 Configuración Inicial e hiperparámetros

Comenzaremos con el archivo config.py, donde estableceremos diferentes path e hiperparámetros para la realización del entrenamiento.

```
import torch
from math import log2

START_TRAIN_AT_IMG_SIZE = 128
DATASET = 'celeb_a256'
CHECKPOINT_GEN = "generator.pth"
CHECKPOINT_CRITIC = "critic.pth"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
SAVE_MODEL = True
LOAD_MODEL = False
LEARNING_RATE = 1e-3
BATCH_SIZES = [32, 32, 32, 16, 16, 16, 16, 8, 4]
CHANNELS_IMG = 3
Z_DIM = 256
IN_CHANNELS = 256
CRITIC_ITERATIONS = 1
LAMBDA_GP = 10
PROGRESSIVE_EPOCHS = [30] * len(BATCH_SIZES)
FIXED_NOISE = torch.randn(8, Z_DIM, 1, 1).to(DEVICE)
NUM_WORKERS = 4
```

Figura 89: Configuración inicial e hiperparametros

Si han leído mis previas implementaciones les será mucho más fácil entender este código. Básicamente definimos la dimensión inicial de las imágenes en la que comenzará el entrenamiento 128x128, donde se generarán los checkpoint, la gpu cuda en device, las flags **Save\_MODEL** y **LOAD\_MODEL**, que nos indicarán si guardaremos los modelos o los cargaremos.

A continuación, decidí una tasa de aprendizaje de 1x0.001 para el entrenamiento la cual funciona de manera estable. Definimos una lista de diferentes tamaños de lotes **BATCH SIZES**. El batch size disminuye a medida que aumentamos progresivamente la resolución de las imágenes para ajustarnos a la memoria de la GPU y no sobreexplotar los recursos.

Indicamos el número de canales de las imágenes a 3 porque son imágenes RGB, definimos **Z DIM**=256 (dimensión del vector aleatorio de ruido ). Con **IN CHANNELS=256** definimos el número de canales de entrada para la primera capa del generador. **CRITIC ITERATIONS = 1 (Critical Steps)**, indicamos las iteraciones del crítico por cada iteración del Generador. Definimos el hyperparameter  $\lambda=10$  (**LAMBDA GP=10**) para el cálculo de gradiente, punto **4.2.2**.

En **PROGRESSIVE\_EPOCHS = [30] \* len(BATCH SIZES)** configuramos una lista que determina cuantas épocas se entrenará a cada fase de progresión basado en los tamaños de lote definidos anteriormente. **Fixed Noise** inicializamos el tensor de ruido aleatorio de la misma manera que en otros códigos implementados anteriormente.

Por último los **NUM WORKERS = 4**, aquí indico el número de procesos secundarios para cargar los datos, cuantos más número de workers mayor rapidez de los datos, sin embargo debes ajustarlo según los recursos de tu ordenador, en mi caso 4 funciono bien.

### 9.3.5 Arquitectura (Model.py)

En model.py, programamos la normalización de pixeles, tasa de aprendizaje ecualizada, generador, discriminador(crítico), bloque convolucional, función de fusión (**fade\_in**) y el minibatch\_std.

#### WSConv2d (Convolucion con escalado de Pesos o tasa de aprendizaje ecualizada)

```

factors = [1, 1, 1, 1, 1 / 2, 1 / 4, 1 / 8, 1 / 16, 1 / 32]

class WSConv2d(nn.Module):
    """Inspirado en :
    https://github.com/nvnbny/progressive_growing_of_gans/blob/master/modelUtils.py
    """
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1, gain=2):
        super(WSConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
        self.scale = (gain / (in_channels * (kernel_size ** 2))) ** 0.5
        self.bias = self.conv.bias
        self.conv.bias = None
        nn.init.normal_(self.conv.weight)
        nn.init.zeros_(self.bias)

    def forward(self, x):
        return self.conv(x * self.scale) + self.bias.view(1, self.bias.shape[0], 1, 1)

```

Figura 90: Código clase WSConv2d

El escalado de pesos es una técnica introducida en Progressive GANas para normalizar la magnitud de los pesos en las capas convolucionales. El objetivo es compensar el hecho de que la magnitud de los gradientes puede variar considerablemente entre diferentes capas de la red. En el punto teórico **4.3.6.** Como vemos en el código escalamos los pesos de la convolución por la raíz de  $(\text{gain} / (\text{in channels} * (\text{kernel\_size} ** 2))) ** 0.5$  asegurando que todas las capas tengan aproximadamente la misma magnitud en sus actualizaciones durante el entrenamiento, esto es importantísimo para estabilizar las redes profundas.

$$\text{Peso ajustado} = \text{Peso} \cdot \sqrt{\frac{2}{k*k}}$$

$k*k$  = Tamaño del kernel/filtro

$$\sqrt{\frac{2}{k*k}} = \text{Desviación estándar de los pesos}$$

En el paso **forward** multiplicamos la entrada por la escala calculada y sumamos el sesgo ajustado **self.bias**. Este método nos asegura que el efecto de escalado se aplique uniformemente, y es más eficiente computacionalmente que reescalar los pesos directamente

```
def forward(self, x):
    return self.conv(x * self.scale) + self.bias.view(1, self.bias.shape[0], 1, 1)
```

Figura 91: Método Forward WSConv2d

### Normalización de Pixelles

```
class PixelNorm(nn.Module):
    def __init__(self):
        super(PixelNorm, self).__init__()
        self.epsilon = 1e-8

    def forward(self, x):
        return x / torch.sqrt(torch.mean(x ** 2, dim=1, keepdim=True) + self.epsilon)
```

Figura 92: Normalización de Pixelles código

Con **PixelNorm** normalizamos las características de la imagen a nivel individual de cada píxel, en lugar de normalizar mediante mini batch (**Batch Normalization**) como hemos hecho anteriormente en otras GANS. Recordemos que el epsilon es una pequeña cantidad

para evitar la división por 0.

El código realmente es adoptar la fórmula expuesta en el punto 4.3.5 :

$$b_{xy} = \frac{a_{xy}}{\sqrt{\frac{1}{N} \sum_{j=0}^{N-1} (a_{x,y}^j)^2 + \epsilon}}$$

Calculamos la raíz cuadrada del promedio del cuadrado de las activaciones y divide las activaciones por este valor, añadiendo un pequeño término epsilon para evitar la división por cero. Con esta capa/layer prevenimos el escalamiento descontrolado de las activaciones en la red.

### Bloque Convolucional

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, use_pixelnorm=True):
        super(ConvBlock, self).__init__()
        self.use_pn = use_pixelnorm
        self.conv1 = WSConv2d(in_channels, out_channels)
        self.conv2 = WSConv2d(out_channels, out_channels)
        self.leaky = nn.LeakyReLU(0.2)
        self.pn = PixelNorm()

    def forward(self, x):
        x = self.leaky(self.conv1(x))
        x = self.pn(x) if self.use_pn else x
        x = self.leaky(self.conv2(x))
        x = self.pn(x) if self.use_pn else x
        return x
```

Figura 93: Bloque Convolucional código

Aquí creamos la clase de bloque convolucional que usaremos en el entrenamiento, compuesto por 2 capas **WSConv2d**, **PixelNorm**, y utilizando la capa de activación(Leaky RELU)entre ambas, esto nos facilita el aprendizaje de características y la estabilización del flujo de gradientes.

## Generador

```

class Generator(nn.Module):
    def __init__(self, z_dim, in_channels, img_channels=3):
        super(Generator, self).__init__()
        self.initial = nn.Sequential([
            PixelNorm(),
            nn.ConvTranspose2d(z_dim, in_channels, 4, 1, 0),
            nn.LeakyReLU(0.2),
            WSConv2d(in_channels, in_channels, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            PixelNorm(),
        ])
        self.initial_rgb = WSConv2d(in_channels, img_channels, kernel_size=1, stride=1, padding=0)
        self.prog_blocks, self.rgb_layers = (
            nn.ModuleList([]),
            nn.ModuleList([self.initial_rgb]),
        )
    for i in range(len(factors) - 1):
        conv_in_c = int(in_channels * factors[i])
        conv_out_c = int(in_channels * factors[i + 1])
        self.prog_blocks.append(ConvBlock(conv_in_c, conv_out_c))
        self.rgb_layers.append(
            WSConv2d(conv_out_c, img_channels, kernel_size=1, stride=1, padding=0)
        )

    def fade_in(self, alpha, upscaled, generated):
        return torch.tanh(alpha * generated + (1 - alpha) * upscaled)

    def forward(self, x, alpha, steps):
        out = self.initial(x)
        if steps == 0:
            return self.initial_rgb(out)
        for step in range(steps):
            upscaled = F.interpolate(out, scale_factor=2, mode="nearest")
            out = self.prog_blocks[step](upscaled)
            final_upscaled = self.rgb_layers[steps - 1](upscaled)
            final_out = self.rgb_layers[steps](out)
        return self.fade_in(alpha, final_upscaled, final_out)

```

Figura 94: Clase Generador implementación Pro-GAN

Veamos el generador, inicializamos el constructor del generador `__init__` que recibe el vector aleatorio de ruido **z dim**, el **in channels**:número de canales de entrada y los canales de la imagen de salida RGB,**img channels**.

En la línea 51, tenemos el bloque inicial :

```

self.initial = nn.Sequential(
    PixelNorm(),
    nn.ConvTranspose2d(z_dim, in_channels, 4, 1, 0),
    nn.LeakyReLU(0.2),
    WSConv2d(in_channels, in_channels, kernel_size=3, stride=1, padding=1),
    nn.LeakyReLU(0.2),
    PixelNorm(),
)

```

Figura 95: Bloque Inicial Generador Pro-GAN

Normalizamos los píxeles para estabilizar las características **PixelNorm()**, aplicamos una capa de convolución traspuesta **nn.ConvTranspose2d(z\_dim, in\_channels, 4, 1, 0)**, para aumentar la resolución de la imagen. Después de la convolución traspuesta, aplicamos una función de activación **Leaky ReLU** con un coeficiente de pendiente de **0.2** para introducir no linealidad en la red. A continuación, realizamos la convolución con escalado de pesos (**W Conv 2d**) que mantiene constante la magnitud de las activaciones, seguida de otra activación **LeakyReLU**. Finalmente, usamos otra normalización de píxeles para estabilizar aún más las características.

En la línea 59 aplicamos la capa RGB

```
self.initial_rgb = WSConv2d(in_channels, img_channels, kernel_size=1, stride=1, padding=0)
```

Figura 96: Aplicación Capa Rgb codigo

convertimos el mapa de características inicial en una imagen RGB utilizando una convolución de un solo canal (**WSConv2d**).

A continuación he implementado los bloques progresivos en la línea 60:

```

self.prog_blocks, self.rgb_layers = (
    nn.ModuleList([]),
    nn.ModuleList([self.initial_rgb]),
)
for i in range(len(factors) - 1):
    conv_in_c = int(in_channels * factors[i])
    conv_out_c = int(in_channels * factors[i + 1])
    self.prog_blocks.append(ConvBlock(conv_in_c, conv_out_c))
    self.rgb_layers.append(
        WSConv2d(conv_out_c, img_channels, kernel_size=1, stride=1, padding=0)
    )

```

Figura 97: Bloques Progresivos

Los bloques progresivos (**self.prog\_blocks**) y las capas RGB (**self.rgb layers**) se configuran para manejar la generación de imágenes a resoluciones crecientes. Inicializo dos listas de módulos (**nn.Module List**), una para los bloques progresivos y otra para las capas RGB. En el

bucle, recorremos los factores de escalado que he definido previamente en la lista de factores. Por cada iteración, se calcula **conv\_in\_c** y **conv\_out\_c**, que son los números de canales de entrada y salida ajustados por el factor de escalado correspondiente. Añadimos bloques convolucionales (**Con Block**) a **prog blocks** para incrementar la resolución y se añaden capas **WSConv2d** a **rgb layers** para convertir los mapas de características en imágenes RGB en cada resolución.

Ahora hacemos la función de **fade\_in** (**transición** punto 4.3.3) para mezclar las resoluciones tal como en el esquema ():

$$\text{Interpolated output} = \text{OldLayer} * (1 - \alpha) + \text{NewLayer} * \alpha$$

```
def fade_in(self, alpha, upscaled, generated):
    return torch.tanh(alpha * generated + (1 - alpha) * upscaled)
```

Figura 98: Formula *fade\_in* en el código Pro-GAN

Mezclamos suavemente entre dos resoluciones de imagen durante el entrenamiento progresivo. En esta función como en la fórmula tomamos un parámetro **alpha**, que es un factor de mezcla entre 0 y 1, y dos imágenes: **upscaled**, es la imagen interpolada desde una resolución inferior, y **generated**, que es la imagen generada en la resolución actual. La mezcla de estas dos imágenes se realiza utilizando una combinación ponderada de **alpha \* generated** y **(1 - alpha) \* upscaled**. Por último, aplica la función **torch.tanh** para normalizar las salidas en el rango [-1, 1].

Por último en el método **forward** del generador explicamos el proceso de cómo el generador toma un vector de ruido y produce una imagen :

```
75     def forward(self, x, alpha, steps):
76         out = self.initial(x)
77         if steps == 0:
78             return self.initial_rgb(out)
79         for step in range(steps):
80             upscaled = F.interpolate(out, scale_factor=2, mode="nearest")
81             out = self.prog_blocks[step](upscaled)
82             final_upscaled = self.rgb_layers[steps - 1](upscaled)
83             final_out = self.rgb_layers[steps](out)
84             return self.fade_in(alpha, final_upscaled, final_out)
```

Figura 99: Función *forward* del generador en PRO-Gan

## Parámetros del Método

**x:** El tensor de entrada, que es un vector latente de baja dimensión.

**alpha:** Un parámetro de mezcla que controla la transición entre resoluciones durante el entrenamiento progresivo. Varía entre 0 y 1.

**steps:** Indica el número de pasos de escalado que se han aplicado hasta el momento, determinando la resolución actual de la imagen generada.

Primero en la línea 76, el vector latente x pasa a través del bloque inicial del generador anterior.

Luego si el steps es igual a 0, el generador está en la resolución inicial (la más baja). En este caso, el mapa de características inicial se convierte directamente en una imagen RGB mediante la capa **initial\_rgb**. Si steps es mayor que 0, el generador debe incrementar progresivamente la resolución de la imagen en múltiples etapas. dentro de un bucle for que gira sobre el número de pasos de escalado que hemos especificado. En cada iteración del bucle:

1. Interpolación(**F.interpolate**): El mapa de características **out** se escala a una resolución superior mediante la interpolación con el método/modelo "nearest".
2. Bloque Progresivo: El mapa de características escalado pasa a través del bloque progresivo correspondiente (**self.prog\_blocks[step]**), donde aplicamos una serie de convoluciones y activaciones para refinar los detalles de la imagen a la nueva resolución.

Después de procesar a través de los bloques progresivos generamos dos versiones rgb

1. **final upscaled:** Lo obtenemos aplicando la capa RGB correspondiente a la resolución anterior (**steps - 1**) al mapa de características escalado (**upscaled**).
2. **final\_out:** Lo obtenemos aplicando la capa RGB correspondiente a la resolución actual (**steps**) al mapa de características procesado (**out**)

Por último devolvemos la mezcla de las imágenes RGB mediante la función **fade in**.

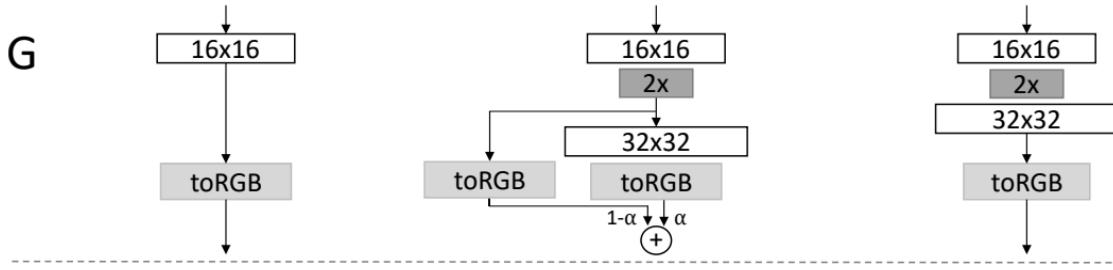


Figura 100: Esquema General del funcionamiento del código del Generador en Pro-GAN

## Discriminador

```

class Discriminator(nn.Module):
    def __init__(self, z_dim, in_channels, img_channels=3):
        super(Discriminator, self).__init__()
        self.prog_blocks, self.rgb_layers = nn.ModuleList([]), nn.ModuleList([])
        self.leaky = nn.LeakyReLU(0.2)
        for i in range(len(factors) - 1, 0, -1):
            conv_in = int(in_channels * factors[i])
            conv_out = int(in_channels * factors[i - 1])
            self.prog_blocks.append(ConvBlock(conv_in, conv_out, use_pixelnorm=False))
            self.rgb_layers.append(
                WSConv2d(img_channels, conv_in, kernel_size=1, stride=1, padding=0)
            )
        self.initial_rgb = WSConv2d(img_channels, in_channels, kernel_size=1, stride=1, padding=0)
        self.rgb_layers.append(self.initial_rgb)
        self.avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)
        self.final_block = nn.Sequential(
            WSConv2d(in_channels + 1, in_channels, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2),
            WSConv2d(in_channels, in_channels, kernel_size=4, padding=0, stride=1),
            nn.LeakyReLU(0.2),
            WSConv2d(in_channels, 1, kernel_size=1, padding=0, stride=1),
        )

    def fade_in(self, alpha, downsampled, out):
        return alpha * out + (1 - alpha) * downsampled

    def minibatch_std(self, x):
        batch_statistics = torch.std(x, dim=0).mean().repeat(x.shape[0], 1, x.shape[2], x.shape[3])
        return torch.cat([x, batch_statistics], dim=1)

    def forward(self, x, alpha, steps):
        cur_step = len(self.prog_blocks) - steps
        out = self.leaky(self.rgb_layers[cur_step](x))
        if steps == 0:
            out = self.minibatch_std(out)
            return self.final_block(out).view(out.shape[0], -1)
        downsampled = self.leaky(self.rgb_layers[cur_step + 1](self.avg_pool(x)))
        out = self.avg_pool(self.prog_blocks[cur_step](out))
        out = self.fade_in(alpha, downsampled, out)
        for step in range(cur_step + 1, len(self.prog_blocks)):
            out = self.prog_blocks[step](out)
            out = self.avg_pool(out)
        out = self.minibatch_std(out)
        return self.final_block(out).view(out.shape[0], -1)

```

Figura 101: Código del Discriminador en implementación Pro-GAN

Empecemos por el constructor

```

87     def __init__(self, z_dim, in_channels, img_channels=3):
88         super(Discriminator, self).__init__()
89         self.prog_blocks, self.rgb_layers = nn.ModuleList([]), nn.ModuleList([])
90         self.leaky = nn.LeakyReLU(0.2)
91         for i in range(len(factors) - 1, 0, -1):
92             conv_in = int(in_channels * factors[i])
93             conv_out = int(in_channels * factors[i - 1])
94             self.prog_blocks.append(ConvBlock(conv_in, conv_out, use_pixelnorm=False))
95             self.rgb_layers.append(
96                 WSConv2d(img_channels, conv_in, kernel_size=1, stride=1, padding=0)
97             )
98         self.initial_rgb = WSConv2d(img_channels, in_channels, kernel_size=1, stride=1, padding=0)
99         self.rgb_layers.append(self.initial_rgb)
100        self.avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)
101        self.final_block = nn.Sequential(
102            WSConv2d(in_channels + 1, in_channels, kernel_size=3, padding=1),
103            nn.LeakyReLU(0.2),
104            WSConv2d(in_channels, in_channels, kernel_size=4, padding=0, stride=1),
105            nn.LeakyReLU(0.2),
106            WSConv2d(in_channels, 1, kernel_size=1, padding=0, stride=1),
107        )

```

Figura 102: Constructor Discriminador Pro-GAN

Escribimos los parámetros que recibe en este caso el discriminador/crítico `in_channels` e `img_channels`. Inicializamos los bloques progresivos y las capas `rgb`, utilizamos los bloques progresivos para reducir gradualmente la resolución de las imágenes de entrada. Seguimos con el bucle `for` del constructor `__init__`:

```

def __init__(self, z_dim, in_channels, img_channels=3):
    for i in range(len(factors) - 1, 0, -1):
        conv_in = int(in_channels * factors[i])
        conv_out = int(in_channels * factors[i - 1])
        self.prog_blocks.append(ConvBlock(conv_in, conv_out, use_pixelnorm=False))
        self.rgb_layers.append(
            WSConv2d(img_channels, conv_in, kernel_size=1, stride=1, padding=0)
        )

```

Figura 103: Cálculo de canales de entrada y salida código Pro-GAN para el discriminador

El bucle se gira sobre los factores de escalado(lista) en orden inverso, empezando desde el último factor hasta el primero. Para cada iteración calculamos `conv_in` y `conv_out` como el número de canales de entrada y salida ajustados por el factor de escalado correspondiente.

Añadimos un bloque de convolución (**Conv Block**) a **prog\_blocks**. En Estos bloques no utilizamos PixelNorm, ya que la normalización de características no es tan crítica en el discriminador como en el generador.

Añadimos una capa **WSConv2d** a **rgb\_layers** para convertir las imágenes RGB en mapas de características con **conv\_in canales**.

Salimos del bucle y definimos la capa inicial RGB y el Pooling medio para reducir la resolución de las imágenes.

```
|     )
self.initial_rgb = WSConv2d(img_channels, in_channels, kernel_size=1, stride=1, padding=0)
self.rgb_layers.append(self.initial_rgb)
self.avg_pool = nn.AvgPool2d(kernel_size=2, stride=2)
```

*Figura 104: Definición de capa inicial RGB y el average pooling en el discriminador*

Por último seguimos en el constructor, línea 101, definimos el bloque final (**self.final\_block**).

```
self.final_block = nn.Sequential(
    WSConv2d(in_channels + 1, in_channels, kernel_size=3, padding=1),
    nn.LeakyReLU(0.2),
    WSConv2d(in_channels, in_channels, kernel_size=4, padding=0, stride=1),
    nn.LeakyReLU(0.2),
    WSConv2d(in_channels, 1, kernel_size=1, padding=0, stride=1),
)
```

*Figura 105: Instancia del proceso del bloque final para el Discriminador*

Incluimos varias capas Con 2d con funciones de activación Leaky ReLU entre ellas, y una capa final que reduce las características a una sola salida.

Ya finalizado el constructor creamos la función de mezcla de resoluciones **fade\_in** mezclando entre dos resoluciones de imagen durante el entrenamiento progresivo.

```
def fade_in(self, alpha, downscaled, out):
    return alpha * out + (1 - alpha) * down
```

*Figura 106: Formula fade\_in Discriminador*

Acto seguido creamos la función minibatch std (desviación estándar del mini lote) de la manera explicada en el punto 4.3.4, agregamos un canal que representa la desviación estándar a través del mini batch que ayuda al discriminador a detectar estadísticas anómalas en las imágenes generadas.

```
def minibatch_std(self, x):
    batch_statistics = torch.std(x, dim=0).mean().repeat(x.shape[0], 1, x.shape[2], x.shape[3])
    return torch.cat([x, batch_statistics], dim=1)
```

Figura 107: Código desviación estándar mini batch Pro-GAN

Por último el método **forward** donde he definido como el discriminador va a procesar una imagen (fake/real) a través de las capas(layers) y las distintas clases

```
def forward(self, x, alpha, steps):
    cur_step = len(self.prog_blocks) - steps
    out = self.leaky(self.rgb_layers[cur_step](x))
    if steps == 0:
        out = self.minibatch_std(out)
        return self.final_block(out).view(out.shape[0], -1)
    downscaled = self.leaky(self.rgb_layers[cur_step + 1](self.avg_pool(x)))
    out = self.avg_pool(self.prog_blocks[cur_step](out))
    out = self.fade_in(alpha, downscaled, out)
    for step in range(cur_step + 1, len(self.prog_blocks)):
        out = self.prog_blocks[step](out)
        out = self.avg_pool(out)
    out = self.minibatch_std(out)
    return self.final_block(out).view(out.shape[0], -1)
```

Figura 108: Método Forward o proceso del Discriminador para implementacion Pro-GAN

**cur\_step(Paso actual)** lo calculamos restando steps del número total de bloques progresivos por tanto esto determina el bloque actual en el que se encuentra el discriminador.

Acto seguido la imagen de entrada **x** se convierte en un mapa de características mediante la capa RGB correspondiente (**self.rgb\_layers[cur\_step]**) y le aplica la activación **LeakyReLU**. Si **steps** es 0, el discriminador está en la resolución inicial más baja. En este caso entonces, aplicamos la desviación estándar(**std**) del mini batch (**minibatch\_std**) y luego pasa a través del bloque final (**self.final block**)(linea 121), que produce la salida final aplanada(es decir en flat layer).

¿Que pasa si los **steps** son mayores que 0? Bien aplicamos el escalado progresivo, el discriminador procesa la imagen a través de múltiples etapas:

Primeramente en la línea 122 y 123 la imagen de entrada la reducimos en resolución mediante **avg\_pool** y la procesamos mediante **rgb layers** en la línea 122. Luego, se pasa a través del bloque progresivo correspondiente (**prog blocks**), reduciendo aún más la resolución y refinando las características. Luego llamamos a la función **fade\_in** e iteramos sobre los

bloques de la línea **125 al 127** aplicando reducción de resolución(**avg\_pool**) y refinamiento de características en cada paso(**prog blocks**).

Después de procesar todas las etapas progresivas, aplicamos la desviación estándar del **mini batch** y luego el bloque final para producir la salida final **aplanada(flatten)** del discriminador.

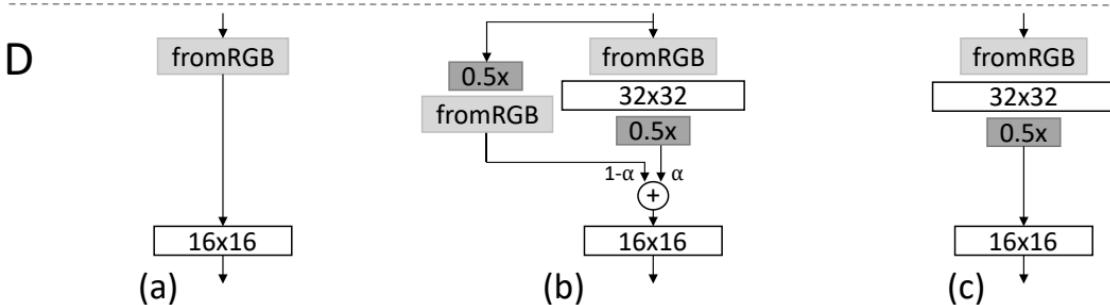


Figura 109: Esquema General código discriminador Pro-GAN

### Test

Por último hago una pequeña prueba/test para comprobar que el modelo y la arquitectura del generador y discriminador funcionan perfectamente en distintos tamaños de imágenes hasta 1024 x 1024.

```
if __name__ == "__main__":
    Z_DIM = 100
    IN_CHANNELS = 256
    gen = Generator(Z_DIM, IN_CHANNELS, img_channels=3)
    critic = Discriminator(Z_DIM, IN_CHANNELS, img_channels=3)

    for img_size in [4, 8, 16, 32, 64, 128, 256, 512, 1024]:
        num_steps = int(log2(img_size / 4))
        x = torch.randn((1, Z_DIM, 1, 1))
        z = gen(x, 0.5, steps=num_steps)
        assert z.shape == (1, 3, img_size, img_size)
        out = critic(z, alpha=0.5, steps=num_steps)
        assert out.shape == (1, 1)
        print(f"Success! At img size: {img_size}")
```

Figura 110: Test de modelos Pro-GAN

La ejecutamos.... y perfectamente vemos que ha compilado la arquitectura de los modelos para la red degenerativa **progresiva (PROGAN)**:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Success! At img size: 8
Success! At img size: 16
Success! At img size: 32
Success! At img size: 64
Success! At img size: 128
Success! At img size: 256
Success! At img size: 512
Success! At img size: 1024
PS C:\PROGANCELEBA>

```

*Figura 111: Resultado Test de los modelos de implementacion Pro-GAN*

Ahora explicare como he definido el entrenamiento **train.py**

### 9.3.6 Utils.py

En utils.py almaceno funciones auxiliares que son un pilar fundamental del entrenamiento las cuales las llamaremos posteriormente en train.py.

Como siempre lo primero los importes que vamos a necesitar son los imports

```

1 import torch
2 import random
3 import numpy as np
4 import os
5 import torchvision
6 import torch.nn as nn
7 import config
8 from torchvision.utils import save_image
9 from scipy.stats import truncnorm

```

*Figura 112: Import de librerias en Utils.py*

y acto seguido empezamos con las funciones, primero **plot\_to\_tensorboard**:

```

def plot_to_tensorboard(writer, loss_critic, loss_gen, real, fake, tensorboard_step):
    writer.add_scalar("Loss Critic", loss_critic, global_step=tensorboard_step)
    with torch.no_grad():
        img_grid_real = torchvision.utils.make_grid(real[:8], normalize=True)
        img_grid_fake = torchvision.utils.make_grid(fake[:8], normalize=True)
        writer.add_image("Real", img_grid_real, global_step=tensorboard_step)
        writer.add_image("Fake", img_grid_fake, global_step=tensorboard_step)

```

*Figura 113: Funcion para mostrar las gráficas en Tensorboard*

La función **plot to tensorboard** la utilzo como el nombre indica para registrar las pérdidas y las imágenes generadas en TensorBoard. La función recibe el escritor de TensorBoard (**writer**), las pérdidas del discriminador (**loss critic**) y del generador (**loss gen**), así como las imágenes reales (**real**) y falsas (**fake**), junto con el paso actual de Tensor Board

**(tensorboard\_step)**. Primero, la función registra las pérdidas del discriminador y del generador. Luego, utilizando torch.no\_grad(), asegurando que las siguientes operaciones no afecten el cálculo de gradientes. Las imágenes reales y falsas se contienen en cuadrículas(grids) utilizando **touchvision.utils.make grid**, y estas cuadrículas/grids se añaden a Tensor Board. Así podemos visualizar las imágenes mientras se van generando

Luego tenemos la función de cálculo de penalidad de gradiente que será muy similar a WGAN-GP punto 4.2.2:

```

19 def gradient_penalty(critic, real, fake, alpha, train_step, device="cpu"):
20     BATCH_SIZE, C, H, W = real.shape
21     beta = torch.rand((BATCH_SIZE, 1, 1, 1)).repeat(1, C, H, W).to(device)
22     interpolated_images = real * beta + fake.detach() * (1 - beta)
23     interpolated_images.requires_grad_(True)
24     mixed_scores = critic(interpolated_images, alpha, train_step)
25     gradient = torch.autograd.grad(
26         inputs=interpolated_images,
27         outputs=mixed_scores,
28         grad_outputs=torch.ones_like(mixed_scores),
29         create_graph=True,
30         retain_graph=True,
31     )[0]
32     gradient = gradient.view(gradient.shape[0], -1)
33     gradient_norm = gradient.norm(2, dim=1)
34     gradient_penalty = torch.mean((gradient_norm - 1) ** 2)
35     return gradient_penalty

```

Figura 114: Penalidad del Gradiente código para implementación Pro-GAN

$\hat{x}$ = muestras interpoladas

$$\hat{x} = \alpha x + (1 - \alpha) \bar{x}$$

$$\text{gradient\_penalty} = (\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2$$

Tomamos como parámetros el discriminador (**critic**), las imágenes **reales (real)** y **falsas (fake)**, **alpha**, el paso de entrenamiento (**train step**) y la GPU (**device**). Primero, se generan imágenes interpoladas entre las reales y las falsas usando un valor aleatorio beta como visto en la teoría. Estas imágenes interpoladas requieren gradientes. Luego, el discriminador evalúa las imágenes interpoladas y se calculan los gradientes con respecto a estas imágenes. La norma de los gradientes se ajusta para asegurar que esté cerca de 1, y la penalización de gradiente se calcula como la media del cuadrado de la diferencia entre la norma del gradiente y 1.

$$\text{gradient\_penalty} = (\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2$$

Luego tenemos la función de save checkpoint que guarda el estado del modelo y del optimizado en el archivo que le indicamos y la de load checkpoint que nos cargará los modelos(generador o crítico):

```
def save_checkpoint(model, optimizer, filename="my_checkpoint.pth.tar"):
    print("=> Saving checkpoint")
    checkpoint = {
        "state_dict": model.state_dict(),
        "optimizer": optimizer.state_dict(),
    }
    torch.save(checkpoint, filename)

def load_checkpoint(checkpoint_file, model, optimizer, lr):
    print("=> Loading checkpoint")
    checkpoint = torch.load(checkpoint_file, map_location="cuda")
    model.load_state_dict(checkpoint["state_dict"])
    optimizer.load_state_dict(checkpoint["optimizer"])
    for param_group in optimizer.param_groups:
        param_group["lr"] = lr
```

Figura 115: Métodos de guarda y carga del checkpoint

Por último la función **generate examples** para generar las imágenes durante el entrenamiento y que se nos almacenen en el path especificado:

```
def generate_examples(gen, steps, truncation=0.7, n=100):
    gen.eval()
    alpha = 1.0
    for i in range(n):
        with torch.no_grad():
            noise = torch.tensor(truncnorm.rvs(-truncation, truncation, size=(1, config.Z_DIM, 1, 1)), device=config.DEVICE, dtype=torch.float32)
            img = gen(noise, alpha, steps)
            save_image(img*0.5+0.5, f"saved_examples/img_{i}.png")
    gen.train()
```

Figura 116: Método para generar las imágenes y almacenarlas en el path

A continuacion vamos a observar el entrenamiento en Train.py

### 9.3.7 Train.py(Entrenamiento)

En train.py programamos el entrenamiento de la PROGAN.

Primero importamos todas las bibliotecas y módulos que necesitamos

```
import torch
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
► Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter
from utils import (
    gradient_penalty,
    plot_to_tensorboard,
    save_checkpoint,
    load_checkpoint,
    generate_examples,
)
from model import Discriminator, Generator
from math import log2
from tqdm import tqdm
import config

torch.backends.cudnn.benchmarks = True
```

*Figura 120: Imports para Train.Py implementacion Pro-Gan*

A parte de las bibliotecas de python podemos, importar la configuración **config.py** y **utils.py** con sus respectivos métodos.

Ahora definimos la función **get loader**, para cargar las imágenes de diferentes resoluciones y transformarlas para el entrenamiento.

Primero, la función define una serie de transformaciones que aplicamos a las imágenes del dataset. Utilizamos **transforms.Compose** para combinar varias transformaciones. En la primera transformación, **transforms.Resize**, cambiamos el tamaño de cada imagen al tamaño especificado por el parámetro **image size**. Luego, **transforms.ToTensor** convierte las imágenes en tensores PyTorch para que las imágenes puedan ser procesadas por la red neuronal.

A continuación, **transforms.RandomHorizontalFlip** aplica una inversión horizontal aleatoria a las imágenes con una probabilidad de 0.5. Es decir aquí estamos aplicando **data augmentation** para mejorar la robustez del modelo al introducir variaciones adicionales en los

datos de entrenamiento. Por último, con transforms.Normalize normalizar las imágenes para que los valores de los píxeles estén en el rango [-1, 1], restando 0.5 y dividiendo por 0.5.

```

21  def get_loader(image_size):
22      transform = transforms.Compose(
23          [
24              transforms.Resize((image_size, image_size)),
25              transforms.ToTensor(),
26              transforms.RandomHorizontalFlip(p=0.5),
27              transforms.Normalize(
28                  [0.5 for _ in range(config.CHANNELS_IMG)],
29                  [0.5 for _ in range(config.CHANNELS_IMG)],
30              ),
31          ]
32      )
33      batch_size = config.BATCH_SIZES[int(log2(image_size / 4))]
34      dataset = datasets.ImageFolder(root=config.DATASET, transform=transform)
35      loader = DataLoader(
36          dataset,
37          batch_size=batch_size,
38          shuffle=True,
39          num_workers=config.NUM_WORKERS,
40          pin_memory=True,
41      )
42      return loader, dataset

```

Figura 121: Preprocesamiento y carga de los datos, implementación Pro-GAN

En la línea 33 calculamos el tamaño del lote (**batch\_size**) basándonos en el tamaño de la imagen. Utilizamos la lista predefinida en config.py de tamaños de lote (**config.BATCH\_SIZES**) y selecciona el tamaño apropiado para la resolución actual. Esto se logra calculando el índice correspondiente en la lista mediante **log2(image size / 4)**. Por ejemplo, si **image\_size** es **128**, entonces **log2(128 / 4)** es 5, lo que selecciona el tamaño de lote correspondiente en la lista el cual sería en este caso 16 de tamaño de bloque, recordemos **BATCH SIZES = [32, 32, 32, 16, 16, 16, 8, 4]**

En la línea 34 creamos el dataset según la ruta del dataset especificada en **config.py**, en mi caso será **DATASET = 'celeb\_a256'**.

Finalmente, creó un DataLoader tomamos como parámetros el dataset, el tamaño de lote, **shuffle=True**, mezclando los datos en cada época para el entrenamiento, **num workers=config.NUM WORKERS**, y **pin memory=True**, que hace que los datos se carguen directamente a la GPU.

Finalmente devolvemos el dataset ya preprocessado.

### Train\_fn

Ahora vamos a ver la función train fn donde realizó principalmente la optimización y las funciones de pérdida tanto del discriminador como del generador.

```

44 . def train_fn(
45     critic,
46     gen,
47     loader,
48     dataset,
49     step,
50     alpha,
51     opt_critic,
52     opt_gen,
53     tensorboard_step,
54     writer,
55     scaler_gen,
56     scaler_critic,
57 ):
58     loop = tqdm(loader, leave=True)
59     for batch_idx, (real, _) in enumerate(loop):
60         real = real.to(config.DEVICE)
61         cur_batch_size = real.shape[0]
62         noise = torch.randn(cur_batch_size, config.Z_DIM, 1, 1).to(config.DEVICE)
63
64         with torch.cuda.amp.autocast():
65             fake = gen(noise, alpha, step)
66             critic_real = critic(real, alpha, step)
67             critic_fake = critic(fake.detach(), alpha, step)
68             gp = gradient_penalty(critic, real, fake, alpha, step, device=config.DEVICE)
69             loss_critic = (
70                 -(torch.mean(critic_real) - torch.mean(critic_fake))
71                 + config.LAMBDA_GP * gp
72                 + (0.001 * torch.mean(critic_real ** 2))
73             )
74
75             opt_critic.zero_grad()
76             scaler_critic.scale(loss_critic).backward()
77             scaler_critic.step(opt_critic)
78             scaler_critic.update()
79
80         with torch.cuda.amp.autocast():
81             gen_fake = critic(fake, alpha, step)
82             loss_gen = -torch.mean(gen_fake)
83
84             opt_gen.zero_grad()
85             scaler_gen.scale(loss_gen).backward()
86             scaler_gen.step(opt_gen)
87             scaler_gen.update()
88
89             alpha += cur_batch_size / (
90                 config.PROGRESSIVE_EPOCHS[step] * 0.5) * len(dataset)
91
92             alpha = min(alpha, 1)
93
94             if batch_idx % 500 == 0:
95                 with torch.no_grad():
96                     fixed_fakes = gen(config.FIXED_NOISE, alpha, step) * 0.5 + 0.5
97                     plot_to_tensorboard(
98                         writer,
99                         loss_critic.item(),
100                         loss_gen.item(),
101                         real.detach(),
102                         fixed_fakes.detach(),
103                         tensorboard_step,
104                     )
105                     tensorboard_step += 1
106
107             loop.set_postfix(
108                 gp=gp.item(),
109                 loss_critic=loss_critic.item(),
110             )
111
112     return tensorboard_step, alpha

```

Figura 122: Funcionamiento Train Fn

Primeramente de la línea 44 a la 57 especificamos cuales son los parámetros que va a recibir

- **critic**: El discriminador del modelo.
- **gen**: El generador del modelo.
- **Loader**: El DataLoader que proporciona los lotes de imágenes.
- **dataset**: El conjunto de datos que se está utilizando.
- **step**: El número de pasos de escalado que se han aplicado, determinando la resolución actual.
- **alpha**: El parámetro de mezcla que controla la transición entre resoluciones.
- **opt\_critic**: El optimizador para el discriminador.
- **option**: El optimizador para el generador.
- **tensorboard\_step**: Contador de pasos para el registro en TensorBoard.
- **Writer**: El escritor de Tensor Board para registrar las métricas.
- **scaler gen**: Escalador de gradientes para el generador
- **scaler critic**: Escalador de gradientes para el discriminador

En la línea 58 comenzamos el ciclo de entrenamiento:

```

59     for batch_idx, (real, _) in enumerate(loop):
60         real = real.to(config.DEVICE)
61         cur_batch_size = real.shape[0]
62         noise = torch.randn(cur_batch_size, config.Z_DIM, 1, 1).to(config.DEVICE)
63
64         with torch.cuda.amp.autocast():
65             fake = gen(noise, alpha, step)
66             critic_real = critic(real, alpha, step)
67             critic_fake = critic(fake.detach(), alpha, step)
68             gp = gradient_penalty(critic, real, fake, alpha, step, device=config.DEVICE)
69             loss_critic = (
70                 -(torch.mean(critic_real) - torch.mean(critic_fake))
71                 + config.LAMBDA_GP * gp
72                 + (0.001 * torch.mean(critic_real)**2)
73             )

```

*Figura 123:Fragmento de Código comienzo ciclo entrenamiento*

utilizamos **tqdm** para poder ver durante el entrenamiento una barra de progreso visual.

Dentro del bucle, las imágenes reales se envían a la GPU y se genera un vector de ruido aleatorio para cada imagen del lote. Luego, dentro de un contexto de **obtenemos que st** para AMP (precisión mixta automática), el generador crea imágenes falsas a partir del ruido. El discriminador evalúa tanto las imágenes reales como las falsas en la línea 66 y 67.

Dicho esto calculamos la pérdida del crítico en este caso vamos a utilizar la función de pérdida que utiliza **WGAN-GP** ya que es la más efectiva.

```

69     loss_critic = (
70         -(torch.mean(critic_real) - torch.mean(critic_fake))
71         + config.LAMBDA_GP * gp
72         + (0.001 * torch.mean(critic_real ** 2))
73     )

```

Figura 124: Código función de perdida del crítico Pro-GAN

$$E_{z \sim p_Z}[D(G(z))] - E_{x \sim p_X}[D(x)] + \lambda E_{\hat{x} \sim p_{\hat{X}}}(\left\| \Delta_x D(\hat{x}) \right\|_2 - 1)^2$$

La pérdida del discriminador (**loss critic**) incluye tres componentes:

1. La diferencia entre la media de las evaluaciones de las imágenes reales y falsas.
2. La penalización de gradiente (**gradient penalty**) para mejorar la estabilidad del entrenamiento.
3. Un pequeño término de regularización para evitar que los valores del discriminador se descontrolen.

Optimizamos el crítico:

```

74
75     opt_critic.zero_grad()
76     scaler_critic.scale(loss_critic).backward()
77     scaler_critic.step(opt_critic)
78     scaler_critic.update()
79

```

Figura 125: Optimización, retropropagación, y actualización del crítico.

se realiza la retropropagación y la actualización de los pesos del discriminador utilizando **scaler\_critic** (veremos posteriormente que es lo que hace en cuestión).

Ahora la línea 80 a la 87, calculamos la pérdida del generador y realizamos la optimización de este.

```

80     with torch.cuda.amp.autocast():
81         gen_fake = critic(fake, alpha, step)
82         loss_gen = -torch.mean(gen_fake)
83
84         opt_gen.zero_grad()
85         scaler_gen.scale(loss_gen).backward()
86         scaler_gen.step(opt_gen)
87         scaler_gen.update()

```

*Figura 126: Optimización y función de pérdida del generador, código Pro-GAN*

muy similar a el discriminador, excepto la función de pérdida que será  $-\mathbf{D}(\mathbf{G}(\mathbf{z}))$ .

Actualizamos alpha, el valor que se va incrementando para la interpolación entre resoluciones:

```

89     alpha += cur_batch_size / (
90         (config.PROGRESSIVE_EPOCHS[step] * 0.5) * len(dataset)
91     )
92     alpha = min(alpha, 1)

```

*Figura 127: Cálculo de alpha, código Pro-GAN.*

Una vez guardadas las métricas las pasamos a tensorboard para poder visualizarlas y monitorearlas

```

94     if batch_idx % 500 == 0:
95         with torch.no_grad():
96             fixed_fakes = gen(config.FIXED_NOISE, alpha, step) * 0.5 + 0.5
97             plot_to_tensorboard(
98                 writer,
99                 loss_critic.item(),
100                loss_gen.item(),
101                real.detach(),
102                fixed_fakes.detach(),
103                tensorboard_step,
104            )
105            tensorboard_step += 1
106

```

*Figura 128. Llamada función plot\_to\_tensorboard*

## Main

```

def main():
    gen = Generator(
        config.Z_DIM, config.IN_CHANNELS, img_channels=config.CHANNELS_IMG
    ).to(config.DEVICE)
    critic = Discriminator(
        config.Z_DIM, config.IN_CHANNELS, img_channels=config.CHANNELS_IMG
    ).to(config.DEVICE)
    opt_gen = optim.Adam(gen.parameters(), lr=config.LEARNING_RATE, betas=(0.0, 0.99))
    opt_critic = optim.Adam(
        critic.parameters(), lr=config.LEARNING_RATE, betas=(0.0, 0.99)
    )
    scaler_critic = torch.cuda.amp.GradScaler()
    scaler_gen = torch.cuda.amp.GradScaler()
    writer = SummaryWriter(f"logs/gan1")

    if config.LOAD_MODEL:
        load_checkpoint(
            config.CHECKPOINT_GEN, gen, opt_gen, config.LEARNING_RATE,
        )
        load_checkpoint(
            config.CHECKPOINT_CRITIC, critic, opt_critic, config.LEARNING_RATE,
        )

    gen.train()
    critic.train()
    tensorboard_step = 0
    step = int(log2(config.START_TRAIN_AT_IMG_SIZE / 4))
    for num_epochs in config.PROGRESSIVE_EPOCHS[step:]:
        alpha = 1e-5
        loader, dataset = get_loader(4 * 2 ** step)
        print(F"Current image size: {4 * 2 ** step}")

        for epoch in range(num_epochs):
            print(F"Epoch [{epoch+1}/{num_epochs}]")
            tensorboard_step, alpha = train_fn(
                critic,
                gen,
                loader,
                dataset,
                step,
                alpha,
                opt_critic,
                opt_gen,
                tensorboard_step,
                writer,
                scaler_gen,
                scaler_critic,
            )

            if config.SAVE_MODEL:
                save_checkpoint(gen, opt_gen, filename=config.CHECKPOINT_GEN)
                save_checkpoint(critic, opt_critic, filename=config.CHECKPOINT_CRITIC)

        step += 1

    if __name__ == "__main__":
        main()

```

Figura 129: Funcion Main implementación Pro-GAN

En la función Main inicializamos los modelos,optimizadores, escaladores de gradientes,

registro de Tensorboard y la carga de checkpoints en caso de que queramos cargar el modelo.

```

114     def main():
115         gen = Generator(
116             config.Z_DIM, config.IN_CHANNELS, img_channels=config.CHANNELS_IMG
117         ).to(config.DEVICE)
118         critic = Discriminator(
119             config.Z_DIM, config.IN_CHANNELS, img_channels=config.CHANNELS_IMG
120         ).to(config.DEVICE)
121         opt_gen = optim.Adam(gen.parameters(), lr=config.LEARNING_RATE, betas=(0.0, 0.99))
122         opt_critic = optim.Adam(
123             critic.parameters(), lr=config.LEARNING_RATE, betas=(0.0, 0.99)
124         )
125         scaler_critic = torch.cuda.amp.GradScaler()
126         scaler_gen = torch.cuda.amp.GradScaler()
127         writer = SummaryWriter(f"logs/gan1")
128
129     if config.LOAD_MODEL:
130         load_checkpoint(
131             config.CHECKPOINT_GEN, gen, opt_gen, config.LEARNING_RATE,
132         )
133         load_checkpoint(
134             config.CHECKPOINT_CRITIC, critic, opt_critic, config.LEARNING_RATE,
135         )
136
137     gen.train()
138     critic.train()

```

Figura 130: Fragmento 1 de código del Main

Primero inicializamos e instanciamos los modelos generador y crítico/discriminador de la línea 115 a la 120. A continuación configuramos los optimizadores Adam para ambos modelos, con la tasa de aprendizaje y los parámetros beta especificados en la configuración. Además, se inicializan los escaladores de gradientes para la precisión mixta automática (AMP). Los escaladores de gradiente que proporciona pytorch y cuda nos ayudan a manejar los gradientes y nos da una precisión mixta. En el writer especificamos para tensor board donde queremos que se guardan los logs de las métricas

Posteriormente de la línea 129 a 135, cargamos los checkpoints de los modelos del generador y del crítico en caso de que queramos es decir LOAD\_MODEL=TRUE. Ahora llamamos a **train()** del generador como del discriminador.

Por ultimo:

```

141     for num_epochs in config.PROGRESSIVE_EPOCHS[step:]:
142         alpha = 1e-5
143         loader, dataset = get_loader(4 * 2 ** step)
144         print(f"Current image size: {4 * 2 ** step}")
145
146         for epoch in range(num_epochs):
147             print(f"Epoch [{epoch+1}/{num_epochs}]")
148             tensorboard_step, alpha = train_fn(
149                 critic,
150                 gen,
151                 loader,
152                 dataset,
153                 step,
154                 alpha,
155                 opt_critic,
156                 opt_gen,
157                 (variable) scaler_gen: GradScaler
158
159                 scaler_gen,
160                 scaler_critic,
161             )
162
163             if config.SAVE_MODEL:
164                 save_checkpoint(gen, opt_gen, filename=config.CHECKPOINT_GEN)
165                 save_checkpoint(critic, opt_critic, filename=config.CHECKPOINT_CRITIC)
166
167             step += 1
168
169     if __name__ == "__main__":
170         main()

```

Figura 131: Fragmento 2 de código del Main

Iniciamos el bucle de entrenamiento de resolución progresivo determinando el paso inicial basado en la resolución de inicio (**START\_TRAIN\_AT\_IMG\_SIZE**) en nuestro caso **128**. Luego, se itera sobre las resoluciones especificadas en **PROGRESSIVE\_EPOCHS** en nuestro caso serán **[30] \* len(BATCH\_SIZES)** como definido en **config.py**.

En la línea 146 dentro del for de entrenamiento progresivo de resolución, hacemos otro for anidado un entrenamiento anidado por épocas, en cada época para realizar el entrenamiento llamamos a la función **train\_fn()** explicada anteriormente. Por último guardamos los modelos en checkpoints, y en la **Línea 170** ejecutamos finalmente el entrenamiento

### 9.3.4 Análisis Resultados

Tras 3 días de entrenamientos hemos conseguido generar caras a partir de simple ruido, a los siguientes resultados:

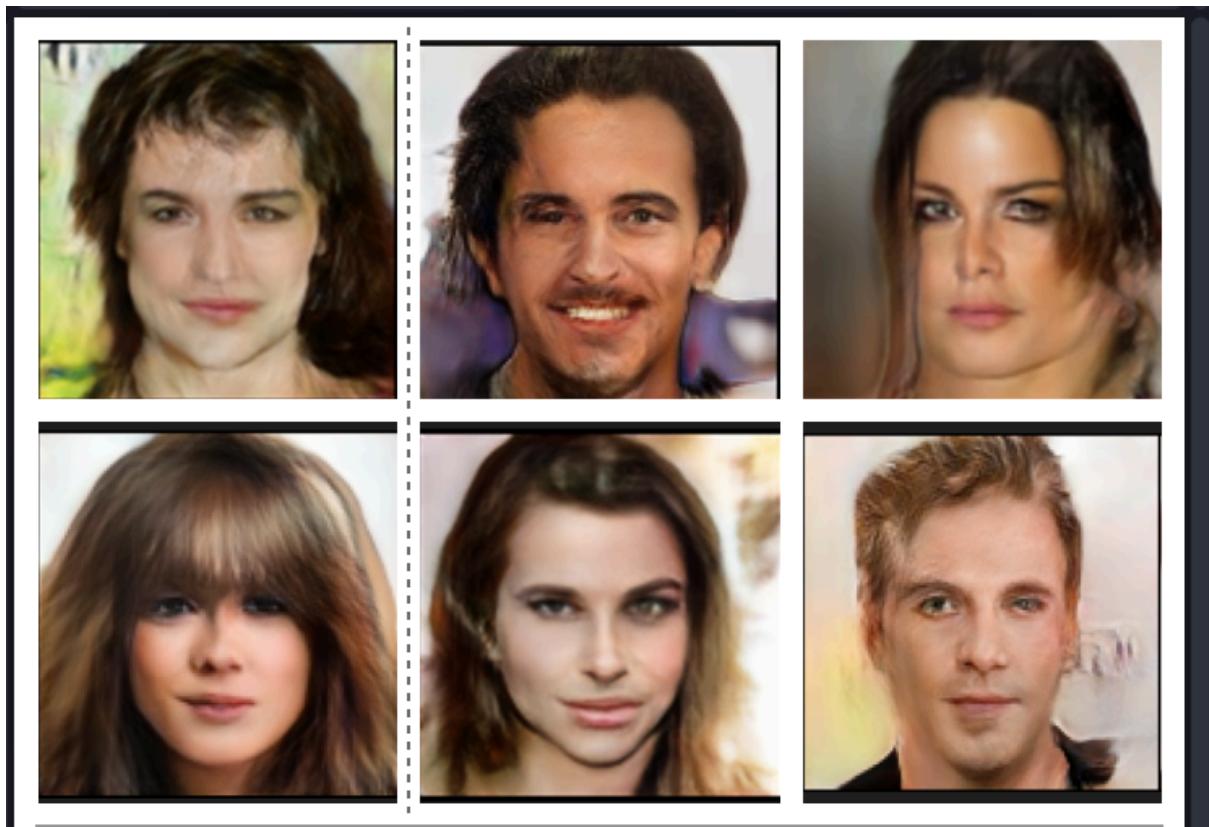


Figura 132: Resultados implementación Pro-GAN

Es verdaderamente impresionante el potencial que tienen las redes generativas adversariales y en especial las progresivas, nos han generado distintas caras humanas de distintas variedad entre ellas, es cierto que se pueden encontrar minúsculos fallos, pero con seguridad con más días de entrenamiento y más requisito computacional como podría tener una empresa los resultados serían incluso mejores. Este algoritmo lo hemos utilizado para caras pero podríamos utilizarlo perfectamente con el dataset de cerebros realizado en la implementación técnica WGAN-GP, o con otros conjuntos de datos como piezas industriales o huesos, cualquiera.

## 10. MEJORA DE MODELOS

---

Una vez ya creadas nuestras redes generativas, y haber generado imágenes completamente artificiales, debemos comprobar a ver si la introducción de estos ejemplos adversariales generados mejorarán la precisión de otros modelos como Clasificadores de imágenes (CNN).

Es importante subrayar que los ejemplos aplicados para la siguiente mejora de modelos, son generados mediante la técnica y código WGAN-GP del punto **6.2**. Según el dataset que utilizaremos la implementación de este código **6.2** cambiara simplemente los hiper parámetros los paths (directorios ), y la arquitectura del generador o discriminador según los tamaños de imágenes que queramos procesar y generar del nuevo dataset, el entrenamiento que usaremos WGAN-GP será exactamente el mismo que para el ejemplo de los cerebros en el punto **6.2**.

### **10.1 ENTORNO Y HERRAMIENTAS**

Todos los modelos a mejorar durante este proyecto de fin de grado, se diseñarán en el entorno collab, utilizando la GPU A100 de Nvidia que nos asegurará un correcto rendimiento de los modelos. En los modelos utilizaremos las librerías de tensor flow y keras para el entrenamiento del modelo.

#### **10.1.1 EfficientNet**

Efficient Net es un método para escalar modelos CNN, con escalar modelos CNN nos referimos a la modificar el tamaño de la red para mejorar el rendimiento, podemos escalar en tres dimensiones la red neuronal ;

**Escalado de Profundidad ( $\alpha$ ):** Aumento de número de capas

**Escalado de ancho( $\beta$ ):** Aumentar el número de canales en cada capa

**Escalado de resolución de entrada( $\gamma$ ) :** Aumenta la resolución de las imágenes de entrada.

EfficientNet escala modelos CNN mediante el método llamado Compound Scaling que quiere decir que En lugar de escalar solo una dimensión del modelo (como profundidad o ancho), EfficientNet escala todas las dimensiones simultáneamente pero de manera balanceada. La idea clave es usar los factores de escala  $\alpha, \beta$  y  $\gamma$ .

EfficientNet define el escalado compuesto mediante la siguiente fórmula o relación:

$$\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$

El objetivo es mantener un equilibrio entre el incremento de profundidad, ancho y resolución para maximizar la eficiencia del modelo.

Dados los factores de escalado  $\phi$ ,  $\alpha$ ,  $\beta$ ,  $\gamma$  se definen las dimensiones escaladas de la siguiente forma

**Profundidad Escalada** =  $d = \alpha^\phi$

**Ancho Escalado** =  $w = \beta^\phi$

resolución Escalada :  $r = \gamma^\phi$

$\phi$  es un hiper parámetro que controla el alcance del estado global, los valores típicos son  $\alpha = 1.2$ ,  $\beta = 1.1$  y  $\gamma = 1.15$ . En el caso de EfficientnetB3 el que utilizaremos en un modelo posterior  $\phi = 3$

En resumen EfficientNet combina lo mejor de las técnicas de escalado, búsqueda de arquitectura neural y optimización de hiper parámetros para ofrecer una solución robusta en el dominio de las CNN

### 10.1.2 Regularización L1/L2

La regularización L1 y L2 son técnicas esenciales para evitar el sobreajuste en modelos de aprendizaje profundo, incluidas las redes neuronales convolucionales (CNN). Estas técnicas penalizan los pesos del modelo durante el entrenamiento para fomentar soluciones más generalizables.

La regularización L1 agrega una penalización basada en la suma de los valores absolutos de los pesos. En el contexto de CNN, esto puede fomentar la esparsidad en los filtros convolucionales, lo que significa que algunos de los filtros pueden volverse exactamente cero, ayudando a seleccionar características más relevantes.

La función de pérdida con regularización L1 se define como:

$$F_{L1} = F(y, \hat{y}) + \lambda \sum_{i=1}^n |w_i|$$

$F(y, \hat{y})$  es la función de pérdida del cnn original como la de entropía cruzada,  $\lambda$  el hiper parámetro que pasamos manualmente en el código este controla la fuerza de la penalización L1 en este caso  $y w_i$  como de costumbre son los pesos

Ahora, la regularización L2 agrega una penalización basada en la suma de los cuadrados de los pesos. Esto tiende a reducir el tamaño de los pesos sin hacer que sean exactamente cero, distribuyendo la carga entre características correlacionadas y reduciendo la varianza del modelo.

La función de pérdida con regularización L2 se define como:

$$F_{L2} = F(y, \hat{y}) + \lambda \sum_{i=1}^n w_i^2$$

## 10.2 MEJORA DE CLASIFICADOR DE TUMORES CEREBRALES

A continuación trataremos de mejorar un CNN que identifica si un cerebro tiene o no un tumor añadiendo los ejemplos adversariales generados en **6.2**.

### 10.2.1 CNN

Mediante el Categorical Cross Entropy clasificamos las imágenes como lo haría el discriminador a través de capas convolucionales.

### 10.2.2 Preparación de Datos

La preparación de datos dependerá de cada dataset, en este caso la estructura de este dataset está definida en el punto **6.2.1**.

Para mezclar los ejemplos generados, tras entrenar y generar la WGAN-GP, descargamos los ejemplos generados.Como hemos dicho anteriormente el dataset **Brain MRI Images for Brain Tumor Detection** contiene 253 imágenes reales, 98 cerebros sin tumor y 155 cerebros con tumor. Lo que faremos sera aumentar este dataset x2 con las imágenes generadas.

Mezclamos 98 imágenes generadas sin tumor, dándonos un total de 196 imágenes(generadas y reales ) sin tumor, y hacemos lo mismo con los cerebros con tumor dándonos un total de 310 imágenes.

La carpeta mixta en caso de este dataset queda así:

/no

 47 no	Archivo JPG	6 KB	No
 48 no	Archivo JPEG	5 KB	No
 49 no	Archivo JPG	6 KB	No
 50 no	Archivo JPG	7 KB	No
 epoch_15_image_1	Archivo PNG	35 KB	No
 epoch_15_image_2	Archivo PNG	33 KB	No
 epoch_15_image_3	Archivo PNG	28 KB	No
 epoch_15_image_4	Archivo PNG	35 KB	No
 epoch_15_image_5	Archivo PNG	28 KB	No
 epoch_15_image_6	Archivo PNG	13 KB	No
 epoch_15_image_7	Archivo PNG	33 KB	No
 epoch_15_image_8	Archivo PNG	32 KB	No

/yes

 epoch_20_image_14	Archivo PNG	33 KB	No
 epoch_20_image_15	Archivo PNG	32 KB	No
 epoch_20_image_16	Archivo PNG	32 KB	No
 epoch_20_image_17	Archivo PNG	32 KB	No
 epoch_20_image_18	Archivo PNG	35 KB	No
 Y1	Archivo JPG	8 KB	No
 Y2	Archivo JPG	8 KB	No
—			

Figura 133: Estructura Dataset Mezclado

Hacemos zip y subimos el dataset con los ejemplos adversariales introducidos en el modelo CNN en este caso en Google Collab.Una vez subido lo unzipeamos

```
 !unzip /content/Brains.zip
```

Figura 134: Unzip mix Dataset

### 10.2.3 Clasificador de Cerebros

Empecemos con la implementación técnica de este clasificador.

Añadimos las librerías que necesitamos mediante los imports:

```
[ ] # import system libs
import os
import time
import shutil
import pathlib
import itertools
from PIL import Image

# import data handling tools
import cv2
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style('darkgrid')
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

# import Deep learning Libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam, Adamax
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout, BatchNormalization
from tensorflow.keras import regularizers

# Ignore Warnings
import warnings
warnings.filterwarnings("ignore")

print ('modules loaded')
```

Figura 135: Imports CNN cerebros

Después iteramos sobre los directorio y inicializamos las etiquetas

```

| # Generate data paths with labels
| data_dir = '/content/Brains'
| filepaths = []
| labels = []

| folds = os.listdir(data_dir)
| for fold in folds:
|     foldpath = os.path.join(data_dir, fold)
|     filelist = os.listdir(foldpath)
|     for file in filelist:
|         fpath = os.path.join(foldpath, file)

|         filepaths.append(fpath)
|         labels.append(fold)

# Concatenate data paths with labels into one dataframe
Fseries = pd.Series(filepaths, name= 'filepaths')
Lseries = pd.Series(labels, name='labels')
df = pd.concat([Fseries, Lseries], axis= 1)

```

Figura 136: Definición de labels y directorios CNN cerebros

Definimos la variable **data\_dir** que apunta al directorio donde se almacena las imágenes. Inicializamos dos listas vacías, **filepaths** y **labels**, para almacenar las rutas de los archivos y las etiquetas correspondientes.

Se lista el contenido del directorio **data\_dir**, que contiene subdirectorios correspondientes a las etiquetas yes(Tumor), no (Sin tumor). Para cada subdirectorío (**fold**), construimos la ruta completa (**foldpath**). Luego, listamos el contenido de cada subdirectorío (**filelist**), que contiene las imágenes. Para cada archivo, construimos la ruta completa (**fpath**), que se agrega a la lista **filepaths**. La etiqueta correspondiente (**fold**) la agregamos a la lista **labels**.

Creamos dos series **Fseries,Lseries** con la librería pandas a partir de los directorios y las etiquetas, **filepaths** y **labels**. Por último concatenamos **df** con las series de directorios y etiquetas.

El dataframe quedaría tal que así:

	filepaths	labels
0	/content/Brains/no/N21.jpg	no
1	/content/Brains/no/40 no.jpg	no
2	/content/Brains/no/25 no.jpg	no
3	/content/Brains/no/No14.jpg	no
4	/content/Brains/no/23 no.jpg	no
...	...	...
248	/content/Brains/yes/Y77.jpg	yes
249	/content/Brains/yes/Y51.jpg	yes
250	/content/Brains/yes/Y13.jpg	yes
251	/content/Brains/yes/Y195.JPG	yes
252	/content/Brains/yes/Y246.JPG	yes

253 rows x 2 columns

Figura 137: Actualización de estructura de datos

### Preprocesamiento de Datos

Dividir los datos en **train(entrenamiento)**, **test(validación)** en un modelo de clasificación de imágenes es indispensable para asegurar que utilicemos datos que no han sido vistos durante el entrenamiento de prueba para evaluar el rendimiento del modelo. Para que sean imágenes “nuevas” (las del test) las clasificadas para evaluar el rendimiento del modelo.

```
strat = df['labels']
train_df, test_df = train_test_split(df, train_size= 0.8, shuffle= True, random_state= 123, stratify= strat)
```

Figura 138: Fragmento I código preprocesamiento de datos, división test y train

La variable **strat** la definimos para contener las etiquetas (**labels**) del DataFrame. Las etiquetas se utilizaremos para la división/estratificación, asegurando que ambos conjuntos (**entrenamiento** y **prueba**) tengan una proporción similar de cada clase.

**train\_test\_split:** Esta función divide el DataFrame **df** en dos subconjuntos: **train\_df** (**entrenamiento**) y **test\_df** (**prueba**).

**train\_size=0.8:** Especifica que el **80%** de los datos se utilizarán para el entrenamiento, mientras que el **20%** restante se utilizarán para la prueba.

**shuffle=True:** Indica que los datos deben ser mezclados antes de dividirse. Esto es importante para asegurar que la estratificación sea efectiva.

**random\_state=123:** Fija una semilla para el generador de números aleatorios, asegurándonos que la división sea reproducible.

**stratify=strat:** Utilizamos las etiquetas para estratificar la división, asegurando que ambos conjuntos mantengan la misma proporción de clases.

En la función **train\_test\_split** dividimos el DataFrame en dos subconjuntos: uno para entrenamiento (**train\_df**) y otro para prueba (**test\_df**), manteniendo la proporción de clases en ambos conjuntos mediante la estratificación.

A continuación

```
batch_size = 8
img_size = (224, 224)
channels = 3
img_shape = (img_size[0], img_size[1], channels)

tr_gen = ImageDataGenerator()
ts_gen = ImageDataGenerator()

train_gen = tr_gen.flow_from_dataframe( train_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                         color_mode= 'rgb', shuffle= True, batch_size= batch_size)

test_gen = ts_gen.flow_from_dataframe( test_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                         color_mode= 'rgb', shuffle= False, batch_size= batch_size)
```

Figura 139 : Fragmento 2 de código preprocesamiento de datos CNN

Primero, definimos el tamaño del lote (**batch\_size**), el tamaño de las imágenes (**img\_size**) **224x224**, y el número de canales (**channels**), **3** **rgb**. La variable **img\_shape** combina estos parámetros para definir la forma de las imágenes, después utilizamos **ImageDataGenerator()** de la librería keras, que nos servirá para cargar y procesar las imágenes en lotes.

Después en **train\_gen** configuramos cargar las imágenes desde el data frame de imágenes de entrenamiento **train\_df**. Especificamos las columnas que contienen las rutas de las imágenes (**x\_col='filepaths'**) y las etiquetas (**y\_col='labels'**). Las imágenes se redimensionan al tamaño definido por **img\_size**, se cargan en modo RGB, y las etiquetas se codifican como categorías. Además, se mezclan aleatoriamente en cada época (**shuffle=True**).

Del mismo modo, el generador de prueba (**test\_gen**) se configura para cargar las imágenes desde el DataFrame **test\_df**. Las configuraciones son similares a las del generador de entrenamiento, pero las imágenes no se mezclan (**shuffle=False**), así preservamos el orden original para la evaluación.

Vamos a visualizar las imágenes del entrenamiento (**train\_gen**):

```

g_dict = train_gen.class_indices      # definimos diccionario
clases = list(g_dict.keys())
images, labels = next(train_gen)      # Obtener las imágenes y etiquetas

plt.figure(figsize= (20, 10))

for i in range(8):
    plt.subplot(2, 4, i + 1)
    image = images[i] / 255           # Escalamos las imágenes para visualizarlos (0 - 255)
    plt.imshow(image)
    index = np.argmax(labels[i])
    class_name = clases[index]       # Obtener etiqueta
    plt.title(class_name, color= 'blue', fontsize= 12)
    plt.axis('off')
plt.show()

```

Figura 140: Código plot para visualizar los datos del CNN

Primero, definimos un diccionario que mapea las clases a índices numéricos. Luego, se obtienen las claves de este diccionario, que representan las etiquetas de las clases. Despues obtenemos las imágenes y las etiquetas del generador de entrenamiento.Y creamos el plot.

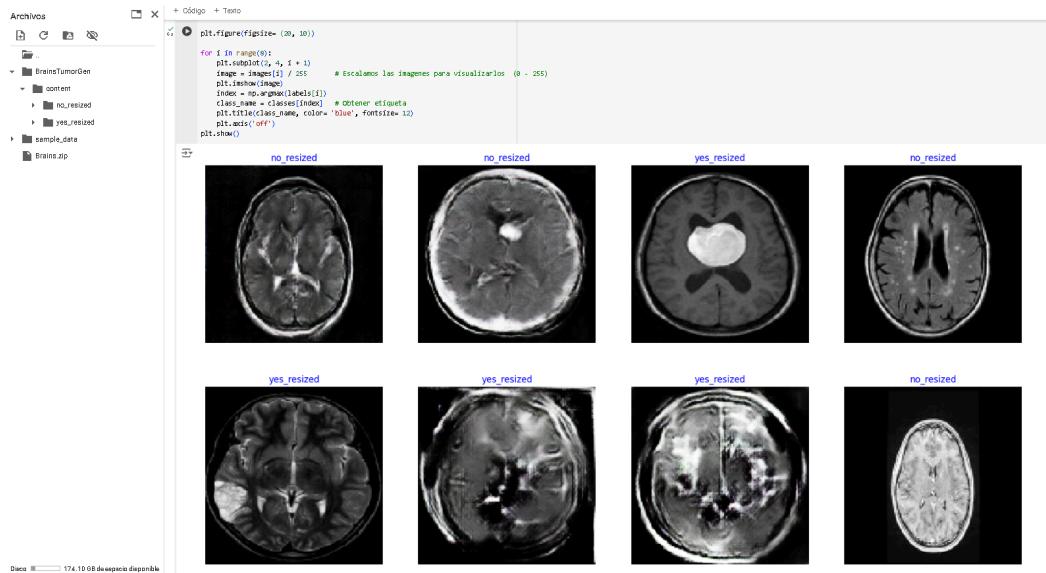


Figura 141: Muestra de Datos del Dataset

## Estructura Del Modelo

Vamos a explicar la estructura del modelo utilizando una **CNN** basada en **EfficientNet**.

```
# Estructura del modelo
img_size = (224, 224)
channels = 3
img_shape = (img_size[0], img_size[1], channels)
class_count = len(list(train_gen.class_indices.keys())) # Definir numero de clases

# creando un modelo preentrenado efficientnet
base_model = tf.keras.applications.efficientnet.EfficientNetB3(include_top= False, weights= "imagenet",
                                                               input_shape= img_shape, pooling= 'max')

model = Sequential([
    base_model,
    BatchNormalization(axis= -1, momentum= 0.99, epsilon= 0.001),
    Dense(256, kernel_regularizer= regularizers.l2(l= 0.016), activity_regularizer= regularizers.l1(0.006),
          bias_regularizer= regularizers.l1(0.006), activation= 'relu'),
    Dropout(rate= 0.45, seed= 123),
    Dense(class_count, activation= 'softmax')
])

model.compile(Adamax(learning_rate= 0.001), loss= 'categorical_crossentropy', metrics= ['accuracy'])

model.summary()
```

Figura 142: Estructura del Modelo CNN basado en EfficientNet

Primero definimos los parámetros el tamaño de la imagen (**img\_size**), el número de canales (**channels**) y la forma de la imagen (**img\_shape**). También contamos el número de clases en el conjunto de datos de entrenamiento para definir la capa densa que usaremos al final del modelo.

Despues en **base\_model = tf.keras.applications.efficientnet.EfficientNetB3(include\_top=False, weights= "imagenet", input\_shape= img\_shape, pooling= 'max')**, creamos el modelo preentrenado **EfficientNet**, cargamos EfficientNetB3, modelo preentrenado en ImageNet, sin incluir su capa superior (include\_top=False).Luego definimos el modelo base secuencial de keras,clasificador que funciona parecido a un crítico.

```
model = Sequential([
    base_model,
    BatchNormalization(axis= -1, momentum= 0.99, epsilon= 0.001),
    Dense(256, kernel_regularizer= regularizers.l2(l= 0.016), activity_regularizer= regularizers.l1(0.006),
          bias_regularizer= regularizers.l1(0.006), activation= 'relu'),
    Dropout(rate= 0.45, seed= 123),
    Dense(class_count, activation= 'softmax')
])
```

Figura 143: Procesos modelo secuencial CNN

En el modelo, normalizamos la salida del modelo base con *Batch Normalization*,

luego añadimos una capa densa de 256 neuronas(**3.6.2**), utilizando la regularización L1 y L2 para evitar el sobreajuste, y la capa de activación **ReLU**. A continuación aplicamos un dropout del 45 % de las neuronas, y por último aplicamos una capa de salida densa con tantas neuronas como clases, y utilizamos la capa de activación **softmax para dar dos salidas** para determinar si es cerebro con tumor o sin.

```
model.compile(Adamax(learning_rate= 0.001), loss= 'categorical_crossentropy', metrics= ['accuracy'])

model.summary()
```

*Figura 144: Compilación de modelo CNN, y summary(resumen)*

Por último compilamos el modelo, con el optimizador **ADAM** con una tasa de aprendizaje 0.001, con una función de pérdida **categorical\_crossentropy** y pasamos la métrica **accuracy** para evaluar el modelo posteriormente. Por último imprimimos el resumen del modelo.

```
✓ Downloading data from https://storage.googleapis.com/keras-applications/efficientnetb3\_notop.h5
43941136/43941136 [=====] - 3s 0us/step
Model: "sequential"

Layer (type)                 Output Shape              Param #
=====
efficientnetb3 (Functional)  (None, 1536)           10783535
)
batch_normalization (Batch   (None, 1536)           6144
Normalization)

dense (Dense)                (None, 256)            393472
dropout (Dropout)             (None, 256)            0
dense_1 (Dense)               (None, 2)              514
=====
Total params: 11183665 (42.66 MB)
Trainable params: 11093290 (42.32 MB)
Non-trainable params: 90375 (353.03 KB)
```

*Figura 145: Resumen Estructura modelo CNN Cerebros*

### Ejecución Del Entrenamiento

En la siguiente celda ejecutamos el entrenamiento:

```
epochs = 30  # numero de épocas

history = model.fit(x= train_gen, epochs= epochs, verbose= 1, validation_data= test_gen,
                     validation_steps= None, shuffle= False)
```

```

Epoch 15/30
47/47 [=====] - 4s 79ms/step - loss: 2.4173 - accuracy: 0.9733 - val_loss: 2.3956 - val_accuracy: 0.94
Epoch 14/30
47/47 [=====] - 4s 79ms/step - loss: 2.3220 - accuracy: 0.9547 - val_loss: 2.2087 - val_accuracy: 0.96
Epoch 15/30
47/47 [=====] - 4s 80ms/step - loss: 2.1056 - accuracy: 0.9680 - val_loss: 2.0902 - val_accuracy: 0.94
Epoch 16/30
47/47 [=====] - 4s 80ms/step - loss: 1.9350 - accuracy: 0.9893 - val_loss: 1.9378 - val_accuracy: 0.93
Epoch 17/30

```

*Figura 146: Ejecucion entrenamiento clasificador CNN de cerebros enfermos*

#### 10.2.4 Resultados y Conclusiones

##### Codigo Para visualizar los Resultados

En este caso no estamos utilizando tensorboard, y vamos a utilizar la librería **matplotlib**, para crear los plots donde visualizaremos las métricas, que nos permitirán evaluar si el modelo con el añadido imágenes/datos generados por entrenamiento adversarial rinde mejor, que el modelo entrenado con simplemente los datos originales.

```

# Definimos las variables para los graficos
tr_acc = history.history['accuracy']
tr_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
index_loss = np.argmin(val_loss)
val_lowest = val_loss[index_loss]
index_acc = np.argmax(val_acc)
acc_highest = val_acc[index_acc]
Epochs = [i+1 for i in range(len(tr_acc))]
loss_label = f'best epoch= {str(index_loss + 1)}'
acc_label = f'best epoch= {str(index_acc + 1)}'

# Plot training history
plt.figure(figsize= (20, 8))
plt.style.use('fivethirtyeight')

plt.subplot(1, 2, 1)
plt.plot(Epochs, tr_loss, 'r', label= 'Training loss')
plt.plot(Epochs, val_loss, 'g', label= 'Validation loss')
plt.scatter(index_loss + 1, val_lowest, s= 150, c= 'blue', label= loss_label)
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(Epochs, tr_acc, 'r', label= 'Training Accuracy')
plt.plot(Epochs, val_acc, 'g', label= 'Validation Accuracy')
plt.scatter(index_acc + 1, acc_highest, s= 150, c= 'blue', label= acc_label)
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout
plt.show()

```

Figura 147: Código de las gráficas para visualización de métricas del entrenamiento CNN

Como observamos en el código previo mostraremos una gráfica que muestre la pérdida durante el entrenamiento del entrenamiento y otra que nos muestre la precisión del modelo durante el entrenamiento.

A continuación muestro el código de la matriz de confusión(**Confusion matrix**), donde veremos de los datos de test que se han puesto a prueba para el clasificador cuántos de ellos ha acertado y cuántas ha fallado.

Por ultimo

```
train_score = model.evaluate(train_gen, verbose= 1)
test_score = model.evaluate(test_gen, verbose= 1)

print("Train Loss: ", train_score[0])
print("Train Accuracy: ", train_score[1])
print('-' * 20)
print("Test Loss: ", test_score[0])
print("Test Accuracy: ", test_score[1])
```

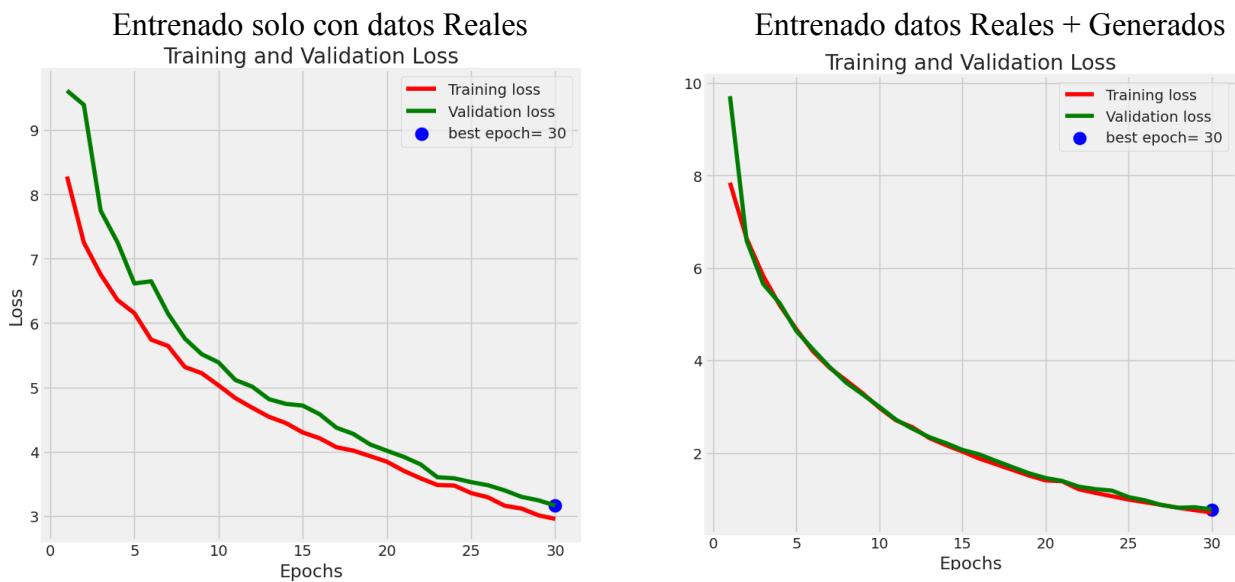
*Figura 148: Código para la muestra de las métricas finales*

Mostraremos las métricas finales y las compararemos entre ambos modelos (mejorado con datos adversariales vs original )

## ANALISIS RESULTADOS

Tras entrenar un modelo entrenado con los datos reales, y el mismo modelo entrenado con la ampliación de los datos adversariales, veamos si verdaderamente esta inclusión de las imágenes generadas por WGAN-GP generan una mejora en el modelo.

Respecto a las graficas: Analizemos Primero la grafica de la pérdida



*Figura 149: Comparación de gráficas de pérdida, datos Reales vs datos R+Generados*

A simple vista podemos ver que la pérdida del modelo mejorado con cerebros generados por

GANS mejora considerablemente la pérdida durante el entrenamiento, analizemoslo en mas profundidad

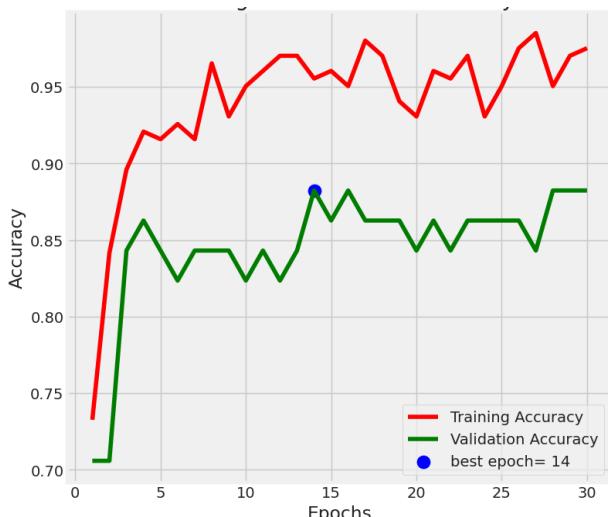
Podemos observar que el modelo entrenado solo con datos originales contiene valores de pérdida mayores durante el entrenamiento en comparación con el modelo mejorado. Es decir el modelo mejorado es más rápido y consistente, esto es debido a que gracias a los imágenes/cerebros generados por GANS el modelo contiene mucha más diversidad en los datos memorando así la habilidad del modelo para generalizar y clasificar los cerebros.

En la última época el modelo entrenado únicamente con datos reales tiene una pérdida final alrededor de 3, mientras que el modelo con datos adicionales generados por GANs alcanza una pérdida final mucho más baja, alrededor de 1.5. Esto indica que la inclusión de imágenes generadas por GANs ayuda al modelo a aprender de manera más efectiva, reduciendo significativamente los errores tanto de entrenamiento como de validación.

Por último podemos observar que en términos de sobreajuste, en ambos modelos, las curvas de pérdida de entrenamiento y validación están estrechamente alineadas, indicando una buena generalización. Sin embargo, el segundo modelo muestra una alineación ligeramente mejor, sugiriendo que los datos adicionales ayudan a reducir el sobreajuste.

Veamos las gráficas de precisión:

Entrenado solo con datos Reales



Entrenado datos Reales + Generados

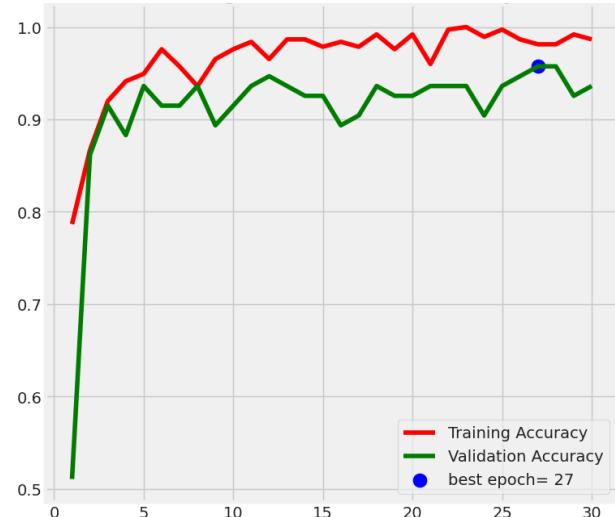


Figura 150: Comparación de gráficas de precisión, datos Reales vs datos R+Generados

Ambos modelos muestran una rápida mejora en la precisión de entrenamiento durante las primeras épocas, lo que indica una buena capacidad de aprendizaje inicial. Sin embargo, el

modelo entrenado solo con datos reales alcanza un pico en la precisión de entrenamiento alrededor de 0.95, mientras que el modelo con datos generados por GANs + originales logra una precisión casi perfecta.

La precisión de validación (**Test**) nos indica la capacidad del modelo para generalizar/clasificar a datos no vistos. El modelo entrenado solo con datos reales alcanza una precisión de validación alrededor de 0.85, pero muestra grandes fluctuaciones, lo que sugiere inestabilidad y posible sobreajuste. En contraste, **el modelo que incluye datos generados por GANs muestra una precisión de validación más alta y estable**, alcanzando alrededor de 0.9. Esto sugiere una mejor capacidad de generalización y una mayor estabilidad en el rendimiento.

### Matriz de confusión

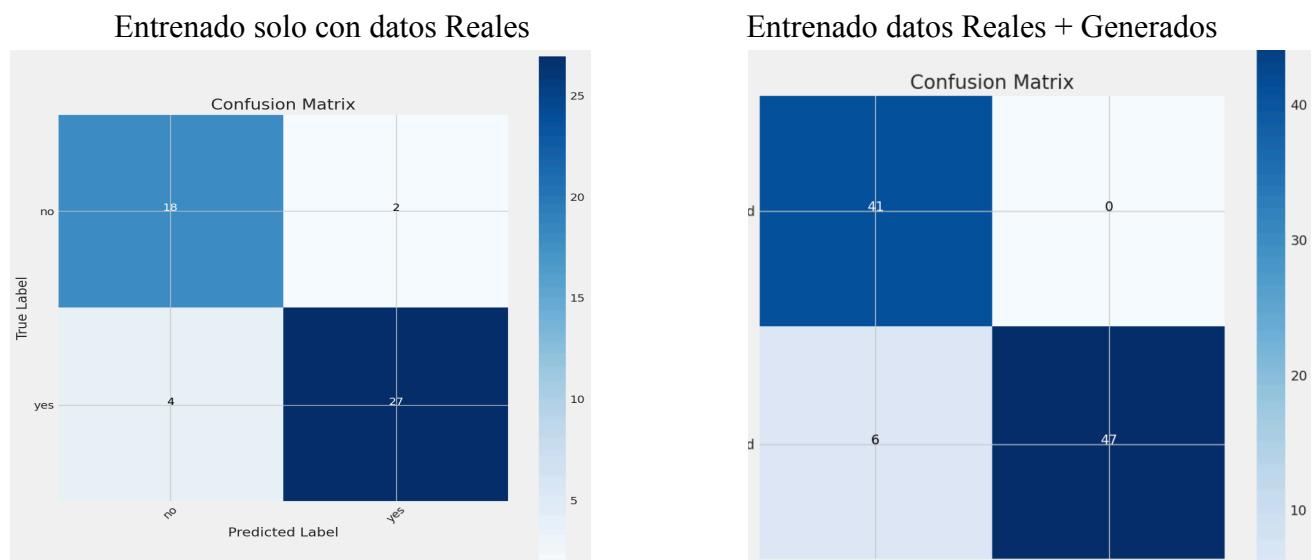


Figura 151: Comparación matrices de confusión

El modelo entrenado solo con datos reales muestra una tasa de error relativamente mayor en comparación con el modelo que incorpora datos generados por GANs. En particular, el primer modelo tiene 6 errores de clasificación en total, mientras que el segundo modelo tiene 6 errores en total pero sobre una base de datos mucho mas grande y ampliada dando por tanto una tasa de error mucho mucho menor que el modelo de datos reales, y más importante, elimina completamente los errores en la clase "no".

El aumento en la precisión y la reducción de los errores en el modelo con datos generados nos sugiere que los cerebros generados proporcionan una mejora significativa en la capacidad del modelo para distinguir entre las clases.

Por ultimo comparemos en total las métricas de entrenamiento de ambos modelos :

Entrenado solo con datos Reales

```
26/26 [=====] - 5s 192ms/step - loss: 2.8753 - accuracy: 1.0000
7/7 [=====] - 1s 177ms/step - loss: 3.1613 - accuracy: 0.8824
Train Loss: 2.8753089904785156
Train Accuracy: 1.0
-----
Test Loss: 3.161306381225586
Test Accuracy: 0.8823529481887817
```

*Figura 152: Métricas finales entrenamiento con datos reales*

Entrenado con datos Reales + Generados

```
47/47 [=====] - 1s 17ms/step - loss: 0.6597 - accuracy: 1.0000
12/12 [=====] - 0s 17ms/step - loss: 0.7807 - accuracy: 0.9362
Train Loss: 0.6596615314483643
Train Accuracy: 1.0
-----
Test Loss: 0.7807276248931885
Test Accuracy: 0.936170220375061
```

*Figura 153: Métricas finales entrenamiento con datos reales + generados*

Repasemos los resultados y hagamos una comparativa:

El modelo entrenado solo con datos reales muestra una pérdida de entrenamiento de 2.8753 y una precisión de entrenamiento perfecta del 100%. Sin embargo, la pérdida de prueba es de 3.1613, con una precisión de prueba del 88.24%. Esto indica que, aunque el modelo aprende bien los datos de entrenamiento, su capacidad para generalizar a nuevos datos no es óptima, lo que se refleja en una mayor pérdida y una menor precisión en el conjunto de prueba.

Por otro lado, el modelo entrenado con datos reales y generados por GANs presenta una pérdida de entrenamiento significativamente menor, de 0.6597, manteniendo también una precisión de entrenamiento del 100%. En el conjunto de prueba, este modelo muestra una pérdida de 0.7807 y una precisión de 93.62%. Estos resultados indican que el modelo no solo aprende eficientemente de los datos de entrenamiento, sino que también generaliza mucho

mejor a datos no vistos, reduciendo la pérdida y aumentando la precisión en el conjunto de prueba.

Comparando ambos modelos, el que fue entrenado solo con datos reales tiene una pérdida de entrenamiento mayor y una precisión de prueba menor en comparación con el modelo que incluye datos generados por GANs. La pérdida de prueba es considerablemente más alta en el primer modelo, lo que sugiere que tiene dificultades para generalizar bien, posiblemente debido a una cantidad insuficiente de datos de entrenamiento diversos.

En contraste, el modelo que incorpora datos generados por GANs no solo tiene una menor pérdida de entrenamiento, sino que también muestra una reducción significativa en la pérdida de prueba y un aumento en la precisión de prueba. Esto indica que los datos adicionales proporcionan ejemplos más diversos y ricos que mejoran la capacidad del modelo para aprender patrones útiles y generalizar mejor.

## 11. CONCLUSIONES Y TRABAJO FUTURO

---

Finalmente podemos afirmar que hemos logrado el objetivo general del proyecto así como los objetivos específicos de este. He explicado y analizado matemáticamente como funcionan las distintas arquitecturas de las redes generativas, hemos implementado y programado distintas redes generativas, hemos mejorado modelos existentes de inteligencia artificial mediante el aumento de datos utilizando datos sintéticos generados por las redes generativas programadas anteriormente.

Tras analizar detenidamente este proyecto sacó dos importantes conclusiones, efectivamente se mejoran los modelos de IA añadiendo datos sintéticos generados por gans, ademas las GANS generan una variedad de datos notable añadiendo variabilidad a los modelos, ya que los datos sintéticos son completamente distintos unos de otros.Pero esta mejora depende de un factor muy importante, **la calidad de los datos generados**, es muy importante que los datos sintéticos se generen de alta calidad es decir que sean indistinguibles a los reales para que estos modelos mejoren. Por tanto es importante entrenar GANS con algoritmos de calidad y confiables que nos aseguran una generación de datos de alta calidad.

Respecto a el trabajo futuro, considero que este proyecto abre un abanico de oportunidades, si un estudiante con presupuesto limitado ha podido generar datos sintéticos y mejorar modelos de inteligencia artificial, con presupuestos y equipos más grandes se pueden mejorar las redes generativas asi como los modelos de IA, esto podría eliminar por completo el problema de la falta de datos en ciertos campos, evitar cuestiones de privacidad de datos y proporcionar ahorros a largo plazo.

Algunas posibles investigaciones e implementaciones que se pueden hacer a partir de este proyecto pueden ser explorar la aplicación de GANs en otros dominios más allá de la generación de imágenes, como la síntesis de texto, audio y video, explorar la integración de GANs con otros tipos de modelos de IA, como modelos de aprendizaje por refuerzo, para mejorar aún más la robustez y precisión de los sistemas de IA. Investigar cómo los datos generados por GANs pueden ser utilizados en diferentes áreas industriales y científicas e Investigar métodos automáticos para ajustar y mejorar continuamente la calidad de los datos sintéticos generados.

## 12. PRINCIPIOS ÉTICOS

---

En el desarrollo de nuestro proyecto de fin de grado de ingeniería, es fundamental considerar los principios éticos que guían nuestra práctica profesional. Estos principios no solo aseguran la integridad y la responsabilidad de nuestro trabajo, sino que también protegen a los individuos y a la sociedad del impacto negativo potencial de la tecnología. En este apartado, discutimos cómo aplicamos los principios éticos en el contexto de nuestro proyecto basado en el uso de Redes Generativas Antagónicas (GANs) para la generación de datos sintéticos y la mejora de modelos de inteligencia artificial (IA).

Uno de los aspectos éticos más relevantes en nuestro proyecto es la privacidad y protección de los datos. La generación de datos sintéticos mediante GANs puede reducir la necesidad de utilizar datos personales sensibles, lo cual es esencial para cumplir con las regulaciones de privacidad como el GDPR en Europa. Sin embargo, debemos asegurar que los datos sintéticos no se utilicen para inferir información sobre individuos reales, lo que podría crear nuevas preocupaciones de privacidad. La autonomía de los individuos debe ser respetada, garantizando que los datos utilizados no comprometan su privacidad sin su consentimiento informado.

La transparencia en la creación y uso de datos generados por GANs es esencial para mantener la confianza de los usuarios y partes interesadas. Es importante comunicar claramente cómo generamos estos datos y los métodos utilizados para garantizar su calidad y seguridad. Esta transparencia ayudará a mitigar la desconfianza y asegurar que los usuarios comprendan y acepten los modelos de IA resultantes, promoviendo así la justicia y equidad en el uso de la tecnología.

El uso de datos sintéticos debe manejarse con cuidado para evitar la introducción de sesgos en los modelos de IA. Es fundamental que los datos generados reflejen de manera precisa la diversidad del mundo real para evitar decisiones discriminatorias o injustas que podrían surgir de modelos entrenados con datos sesgados. La vigilancia continua y la auditoría de los datos y modelos son necesarias para asegurar la equidad y la inclusión. La justicia se ve aquí reflejada

en nuestro compromiso con la igualdad de oportunidades y la no discriminación en las aplicaciones tecnológicas.

Como ingenieros, debemos asumir la responsabilidad y rendición de cuentas por las tecnologías que desarrollamos. Esto incluye asegurarnos de que los modelos de IA sean seguros y confiables para su uso previsto. En el caso de errores o malfuncionamientos, deben existir mecanismos para abordar y rectificar estos problemas de manera oportuna y efectiva. La responsabilidad compartida entre nosotros, los desarrolladores, y los usuarios de la tecnología es esencial para garantizar un impacto positivo y minimizar daños.

El principio de beneficencia exige que nuestras acciones busquen maximizar los beneficios y minimizar los daños. En este contexto, la generación de datos sintéticos puede beneficiar significativamente a sectores con escasez de datos, mejorando la precisión y robustez de los modelos de IA. Sin embargo, debemos tener cuidado de no causar maleficencia, es decir, no introducir daños inadvertidos a través del uso inapropiado o sesgado de la tecnología.

Nuestro proyecto también tiene implicaciones sociales y económicas. Al proporcionar a pequeñas empresas y startups la capacidad de generar datos sintéticos de alta calidad, nivelamos el campo de juego, promoviendo la innovación y permitiendo una mayor competencia en el mercado. Sin embargo, también es necesario considerar los riesgos asociados con la dependencia excesiva de datos sintéticos, como la posible pérdida de la importancia de recolectar datos del mundo real, lo que podría llevar a sesgos y falta de representación.

La integración de principios éticos en el desarrollo y aplicación de tecnologías basadas en GANs es esencial para asegurar que los beneficios de estas innovaciones sean realizados de manera responsable y equitativa. Al abordar cuestiones de privacidad, transparencia, sesgo, responsabilidad, autonomía, beneficencia, justicia e impacto social, nuestro proyecto de fin de grado no solo avanza en la frontera tecnológica, sino que también contribuye a un desarrollo ético y sostenible en la ingeniería. Para más detalles sobre los impactos sociales y éticos de las GANs, consulte el documento original en la sección de impacto social y ético.

## 12. ¿ES LA TECNOLOGIA NEUTRA?

---

Como ingenieros y creadores de tecnología, solemos encontrarnos con el concepto de la neutralidad tecnológica. A menudo se dice que la tecnología es simplemente una herramienta, neutral por naturaleza, y que su impacto depende únicamente del uso que los seres humanos hagan de ella. Sin embargo, al profundizar en nuestro proyecto de fin de grado, centrado en mejorar modelos de inteligencia artificial mediante datos generados por Redes Generativas Antagónicas (GANs), he llegado a cuestionar esta afirmación. Sostengo que la tecnología nunca es verdaderamente neutra y que nuestras creaciones tecnológicas, incluyendo nuestro proyecto, llevan consigo valores y potenciales impactos que debemos considerar con seriedad.

En primer lugar, debemos reconocer que la tecnología no surge en un vacío. Es el producto de decisiones humanas, intenciones y contextos socioeconómicos específicos. En el caso de las GANs, estas han sido diseñadas y desarrolladas con el propósito de generar datos sintéticos que puedan mejorar la robustez y precisión de los modelos de IA. Esta intención, aunque positiva, ya incorpora un valor: la búsqueda de la optimización y la eficiencia en la inteligencia artificial. Este objetivo refleja una visión del progreso tecnológico donde la precisión y la capacidad de los modelos de IA son altamente valoradas. Además, el uso de GANs para generar datos sintéticos plantea implicaciones éticas y sociales que demuestran aún más la no neutralidad de esta tecnología. Al generar datos que pueden ser utilizados en la formación de modelos de IA, las GANs tienen el potencial de influir en la toma de decisiones automatizadas en diversas áreas, desde la medicina hasta la seguridad. Si estos datos sintéticos no son representativos de la diversidad del mundo real, pueden perpetuar o incluso exacerbar sesgos existentes. Así, la tecnología de las GANs no es neutral, sino que tiene el poder de afectar a individuos y comunidades de manera significativa, dependiendo de cómo se diseña y se implemente.

Otro aspecto importante es la transparencia y la confianza. La manera en que comunicamos el funcionamiento y los resultados de los datos generados por GANs influye en la percepción y aceptación de esta tecnología por parte de la sociedad. Si no somos transparentes sobre los límites y las capacidades de estas tecnologías, podríamos generar desconfianza y resistencia. Esto muestra que la tecnología no solo tiene un componente técnico, sino también un componente social y ético que no puede ser ignorado. También debemos considerar la

responsabilidad compartida en el desarrollo y uso de esta tecnología. Como ingenieros, no solo somos responsables de crear tecnologías eficientes y precisas, sino también de anticipar y mitigar posibles impactos negativos. Esto implica un reconocimiento de que nuestras creaciones no son neutrales, sino que pueden tener consecuencias imprevistas que debemos gestionar con cuidado y responsabilidad. La teoría de la no neutralidad tecnológica también se puede ver reflejada en la estructura de poder y control que estas tecnologías pueden reforzar. Las GANs y las tecnologías de IA en general tienen el potencial de centralizar el poder en manos de quienes poseen los recursos y conocimientos para desarrollarlas y utilizarlas. Esto puede aumentar las desigualdades existentes si no se implementan de manera justa y equitativa.

En conclusión, la tecnología que desarrollamos en nuestro proyecto de fin de grado, al igual que cualquier otra, no es neutral. Lleva consigo valores, intenciones y potenciales impactos que debemos considerar con detenimiento. Como ingenieros, es nuestra responsabilidad reconocer y abordar estos aspectos, asegurando que nuestras creaciones contribuyan de manera positiva y equitativa a la sociedad. Al aceptar que la tecnología nunca es neutral, podemos trabajar de manera más consciente y ética, anticipando y mitigando los efectos negativos, y promoviendo un desarrollo tecnológico que beneficie a todos.

## **13. PLAN DE TRABAJO Y PRESUPUESTO**

---

### **13.1 PLAN DE TRABAJO**

Para la realización del proyecto realice el siguiente plan de trabajo progresivamente

#### 1. Fase De Investigacion y Analisis

LLeve a cabo una fase de investigacion exhaustiva y de Analisis, donde investigue distintos papers, videos de youtube, el libro de generative deep learning de David foster etc... (muchos de ellos adjuntos en la bibliografia), gracias a esta investigacion forje mi base de conocimiento teórico y práctico de el deep learning, en específico las redes generativas adversariales. Estudie las distintas arquitecturas de GANS cómo funcionaban profundamente para poderlas comprender prácticamente a su vez.

#### 2. Fase De Diseño y Configuración

En esta fase me dediqué a diseñar y configurar las distintas gans y modelos de IA que iba a implementar en este proyecto, para ello comprobé distintas librerías, gpus, tecnologías en la

nube o dispositivos como el Jetson nano. A partir de estas tecnologías desarrolle y probe código en distintas plataformas, evalúe sus rendimientos y finalmente me decante por una serie de tecnologías para cada implementación de los distintos modelos de este proyecto. Una parte importante de este proyecto también fue la estimación y diseño del uso de memoria y de recursos que me iba a costar cada implementación, así que las dependencias de las librerías fuesen todas las correctas.

### 3.Fase de Creacion e Implementación

Una vez diseñado y configurados los entornos así como el plan del proyecto,llegó la hora de desarrollar el código y por ende crear y entrenar las siguientes arquitecturas GANS:DCGAN,PROGAN,WGAN-GP con los datasets que seleccione. Tras ver los diferentes resultados experimente con el ajuste de los hiper parámetros y cambiando los entrenamientos de los modelos discriminador / generador para generar las imágenes de mayor calidad.Una vez ya optimizados y creados los modelos de manera eficiente, descargo las imágenes generadas por las redes adversariales y lleve a cabo la siguiente fase.

### 4.Fase de Integración Evaluación y Mejora de Modelos

Una vez generadas las imágenes, busqué distintos modelos CNN que podía mejorar mediante la integración de las imágenes en los datos de entrenamiento/test. Probe los distintos modelos CNN con datos reales primero evalúe su rendimiento mediante las métricas de entrenamiento y plts. Una vez hecho mezcló los ejemplos generados (imágenes sintéticas ) mediante GANS. Entreno el modelo, y veo cómo se comporta, dados ambos resultados comparó el entrenamiento con el conjunto de datos original vs el mejorado con imágenes generadas a partir del vector ruido.

Durante esta fase he probado diversos modelos y diversos ejemplos generados llegando a la principal conclusión, que los modelos mejoran siempre que las imágenes generadas por las GANS sean de buena calidad.

### 5.Fase de Documentación y Divulgación

Aunque esta fase la lleve a cabo progresivamente con el desarrollo de las otras fases del proyecto, en el último tramo me asegure de elaborar la memoria de acorde a sus normas, incluyendo la metodología, resultados y conclusiones. En esta fase también prepare mi defensa del proyecto.

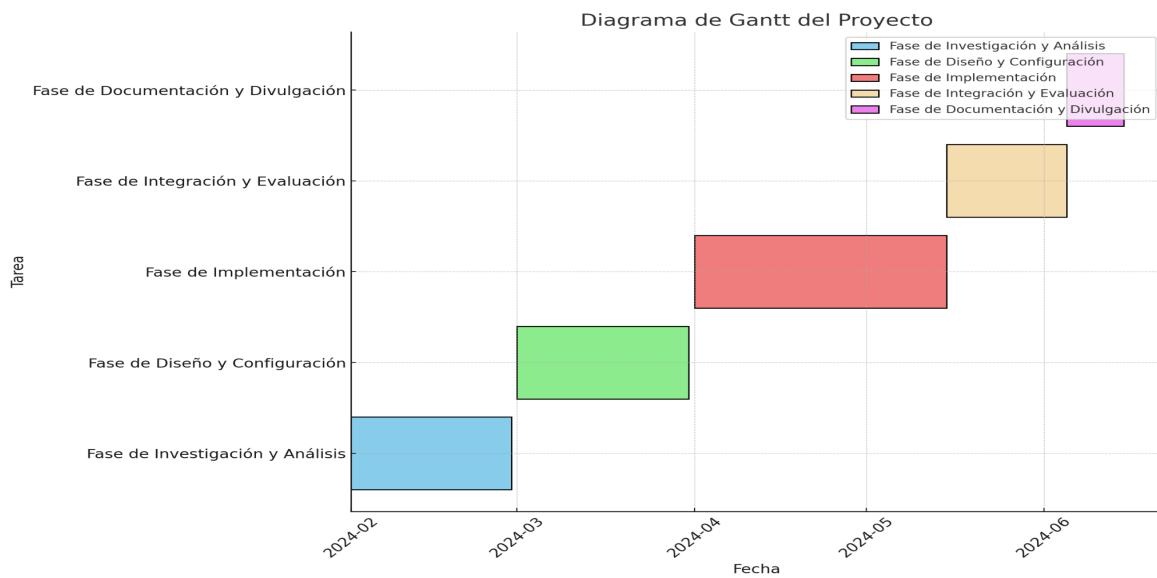


Figura 154: Diagrama de Gantt

## 13.2 PRESUPUESTO

### 13.2.1 Presupuesto Personal

El presupuesto real y personal que he invertido en este proyecto de fin de grado es el siguiente:

La suscripción a Google Colab Pro para aprovechar la GPU A100 ha tenido un coste de €90. Además, he utilizado mi laptop personal, un Asus FX505DV, que cuenta con una GPU NVIDIA GeForce RTX 2060, como componente de hardware para el entrenamiento local como en la progar y la degan del proyecto. También intenté utilizar una Jetson Nano, aunque sin éxito en el proyecto.

### 13.2.2 Presupuesto Caso de Uso Real

A continuación, se presenta el presupuesto estimado para una empresa que desee llevar a cabo el objetivo de este proyecto de fin de grado, considerando los costos asociados a la investigación, desarrollo, implementación y divulgación de los resultados.

#### Investigación y Análisis

Para la fase de investigación y análisis, estimamos un coste total de €15.000. Este importe incluye la adquisición de libros y materiales de referencia, el acceso a publicaciones académicas y la capacitación del personal en técnicas avanzadas de Inteligencia Artificial y Redes Generativas Antagónicas (GANs).

## Diseño y Configuración

En la fase de diseño y configuración, hemos considerado los siguientes costes:

En hardware, se estima un coste de €25.000 para servidores de alto rendimiento necesarios para el entrenamiento de modelos y €30.000 para GPUs de última generación, como la NVIDIA A100, lo que suma un total de €55.000 ([Supermicro](#)).

En software, se ha presupuestado €3.000 para licencias de software de desarrollo y herramientas especializadas, como PyCharm Professional y MATLAB, y €10.000 para servicios en la nube, incluyendo almacenamiento y procesamiento de datos en plataformas como AWS y Google Cloud, lo que da un total de €13.000 ([Akkio](#)).

## Creación e Implementación

Los costes de computación y desarrollo se estiman en €20.000. Este importe incluye el alquiler de servidores en la nube para el entrenamiento de modelos, los costes de electricidad y el mantenimiento de hardware. También se consideran los gastos asociados a la adquisición y preparación de datasets específicos para el entrenamiento y prueba de los modelos.

### Integración, Evaluación y Mejora de Modelos

Para la integración, evaluación y mejora de los modelos, hemos estimado un coste de €30.000. Este presupuesto cubre los honorarios de consultores expertos en IA, ingenieros de datos y científicos de datos necesarios para asegurar la calidad y efectividad de los modelos desarrollados.

## Documentación y Divulgación

La documentación y divulgación del proyecto tienen un coste estimado de €5.000. Este importe incluye la preparación de informes detallados, manuales de usuario, materiales de presentación, y la realización de seminarios o workshops para la divulgación de los resultados.

### Costes Adicionales y Contingencias

Se ha asignado un 15% del total del presupuesto para contingencias, representando un importe de €19.800, para cubrir cualquier imprevisto que pudiera surgir durante el desarrollo del proyecto.

### Resumen del Presupuesto

El presupuesto para la investigación y análisis es de €15.000. Para el diseño y configuración, se estiman €68.000. La creación e implementación tiene un coste de €20.000. La integración, evaluación y mejora de modelos se presupuestan en €30.000. La documentación y divulgación ascienden a €5.000. Finalmente, se asignan €19.800 para contingencias.

Total General del Proyecto: €157.800

Este presupuesto refleja una estimación realista y profesional de los costes necesarios para que una empresa pueda aplicar los objetivos de mejora de modelos utilizando ejemplos generados por GANS, asegurando que todas las fases del desarrollo estén debidamente financiadas para alcanzar los objetivos propuestos.

## REPOSITORIOS DEL CÓDIGO

---

En esta sección comparto el código que he desarrollado durante este proyecto.

### **Implementación de DCGANMNIST**

<https://github.com/ibarrita17/PFGJAVIERIBARRA>

### **Implementación de WGAN-GP**

[https://colab.research.google.com/drive/1Gq1VgCZIKgC4y9pV\\_miZycVDh0wJWtIB?usp=sharing](https://colab.research.google.com/drive/1Gq1VgCZIKgC4y9pV_miZycVDh0wJWtIB?usp=sharing)

### **Implementación Pro-GAN**

<https://github.com/ibarrita17/PFGJAVIERIBARRA>

### **Mejora del Modelo CNN**

[https://colab.research.google.com/drive/1gbV0jfZjRoA3kQ1FqIEk\\_8r4VHgwhavw?usp=sharing](https://colab.research.google.com/drive/1gbV0jfZjRoA3kQ1FqIEk_8r4VHgwhavw?usp=sharing)

## BIBLIOGRAFÍA

---

- [1] David Foster, Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play, O'Reilly Media, 2019.
- [2] LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* 521, 436–444 (2015). <https://doi.org/10.1038/nature14539>
- [3] Yilmaz, B., & Korn, R. (2023). Understanding the mathematical background of Generative Adversarial Networks (GANs). Mathematical Modelling and Numerical Simulation With Applications, 3(3), 234-255. <https://doi.org/10.53391/mmnsa.1327485>
- [4] Zhengwei WGANG, Qi She, and Tomás E. Ward. 2021. Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy. ACM Comput. Surv. 54, 2, Article 37 (March 2022), 38 pages. <https://doi.org/10.1145/3439723>
- [5] Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv preprint arXiv:1511.06434.
- [6] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Nets. Advances in Neural Information Processing Systems, 27.
- [7] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved Techniques for Training GANs. Advances in Neural Information Processing Systems, 29.
- [8] Xu, B., WGANG, N., Chen, T., & Li, M. (2015). Empirical Evaluation of Rectified Activations in Convolutional Network. arXiv preprint arXiv:1505.00853.
- [9] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.
- [10] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. (2017). Improved Training of Wasserstein GANs. arXiv preprint arXiv:1704.00028. <https://doi.org/10.48550/arXiv.1704.00028>
- [11] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation. International Conference on Learning Representations(ICLR): [https://research.nvidia.com/sites/default/files/pubs/2017-10\\_Progressive-Growing-of/karras2018iclr-paper.pdf](https://research.nvidia.com/sites/default/files/pubs/2017-10_Progressive-Growing-of/karras2018iclr-paper.pdf)
- [12] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation. International Conference on Learning

Representations(ICLR)GitHubRepository:  
[https://github.com/tkarras/progressive\\_growing\\_of\\_gans](https://github.com/tkarras/progressive_growing_of_gans)

- [13] Expansion of the CT-scans image set based on the pretrained DCGAN for improving the performance of the CNN, Ruoxi Cheng 2023 J. Phys.: Conf. Ser. 2646 012015: <https://iopscience.iop.org/article/10.1088/1742-6596/2646/1/012015>
- [14] Kocagil, C. (2023). DCGAN. GitHub repository. <https://github.com/cankocagil/DCGAN>
- [15] Dupont,E.(2023).wgan-gp. GitHub repository.:<https://github.com/EmilienDupont/wgan-gp>
- [16] [https://www.researchgate.net/figure/A-simple-illustration-of-how-one-can-use-discriminative-vs-generative-models-The-former\\_fig1\\_341478640](https://www.researchgate.net/figure/A-simple-illustration-of-how-one-can-use-discriminative-vs-generative-models-The-former_fig1_341478640)
- [17] [https://www.researchgate.net/figure/Generative-Adversarial-Networks-Architecture-AltexSoft-2022\\_fig4\\_370761753](https://www.researchgate.net/figure/Generative-Adversarial-Networks-Architecture-AltexSoft-2022_fig4_370761753)
- [18] [https://www.researchgate.net/figure/Figura-5-Exemplo-de-filtro-de-convolucao\\_fig3\\_365985201](https://www.researchgate.net/figure/Figura-5-Exemplo-de-filtro-de-convolucao_fig3_365985201)
- [19] [https://www.researchgate.net/publication/349244221\\_The\\_Application\\_of\\_a\\_Deep\\_Convolutional\\_Generative\\_Adversarial\\_Network\\_on\\_Completing\\_Global\\_TEC\\_Maps](https://www.researchgate.net/publication/349244221_The_Application_of_a_Deep_Convolutional_Generative_Adversarial_Network_on_Completing_Global_TEC_Maps)
- [20] Doshi, K. (2021, May 18). Batch Norm Explained Visually — How it works, and why neural networks need it: A Gentle Guide to an all-important Deep Learning layer, in Plain English. TowardsDataScience.Retrieved from<https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739>
- [21] User241481. (2020, September 2). How to implement a neural network with a not fully connected layer as the final layer? Stack Overflow. <https://stackoverflow.com/questions/63675602/how-to-implement-a-neural-network-with-a-not-fully-connected-layer-as-the-final>
- [22] Gautm,S.(n.d.).HowProGANworks:<https://sushantgautm.medium.com/how-progan-woks-5e14d2baf29c>
- [23] Brownlee, J. (2019, July 10). Introduction to progressive growing generative adversarial networks.MachineLearningMastery.<https://machinelearningmastery.com/introduction-to-progressive-growing-generative-adversarial-networks/>
- [24] HojjatK.(n.d.).MNISTdataset.Kaggle:<https://www.kaggle.com/datasets/hojjatk/mnist-dataset>
- [25] Navoneel. (n.d.). Brain MRI images for brain tumor detection. Kaggle. Retrieved from <https://www.kaggle.com/datasets/navoneel/brain-mri-images-for-brain-tumor-detection>
- [26] CelebA-HQ. (n.d.). Papers with Code.<https://paperswithcode.com/dataset/celeba-hq>

- [27] Rosebrock, A. (2021, May 14). *Convolutional Neural Networks (CNNs) and Layer Types*. PyImageSearch.  
<https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/>
- [28] Qin, Zhaoxiang & Shan, Yuntao. (2021). Generation of Handwritten Numbers Using Generative Adversarial Networks. Journal of Physics: Conference Series. 1827. 012070. 10.1088/1742-6596/1827/1/012070.
- [29] Jamhuri, S. (2020, September 17). *Understanding the Adam optimization algorithm: A deep dive into the formulas*. Medium.  
<https://jamhuri.medium.com/understanding-the-adam-optimization-algorithm-a-deep-dive-into-the-formulas-3ac5fc5b7cd3>
- [30] GeeksforGeeks. (n.d.). *Adam Optimizer*. GeeksforGeeks. Recuperado de  
<https://www.geeksforgeeks.org/adam-optimizer/>
- [31] Maniyar, Huzaifa & Budihal, Suneeta & Siddamal, Saroja. (2022). Persons facial image synthesis from audio with Generative Adversarial Networks. ECTI Transactions on Computer and Information Technology (ECTI-CIT). 16. 135-141. 10.37936/ecticit.2022162.246995.
- [32] Guardia, O. (2023). *Master Thesis*. Universitat Politècnica de Catalunya. Recuperado de  
[https://upcommons.upc.edu/bitstream/handle/2117/402137/Master\\_Thesis\\_Oriol\\_Guardia.pdf?sequence=3](https://upcommons.upc.edu/bitstream/handle/2117/402137/Master_Thesis_Oriol_Guardia.pdf?sequence=3)
- [33] Nayak, S. (2019, June 19). *Demystified: Wasserstein GAN with Gradient Penalty*. Towards Data Science. Recuperado de  
<https://towardsdatascience.com/demystified-wasserstein-gan-with-gradient-penalty-ba5e9b905ead>