

# Estudio de la mejora del aprendizaje de Redes Neuronales mediante el uso de Redes Generativas Antagónicas

Autor

**ÁLVARO SERNA PRIETO**  
([alserpr1@posgrado.upv.es](mailto:alserpr1@posgrado.upv.es))

Director: JOSÉ ENRIQUE SIMO TEN, JOSÉ LUIS NAVARRO HERRERO  
([jsimo@disca.upv.es](mailto:jsimo@disca.upv.es), [joseluis@isa.upv.es](mailto:joseluis@isa.upv.es))

**TRABAJO FIN DE MASTER**  
**MASTER DE AUTOMÁTICA E INFORMÁTICA INDUSTRIAL**  
**UNIVERSITAT POLITÈCNICA DE VALÈNCIA**



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

**UNIVERSITAT POLITÈCNICA DE VALÈNCIA**  
Valencia, 5 de Noviembre de 2020

A mi familia, por velar tanto por mi educación, mis amigos por apoyarme en cualquier decisión y a Lidia, por los buenos momentos compartidos.

## **AGRADECIMIENTOS**

A José Enrique Simo Ten y a José Luis Navarro Herrero, por el trabajo de guiarme y aconsejarme en este proyecto, además de darme a conocer un mundo tan interesante y útil como son las redes neuronales y su aplicación a la industria.

## TABLA DE CONTENIDO

1. <b>INTRODUCCION</b> .....	XI
2. <b>JUSTIFICACION</b> .....	XII
3. <b>OBJETIVOS</b> .....	XIII
3.1 Objetivo general .....	XIII
3.2 Objetivos específicos .....	XIII
4. <b>ESTADO ACTUAL</b> .....	1
4.1 Inteligencia Artificial.....	1
4.1.1 Machine learning. ....	1
4.1.1.1 Deep learning.....	2
4.2 Redes neuronales .....	2
4.2.1 Neuronas artificiales, la unidad básica. ....	3
4.2.1 Arquitectura de redes neuronales.....	5
4.2.1.1 Perceptrón (Rosenblatt, 1959). ....	6
4.2.1.2 Redes neuronales monocapa. ....	6
4.2.1.3 Redes neuronales multicapa.....	7
4.2.1.4 Redes neuronales recurrentes.....	7
4.2.1.5 Redes neuronales competitivas.....	8
4.2.1 Aprendizaje de las redes neuronales.....	8
4.2.1.1 Algoritmo “Backpropagation”.....	10
4.3 Redes Neuronales Convolucionales .....	10
4.4 Redes Generativas Antagónicas (GAN).....	13
4.5 Herramientas utilizadas para el desarrollo del trabajo.....	16
4.5.1 Python. ....	16
4.5.1.1 Scikit.....	16
4.5.1.2 OpenCV. ....	17
4.5.1.3 Tensorflow.....	17
4.5.1.4 Keras.....	17
5. <b>DESARROLLO</b> .....	18
5.1 Ejemplos de redes gan desarrolladas en internet .....	18
5.1.1 Red GAN generadora de números escritos a mano MNIST. ....	18
5.1.1.1 Dataset MNIST.....	18
5.1.1.2 Discriminador del modelo.....	19

5.1.1.3	Generador del modelo.....	21
5.1.1.4	Modelo GAN.....	24
5.1.2	Red GAN CIFAR-10 generadora de pequeñas fotografías.....	26
5.1.2.1	Dataset CIFAR-10.....	26
5.1.2.2	Discriminador del modelo.....	27
5.1.2.3	Generador del modelo.....	31
5.1.2.4	Modelo GAN.....	35
5.2	Adaptación de las redes GAN de ejemplo al caso de estudio.....	38
5.2.1	Red GAN adaptada generadora de números escritos a mano MNIST. 38	
5.2.1.1	Preparación del dataset. ....	38
5.2.1.2	Discriminador adaptado del modelo.....	39
5.2.1.3	Generador adaptado del modelo.....	41
5.2.2	Red GAN adaptada CIFAR-10 generadora de pequeñas fotografías.42	
5.2.2.1	Preparación del dataset. ....	42
5.2.2.2	Discriminador adaptado del modelo.....	44
5.2.2.3	Generador adaptado del modelo.....	44
5.2.3	Generador de imágenes para entrenamiento de la red clasificadora basado en la red GAN CIFAR-10. ....	46
6.	<b>RESULTADOS</b> .....	48
6.1	Redes GAN de ejemplo desarrolladas en Internet .....	48
6.1.1	Red GAN generadora de números escritos a mano MNIST. ....	48
6.1.2	Red GAN CIFAR-10 generadora de pequeñas fotografías.....	50
6.2	Adaptación de las redes GAN de ejemplo al caso de estudio para la generación de imágenes.....	52
6.2.1	Red GAN adaptada generadora de números escritos a mano MNIST. 52	
6.2.2	Red GAN adaptada CIFAR-10 generadora de pequeñas fotografías.55	
6.2.2.1	Entrenamiento con todas las clases.....	55
6.2.2.2	Entrenamiento por clases.....	58
6.3	Entrenamiento de la red clasificadora con imágenes generadas .....	63
6.3.1	Entrenamiento con 2 clases.....	63
6.3.2	Entrenamiento con 4 clases.....	64
6.3.3	Entrenamiento con 8 clases.....	65
6.3.4	Entrenamiento con 10 clases.....	65

6.3.5	Entrenamiento con 12 clases.....	66
6.3.6	Entrenamiento con 15 clases.....	67
<b>7.</b>	<b>ANALISIS Y DISCUSION DE RESULTADOS</b> .....	<b>69</b>
7.1	Redes gan de ejemplo desarrolladas en internet .....	69
7.1.1	Red GAN generadora de números escritos a mano MNIST. ....	69
7.1.1	Red GAN CIFAR-10 generadora de pequeñas fotografías.....	70
7.2	Adaptación de las redes GAN de ejemplo al caso de estudio para la generación de imágenes.....	70
7.2.1	Red GAN adaptada generadora de números escritos a mano MNIST. 70	
7.2.2	Red GAN adaptada CIFAR-10 generadora de pequeñas fotografías.71	
7.2.2.1	Entrenamiento con todas las clases.....	71
7.2.2.2	Entrenamiento por clases.....	72
7.3	Entrenamiento de la red clasificadora con imágenes generadas .....	72
7.3.1	Entrenamiento con diferente número de clases.....	72
7.3.2	Comparativa de resultados. ....	73
<b>8.</b>	<b>CONCLUSIONES</b> .....	<b>76</b>
<b>9.</b>	<b>RECOMENDACIONES Y TRABAJOS FUTUROS</b> .....	<b>78</b>
<b>10.</b>	<b>REFERENCIAS BIBLIOGRÁFICAS</b> .....	<b>79</b>
<b>11.</b>	<b>ANEXOS</b> .....	<b>81</b>
11.1	Anexo 1. Código relativo a la RED GAN MNIST completa (GAN_MNIST.py).....	81
11.2	Anexo 2. código relativo al discriminador de la red Cifar-10 (CIFAR_10_DISCRIMINATOR.PY).....	84
11.3	Anexo 3. código relativo al generador de la red cifar-10 (CIFAR_10_GENERATOR.PY).....	86
11.4	Anexo 4. código relativo a la red gan cifar-10 completa (CIFAR_10_GAN.PY) .....	87
11.5	Anexo 5. código relativo a la red GAN MNIST adaptada para el caso de estudio de imágenes de (GAN_MNIST_CHUCHES.PY) .....	91
11.6	Anexo 6. Código relativo a la red GAN CIFAR-10 adaptada para el caso de imágenes de (GAN_CIFAR_10_CHUCHES.PY) .....	95
11.7	Anexo 7. Código relativo a la red clasificadora de imágenes de (CLASIFICADOR_CON_GAN.PY).....	100

## LISTA DE TABLAS

Tabla 1.- Funciones de activación mas utilizadas en los modelos de neuronas artificiales dentro de las redes neuronales artificiales (Fuente: [15]).	4
Tabla 2.- Evolución de las imágenes generadas del dataset MNIST con respecto a la iteración del entrenamiento con la red GAN MNIST.	50
Tabla 3.- Evolución de las imágenes generadas del dataset CIFAR-10 con respecto a la iteración del entrenamiento con la red GAN CIFAR-10.	52
Tabla 4.- Evolución de las imágenes generadas del dataset golosinas de clase “Beso” con respecto a la iteración del entrenamiento con la red GAN MNIST adaptada.	54
Tabla 5.- Evolución de las imágenes generadas del dataset de golosinas con respecto a la iteración del entrenamiento con la red CIFAR-10 adaptada.	57
Tabla 6.- Imágenes generadas obtenidas entrenando la red GAN CIFAR-10 adaptada para cada clase después de 300 iteraciones.	63
Tabla 7.- Ficha del experimento para 2 clases.	63
Tabla 8.- Resultados del entrenamiento con 2 clases.	64
Tabla 9.- Ficha del experimento para 4 clases.	64
Tabla 10.- Resultados del entrenamiento con 4 clases.	65
Tabla 11.- Ficha del experimento para 8 clases.	65
Tabla 12.- Resultados del entrenamiento con 8 clases.	65
Tabla 13.- Ficha del experimento para 10 clases.	66
Tabla 14.- Resultados del entrenamiento con 10 clases.	66
Tabla 15.- Ficha del experimento para 12 clases.	66
Tabla 16.- Resultados del entrenamiento con 12 clases.	67
Tabla 17.- Ficha del experimento para 15 clases.	67
Tabla 18.- Resultados del entrenamiento con 15 clases.	68
Tabla 19.- Tabla comparativa de resultados de la red clasificadora con y sin imágenes generadas.	74

## LISTA DE FIGURAS

Figura 1.- Diferencia entre machine learning y Deep learning (Fuente: [16]).	2
Figura 2.- Modelo genérico de neurona artificial (Fuente: [15]).	3
Figura 3.- Conexión entre dos neuronas artificiales en una red neuronal (Fuente: [15]).	5
Figura 4.- Perceptrón simple (Fuente: [15]).	6
Figura 5.- Red neuronal Monocapa (Fuente: [15]).	7
Figura 6.- Red neuronal multicapa (Fuente: [15]).	7
Figura 7.- Red neuronal recurrente (Fuente: [15]).	8
Figura 8.- Red neuronal competitiva (Fuente: <a href="https://es.slideshare.net/mentelibre/redes-neuronales-aprendizaje-competitivo-cooperativo">https://es.slideshare.net/mentelibre/redes-neuronales-aprendizaje-competitivo-cooperativo</a> ).	8
Figura 9.- Imagen y su descomposición en píxeles (Fuente: [3]).	11
Figura 10.- Proceso de convolución (Fuente: <a href="https://es.quora.com/C%C3%B3mo-funcionan-las-redes-neuronales-convolucionales">https://es.quora.com/C%C3%B3mo-funcionan-las-redes-neuronales-convolucionales</a> ).	11
Figura 11.- Función de activación Softmax (Fuente: <a href="https://themaverickmeerkat.com/2019-10-23-Softmax/">https://themaverickmeerkat.com/2019-10-23-Softmax/</a> ).	13
Figura 12.- Esquema de arquitectura de una red neuronal convolucional para la clasificación de imágenes (Fuente: [3]).	13
Figura 13.- Esquema de funcionamiento de una red GAN (Fuente: <a href="https://proyectoidis.org/red-generativa-antagonica-gan/">https://proyectoidis.org/red-generativa-antagonica-gan/</a> ).	14
Figura 14.- Modelo desconvolucional de un generador en una red GAN (Fuente: [9]).	15
Figura 15.- Ejemplo de imágenes del dataset MNIST (Fuente: <a href="https://deepai.org/dataset/mnist">https://deepai.org/dataset/mnist</a> ).	19
Figura 16.- Resumen del modelo discriminador de la red GAN MNIST.	21
Figura 17.- Resumen del modelo generador de la red GAN MNIST.	23
Figura 18.- Ejemplo de uso del modelo generador sin entrenar en el caso MNIST.	24
Figura 19.- Ejemplo de imágenes del dataset CIFAR-10 (Fuente: [10]).	27
Figura 20.- Resumen del modelo discriminador de la red GAN CIFAR-10 (Fuente: [10]).	29
Figura 21.- Resultado de precisión obtenida al entrenar la red discriminadora.	31
Figura 22.- Resumen del modelo generador de la red GAN CIFAR-10 (Fuente: [10]).	33
Figura 23.- Resultado obtenido de la ejecución del código del modelo del generador para la red GAN CIFAR-10 (Fuente: [10]).	34
Figura 24.- Resumen del modelo discriminador adaptado de la red GAN MNIST.	40
Figura 25.- Resumen del modelo generador adaptado de la red GAN MNIST.	42
Figura 26.- Entrenamiento con 2 clases.	64
Figura 27.- Entrenamiento con 4 clases.	64
Figura 28.- Entrenamiento con 8 clases.	65
Figura 29.- Entrenamiento con 10 clases.	66
Figura 30.- Entrenamiento con 12 clases.	67
Figura 31.- Entrenamiento con 15 clases.	67
Figura 32.- Imágenes generadas en la iteración 47 del entrenamiento de la red GAN con dataset MNIST.	69
Figura 33.- Comparación entre imagen real y generadas por la red GAN MNIST adaptada de la clase "Beso" del dataset golosinas.	71
Figura 34.- Gráfica de tendencia del tiempo de entrenamiento conforme a las clases tenidas en cuenta.	73

Figura 35.- Comparativa gráfica de datos de validación de la red entrenada con y sin imágenes generadas. .... 75

## RESUMEN EXTENDIDO

La Inteligencia Artificial y el Machine Learning (aprendizaje automático) han sido unas de las tecnologías recientes que más se han desarrollado en los últimos años, hasta llegar al punto, de ser muchas veces incorporadas en los nuevos entornos industriales, proporcionando gran variedad de aplicaciones y herramientas para mejorar la productividad y eficiencia de las industrias.

Este Trabajo Fin de Máster, se basa en el estudio y desarrollo de un tipo de redes neuronales, las redes generativas antagónicas (o redes GAN), para aplicarlas en el entrenamiento de otra red neuronal ya desarrollada en un Trabajo Fin de Grado por su autor, Andreu Crespo Barberá. Estas redes son capaces de generar imágenes falsas pero que son lo suficientemente plausibles como para engañar incluso al ojo humano. Con ello lo que se pretende es generar un banco de imágenes lo suficientemente grande como para, al entrenar la red ya desarrollada, se mejoren los resultados de precisión de la misma para clasificar imágenes (en este caso de productos alimentarios como son las golosinas).

Para ello, se realizará un recorrido sobre el estado actual de la Inteligencia Artificial, diferenciando conceptos como Machine Learning o Deep Learning, viendo que son las redes neuronales y cómo se construyen, analizando lo que son las redes convolucionales utilizadas en este trabajo y en el trabajo en el que se basa este estudio y por último viendo en detalle en qué se basa una red GAN y cuáles son las herramientas para desarrollar todas estas redes.

Con esos conocimientos se pasa a obtener ejemplos de redes GAN de internet con pruebas de éxito en sus respectivos campos, una con imágenes de dígitos manuscritos y otra con pequeñas imágenes a color. Estas redes se analizarán y se podrán comprobar sus resultados después de entrenarlas, para después modificarlas al caso de estudio de este trabajo, las imágenes de golosinas. Con estas adaptaciones se podrán generar imágenes falsas con cada red y ver que red es capaz de generar las imágenes más plausibles, que serán utilizadas para el entrenamiento de la red ya desarrollada en el Trabajo Fin de Grado en el que se basa este estudio.

Se realizará un estudio sobre el entrenamiento de la red con el banco de imágenes generadas y una comparativa de resultados con los obtenidos en el Trabajo Fin de Grado de Andreu Crespo Barberá, aportando las conclusiones a las que se llega con el desarrollo de este trabajo, sus limitaciones y los posibles trabajos futuros que se puedan realizar sobre él.

## 1. INTRODUCCION

Hoy en día todas las partes de la industria ya trabajan de manera informatizada y poco a poco, automatizada. Esto hace que la mejora de estas partes mejoren en si la producción y la organización de la industria, ya sea con ahorro energético, consiguiendo una manera más rápida y con menos recursos de realizar ciertas tareas, costes de materia prima, consiguiendo un mejor provecho de la materia prima o consiguiendo una producción más rápida gracias a la automatización.

En este contexto se presenta un campo interesante a la par que útil para la industria relacionado con la informática industrial, las redes neuronales. Estas redes básicamente son programas que se dedican a optimizar un índice de coste elegido por el programador, y esto hace que podamos usarlas en muchos ámbitos, desde la visión por computador en la industria, mediante el reconocimiento de imágenes o patrones en video, hasta conseguir un control inteligente para regular cualquier proceso de manera informatizada. Éstas también nos dan una herramienta muy útil en el campo de la automatización de procesos, ya sea para clasificación de productos, robótica o control de procesos.

Con este pretexto, se abrió un campo interesante en el que investigar y trabajar, al ver el uso de estas redes en el control de procesos. Se presenta la oportunidad de poder sumergirse en este mundo de la inteligencia artificial debido a un trabajo anterior realizado por un estudiante del grado de Ingeniería Informática, Andreu Crespo Barberá, el cual desarrolla un sistema empotrado de visión por computador que utiliza una red neuronal de clasificación de imágenes para la clasificación de golosinas en la producción de éstas.

El trabajo consiste en estudiar la posibilidad de introducir una red neuronal GAN (Generative Adversarial Network) para comprobar si hay una mejora o no en el aprendizaje y por lo tanto en el rendimiento de la red neuronal presente en el sistema empotrado. Estas redes GAN básicamente lo que buscan es crear imágenes falsas a partir de imágenes reales, consiguiendo la cantidad que se quieran de imágenes para entrenar mejor la red de clasificación y ver si así se mejora el rendimiento de la red.

El trabajo se estructura dando en un principio la justificación y objetivos que se pretenden alcanzar en este trabajo, pasando a la exposición del estado del arte de la inteligencia artificial y redes neuronales, donde se explicará con profundidad los conceptos base del trabajo, seguido del desarrollo de éste, explicando y mostrando los diferentes programas creados. Después se muestran los resultados obtenidos y el análisis de los mismos para terminar con un apartado de conclusiones, trabajos futuros y la bibliografía utilizada. También se incluye un apartado de anexos donde se pueden consultar todos los códigos utilizados al completo.

## 2. JUSTIFICACION

Este trabajo trata de un campo interesante y útil como son las redes neuronales y su aplicación en la industria. Con afán de indagar más sobre el tema, se presenta la posibilidad de realizar este trabajo y poder así ver más aplicaciones de las redes neuronales en este ámbito, en este caso para mejorar otra aplicación ya desarrollada en el ámbito de la visión por computador y la industria. Por ello se verá en este trabajo el desarrollo de la red GAN para la mejora de la red anteriormente propuesta, la implementación y el estudio comparativo de resultados.

Con este trabajo se busca estudiar la mejora de las redes neuronales de clasificación, más concretamente la clasificación de imágenes, mediante la generación de imágenes falsas obtenidas con una red GAN. Esto resulta de gran utilidad debido a la menor necesidad de obtener imágenes reales de la materia a clasificar, reduciendo en una enorme medida el tiempo y medios materiales de trabajo en cualquier proceso de obtención de imágenes para la industria.

La metodología de este trabajo consiste en primeramente probar redes GAN más sencillas previamente desarrolladas y adaptarlas al problema concreto, modificando la estructura o ciertas funciones. Se escoge esta metodología ya que existe una gran cantidad de desarrollos de dichas redes ya contrastados y con ellos se puede trabajar de manera cómoda y rápida obteniendo muy buenos resultados.

Se escogen las redes GAN para la mejora de la red anterior debido a que se requerirían muchas horas y medios para obtener miles y miles de imágenes para entrenar la red de clasificación de imágenes, por lo que esta red GAN, que generará imágenes muy parecidas a las previamente obtenidas en un tiempo mucho menor, cambiando las orientaciones de los objetos o iluminación, se presenta como la mejor opción de mejora de la red de clasificación anteriormente desarrollada.

### **3. OBJETIVOS**

Atendiendo a las motivaciones y razones argumentadas anteriormente, en este trabajo se buscan principalmente los siguientes objetivos.

#### **3.1 OBJETIVO GENERAL**

El objetivo en general de este trabajo es el desarrollo de redes neuronales generadoras-antagónicas para un problema concreto y estudiar si las imágenes generadas por éstas son plausibles y si su uso es útil para la mejora de entrenamiento de una red neuronal de clasificación de imágenes convencional.

#### **3.2 OBJETIVOS ESPECÍFICOS**

Los objetivos específicos de este trabajo se listan a continuación:

- Estudio de algunas redes GAN ya existentes para su viabilidad para el trabajo propuesto.
- Desarrollo de una red GAN mediante la modificación de redes GAN ya existentes que sea viable para el trabajo propuesto.
- Generación de gran cantidad de imágenes por la red GAN para el reentrenamiento de la red clasificadora.
- Reentrenamiento y estudio comparativo de la red clasificadora con y sin imágenes adicionales de la red GAN.
- Evaluación de los resultados obtenidos.

## 4. ESTADO ACTUAL

En este apartado se hablará sobre el estado actual de la inteligencia artificial, las redes neuronales y más concretamente sobre las redes neuronales convolucionales y las redes generativas antagónicas.

### 4.1 INTELIGENCIA ARTIFICIAL

La inteligencia artificial ha sido noticia y lo sigue siendo en los últimos tiempos debido a los cambios que puede conllevar en el mundo moderno. La inteligencia artificial es un intento de emular la inteligencia humana usando un software o sistema. Con esta simple definición se pueden distinguir cuatro tipos de inteligencia artificial:

- Sistemas que piensan como humanos, las redes neuronales.
- Sistemas que actúan como humanos, los sistemas robóticos.
- Sistemas que usan la lógica racional, los sistemas expertos.
- Sistemas que actúan racionalmente, los agentes inteligentes.

En general, la inteligencia artificial es un campo científico en la informática que se enfoca en crear programas y sistemas que pueden mostrar comportamientos considerados inteligentes. Normalmente son capaces de procesar grandes cantidades de datos (Big Data), identificar patrones o formular predicciones.

Esta inteligencia artificial está presente en el día a día de la mayoría de personas, en las recomendaciones de compras por internet, en la muestra de resultados de búsqueda en un navegador web, en las traducciones automáticas, en los vehículos, etc. Debido a la amplitud de este campo, las aplicaciones son muy numerosas, aunque si nos fijamos en el ámbito de la industria, podemos destacar su uso en la robótica, mecatrónica, procesamiento de datos, control de procesos, simulación de procesos, modelado, toma de decisiones, etc. Estas aplicaciones son algunas de las que definen la industria 4.0, es decir, la cuarta revolución industrial en la que los sistemas automatizados e informáticos son los protagonistas. Es por esto que en los últimos años se ha podido observar que la inversión en estas tecnologías y desarrollos ha aumentado de manera descomunal [14].

#### 4.1.1 Machine learning.

El machine learning (aprendizaje automático) es una técnica de inteligencia artificial la cual busca el desarrollo de métodos para que permiten aprender a una

computadora a realizar una actividad. Este se basa en el análisis de datos y sus resultados hasta que se obtiene un algoritmo matemático que se comporte de manera similar al introducir nuevos datos. Los problemas que se resuelven son problemas de identificación, clasificación y predicción. Esta técnica trabaja con algoritmos de regresión o árboles de decisión.

#### 4.1.1.1 Deep learning.

El Deep learning es una rama del machine learning, en la cual lo que principalmente cambia son los algoritmos utilizados. Se puede considerar una evolución del machine learning, en el cual se trata de imitar el aprendizaje humano, utilizando otros algoritmos y sistemas para este aprendizaje. Estos trabajan normalmente con redes neuronales, no con algoritmos de regresión como en el caso del machine learning (figura 1).

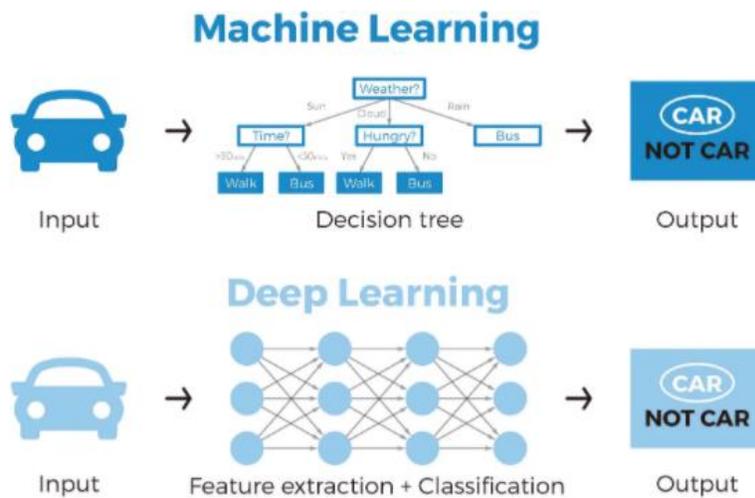


Figura 1.- Diferencia entre machine learning y Deep learning (Fuente: [16]).

Esta técnica usa una cascada de capas con unidades de procesamiento, conectando las salidas con las entradas de las anteriores. Pueden utilizar aprendizaje supervisado (en el cual los datos consisten en pares de objetos, los datos de entrada y los resultados esperados) o no supervisado (los datos de entrada son variables aleatorias y se construye un modelo de densidad para este conjunto) y las principales aplicaciones son la modelización de datos y reconocimiento de patrones [16].

## 4.2 REDES NEURONALES

Las redes neuronales artificiales son un campo importante dentro de la Inteligencia Artificial, las cuales se inspiran en el cerebro humano para crear modelos computacionales. Éstas, al igual que el cerebro humano, se basan en la unión de neuronas entre sí para trabajar en grupo y aprender mediante el refuerzo de estas

conexiones a realizar algunas tareas con la experiencia. En el modelo computacional, la información va pasando por estas “neuronas” donde se realizan diferentes operaciones, produciendo una salida.

#### 4.2.1 Neuronas artificiales, la unidad básica.

La neurona biológica tiene tres partes principales, las dentritas, las cuales son los canales de entrada de información a la neurona que están unidas con las salidas de otras neuronas, los axones, que son la salida de información de cada neurona, unidos a las dentritas de otras y el soma, que son la parte que “procesa” dicha información. Una neurona puede tener unas 10000 entradas y cientos de salidas, esta conexión entre neuronas se llama sinapsis, es unidireccional y se produce de forma eléctrica y química gracias a sustancias específicas llamadas neurotransmisores [15].

El modelo estándar de una neurona artificial se puede observar en la figura 2.

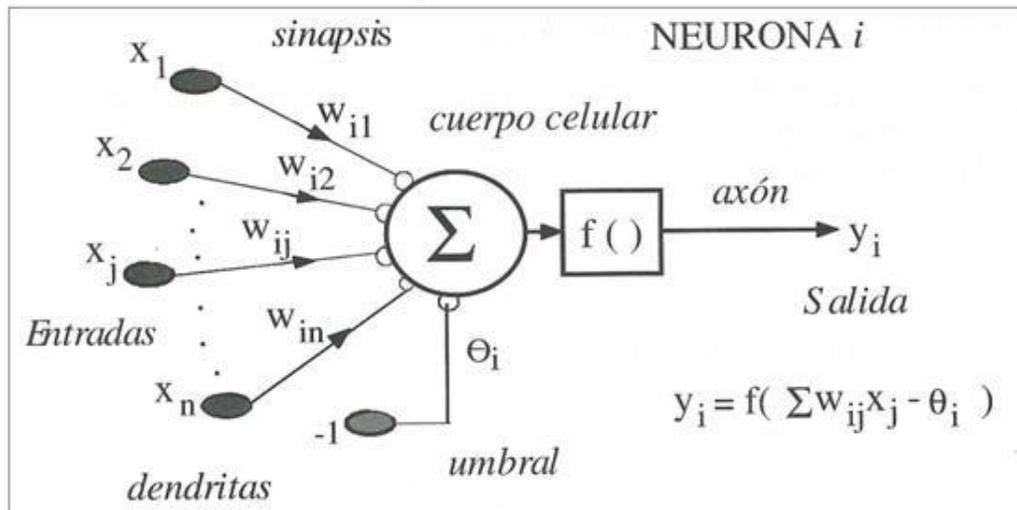


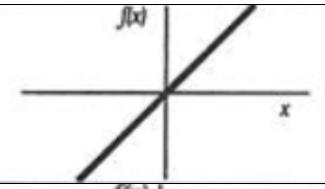
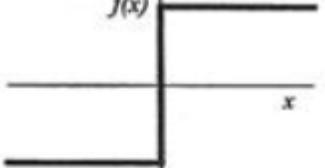
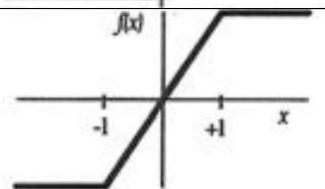
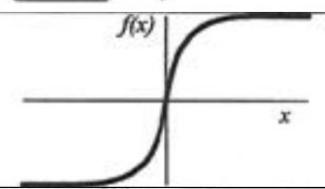
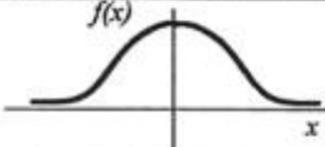
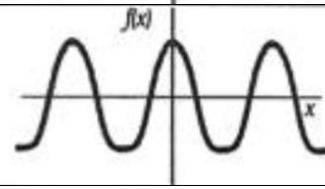
Figura 2.- Modelo genérico de neurona artificial (Fuente: [15]).

Los elementos visibles en este modelo son:

- Entradas:  $x_j(t)$ , pueden ser binarias o continuas dependiendo del modelo.
- Pesos sinápticos:  $w_{ij}$ , que representan la intensidad de interacción de cada neurona entrante.
- Regla de propagación:  $h_i(t) = \sigma(w_{ij}, x_j(t))$ , la más común suele ser la suma ponderada de las entradas con sus pesos sinápticos, es decir:  $h_i(t) = \sum w_{ij} x_j$ .
- Función de activación:  $y_i(t) = f_i(h_i(t))$ , que representa a la vez la salida de la neurona y su estado de activación.

En muchas ocasiones se introduce un parámetro  $\theta_i$ , que se denomina umbral, el cual se resta en la función de activación como se puede observar en la figura 2, esto hace que la salida no se active a no ser que la suma ponderada sobrepase el umbral [15].

En cuanto a las funciones de activación  $f_i$  existen numerosos tipos, se resumen las más utilizadas en la tabla 1.

Función	Ecuación	Rango	Gráfica
Identidad	$y = x$	$[-\infty, \infty]$	
Escalón	$y = \text{sign}(x)$ $y = H(x)$	$\{-1, 1\}$ $\{0, 1\}$	
Lineal a tramos	$y = \begin{cases} -1, & \text{si } x < -l \\ x, & \text{si } -l \leq x \leq l \\ 1, & \text{si } x > l \end{cases}$	$[-1, 1]$	
Sigmoidea	$y = \frac{1}{1 + e^{-x}}$ $y = \text{tgh}(x)$	$[0, 1]$ $[-1, 1]$	
Gaussiana	$y = Ae^{-Bx^2}$	$[0, 1]$	
Sinusoidal	$y = A \text{sen}(\omega x + \varphi)$	$[-1, 1]$	

**Tabla 1.- Funciones de activación más utilizadas en los modelos de neuronas artificiales dentro de las redes neuronales artificiales (Fuente: [15]).**

En la gran mayoría de casos, la función de salida es simplemente la identidad, de modo que la activación de la neurona se considera la propia salida. Esta salida como se ha comentado anteriormente, será la entrada de otra neurona (figura 3), formando así una red neuronal artificial.

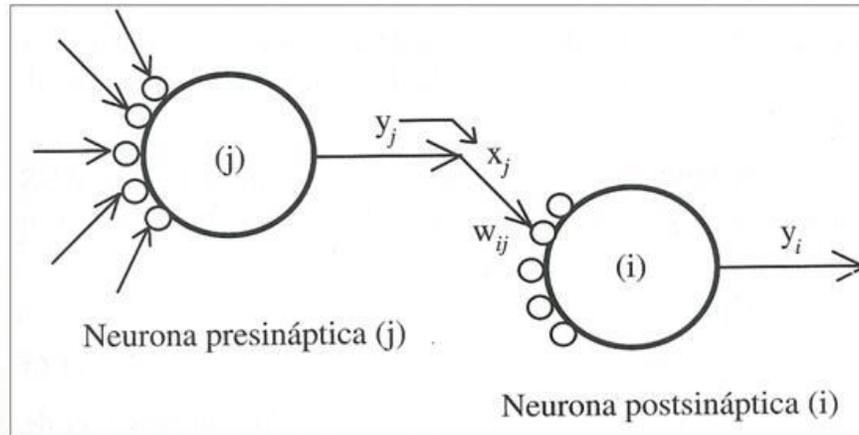


Figura 3.- Conexión entre dos neuronas artificiales en una red neuronal (Fuente: [15]).

Una vez vista la unidad básica de las redes neuronales, se pasa a analizar las redes, cómo se conectan y como “aprenden”.

#### 4.2.1 Arquitectura de redes neuronales.

La estructura de las redes neuronales se define por la forma en la que se conectan un número variable de neuronas artificiales. Estas conexiones sinápticas determinan el comportamiento de la red en general, siendo uno de los factores más importantes en el comportamiento. Las conexiones son direccionales, la información fluye en un solo sentido (de la neurona presináptica a la postsináptica), agrupándose en unidades estructurales llamadas capas. Dentro de cada capa las neuronas se pueden agrupar en grupos neuronales y dentro de estos grupos y de las capas las neuronas suelen ser del mismo tipo. Un conjunto de capas forma una red neuronal [15].

Existen tres tipos de capas:

- Capa de entrada: Neuronas que reciben los datos del entorno.
- Capa oculta: Neuronas intermedias en la red, que reciben los datos de la capa de entrada y los transforman hasta la capa de salida.
- Capa de salida: Neuronas que proporcionan la respuesta de la red neuronal.

También pueden existir conexiones intracapa, las cuales conectan neuronas de la misma capa entre sí, a diferencia de las más normales, las intercapa, las cuales conectan neuronas de distintas capas.

Una red neuronal se puede definir como un grafo matemático, donde a cada nodo se le asocia una variable de estado, un umbral y una función que define el nuevo estado del nodo (dependiente del estado, umbral y pesos de sus conexiones),

mientras que a cada conexión se le asocia un peso sináptico como se ha visto anteriormente.

Existen multitud de tipos de redes neuronales, ya que existen multitud de formas de conectar sus elementos básicos. Se hablarán de las topologías mas comunes, por el tema de este trabajo en los siguientes apartadaos, de las redes que tratan con imágenes, las redes convolucionales y por último de las redes GAN (Generativas Antagónicas).

#### 4.2.1.1 Perceptrón (Rosenblatt, 1959).

Se inspira en el procesamiento de algunos sistemas sensoriales de animales, la información atraviesa sucesivas capas de neuronas que realizan un procesamiento cada vez de mayor nivel.

Es un modelo unidireccional formado por dos capas de neuronas, entrada y salida. Las neuronas de entrada no realizan ninguna operación, solo pasan la información a las neuronas de salida. Las neuronas de salida tienen una función de activación tipo escalón o Heaviside (H) de forma que al realizar un sumatorio de las entradas a la neurona con un peso sináptico, si se pasa el valor umbral, tendrá un valor y si no otro de manera discreta [15].

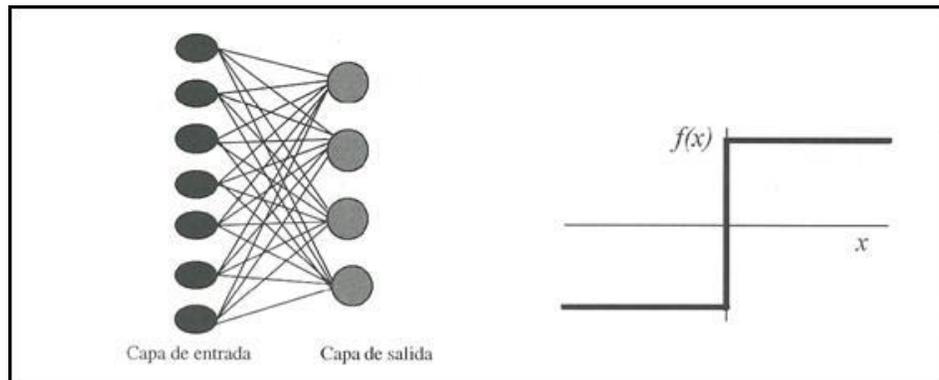


Figura 4.- Perceptrón simple (Fuente: [15]).

#### 4.2.1.2 Redes neuronales monocapa.

Es la más sencilla, consta de una capa de neuronas que mandan las entradas a una capa de salida donde se realizan todos los cálculos (figura 4).

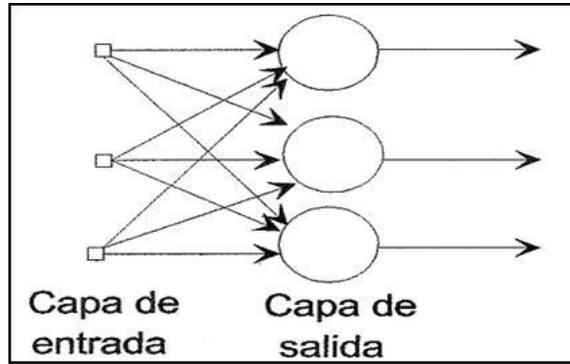


Figura 5.- Red neuronal Monocapa (Fuente: [15]).

#### 4.2.1.3 Redes neuronales multicapa.

En este caso, a diferencia del caso anterior, se introducen capas intermedias entre la entrada y la salida. Estas capas intermedias se denominan capas ocultas y pueden estas totalmente (todas las neuronas de una capa reciben datos de todas las neuronas de la capa anterior) o parcialmente (no todas las neuronas de una capa reciben datos de todas las neuronas de la capa anterior) conectadas.

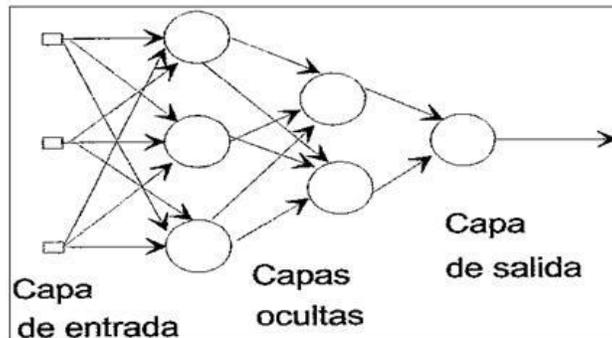


Figura 6.- Red neuronal multicapa (Fuente: [15]).

#### 4.2.1.4 Redes neuronales recurrentes.

En estas redes existen conexiones de realimentación, pueden ser entre neuronas de una misma capa o de diferentes capas, incluso entre una misma neurona. Esta topología las hace adecuada para sistemas no lineales.

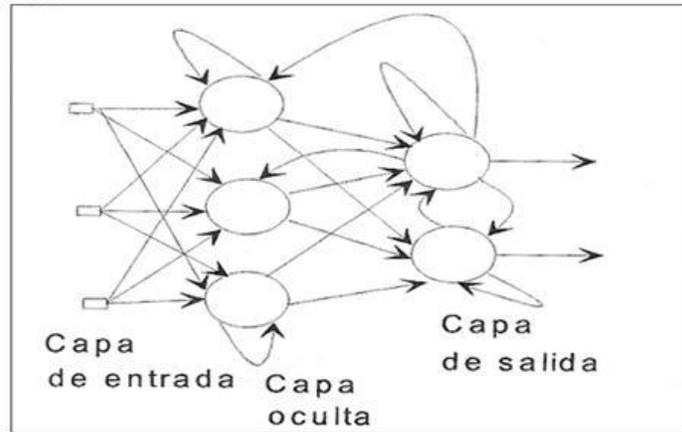


Figura 7.- Red neuronal recurrente (Fuente: [15]).

#### 4.2.1.5 Redes neuronales competitivas.

Existen conexiones entre neuronas de la misma capa, normalmente de salida, las neuronas compiten entre sí, de forma que se activará la salida de una, inhibiendo las demás.

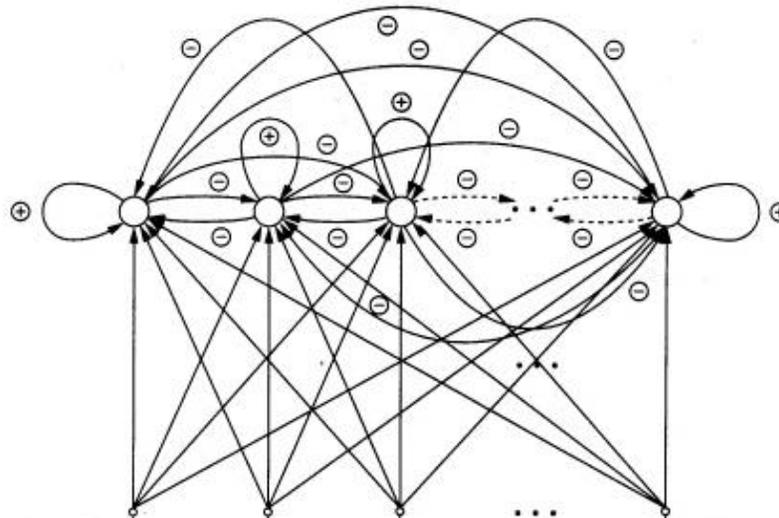


Figura 8.- Red neuronal competitiva (Fuente: <https://es.slideshare.net/mentelibre/redes-neuronales-aprendizaje-competitivo-cooperativo>).

#### 4.2.1 Aprendizaje de las redes neuronales.

El aprendizaje de una red neuronal se puede definir como el proceso para ajustar los parámetros variables de la red mediante la estimulación de ésta por el entorno que rodea la red. En general, se trata de determinar los pesos sinápticos de las conexiones que hagan que la red realice de manera correcta el tipo de tarea que queramos.

Este aprendizaje requiere partir de unos pesos sinápticos iniciales (aleatorios o nulos) y de una estructura de red determinada. Después, se puede entrenar de dos maneras:

- Modificando los pesos sinápticos mediante una regla de aprendizaje determinada, normalmente mediante la optimización de una función de coste que mide la eficiencia de la red, es decir, si las salidas obtenidas con ciertas entradas se acercan más o menos a las deseadas. Normalmente este proceso es iterativo, actualizándose los pesos sinápticos en cada iteración hasta que la red alcanza la eficiencia requerida.
- Creando y destruyendo neuronas, lo que implica una modificación de la arquitectura de la red.

Existen varios tipos de aprendizaje (o entrenamiento) que se basan en métodos numéricos iterativos para reducir una función de coste, que a veces pueden ocasionar problemas de convergencia en el algoritmo. Esta convergencia puede hacer comprobar si una arquitectura con su aprendizaje es capaz de obtener una cierta salida o no.

- Aprendizaje supervisado: Se proporciona a la red un conjunto de datos de entrada y las salidas deseadas, con éstos la red iterativamente ajusta sus pesos sinápticos hasta que la salida obtenida tiende a la deseada, con información del error cometido en cada iteración.
- Aprendizaje no supervisado: Se puede describir como la estimación de una función densidad de probabilidad que describen los datos de entrada. En este aprendizaje se proporciona a la red solo los datos de entrada, sin dar la respuesta que se espera y la red, mediante la función densidad de probabilidad, podrá reconocer patrones o rasgos de ese conjunto de datos.
- Aprendizaje híbrido: Coexisten el aprendizaje supervisado y no supervisado en la misma red, usualmente en distintas capas de neuronas.
- Aprendizaje reforzado: En este caso, existe una medida de error para la red, que represente la eficiencia global de la red, es decir, lo bien o mal que está respondiendo, sin darle una salida esperada para unos datos de entrada.

Es importante tener en cuenta el error que comete la red al final del aprendizaje para datos que no son de entrenamiento, esto nos da una idea de la generalización de la red, es decir, de su uso para datos y patrones que, aunque son de la misma naturaleza que los que se utilizan para entrenarla, son nuevos [15].

#### 4.2.1.1 Algoritmo “Backpropagation”.

Como resulta obvio, la salida de una neurona no dependerá únicamente de las entradas de la red, si no que dependerá de las salidas de otras neuronas intermedias que le lleguen como entrada, por ello, se puede decir que el error cometido por una neurona no solo depende de sus pesos sinápticos, si no que dependerá también de los errores cometidos por otras neuronas precedentes. Esto se puede observar como una propagación del error desde la primera capa hasta la última.

Al aplicar una entrada, esta se propaga por la red hasta la salida, generando un error que, a su vez, se propaga hacia atrás (de ahí el nombre de “Backpropagation”), haciendo que cada neurona de las capas ocultas reciban una fracción del error basado en la contribución de cada neurona oculta en la salida. Esto hace que todas las neuronas modifiquen sus pesos según dicho error y dicha contribución y se obtenga así un aprendizaje basado en el cálculo del gradiente mediante este algoritmo en un aprendizaje supervisado [15].

### 4.3 REDES NEURONALES CONVOLUCIONALES

Las redes convolucionales son el algoritmo más ampliamente utilizado para trabajar con imágenes en el campo de las redes neuronales. Con ellas se pueden clasificar imágenes, detectar patrones en imágenes, detectar anomalías en imágenes, etc.

Éstas son un tipo de red neuronal de aprendizaje supervisado que imita al cortex visual del ojo humano para identificar características en las entradas, que son las imágenes y con ello identificar objetos. Poseen varias capas ocultas que se especializan en diferentes problemas y con jerarquía, desde las primeras capas que son capaces de reconocer líneas y curvas hasta que se van especializando en patrones más complejos como formas o patrones de formas.

Normalmente se requiere una gran cantidad de imágenes para un entrenamiento adecuado, llegando a ser muy complicado en algunos obtener una buena efectividad debido a la escasez de imágenes, ya que se pueden requerir del orden de más de 10000 imágenes de una misma clase para el entrenamiento, dependiendo de la dificultad del problema [3].

Los píxeles de una imagen se pueden ver como un valor entre 0 y 1 (de 0 a 255 se normaliza para la red) y una imagen como una matriz de esos valores como se puede observar mas gráficamente en la figura 8.

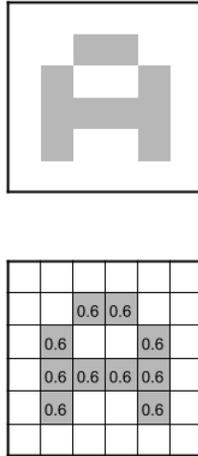


Figura 9.- Imagen y su descomposición en píxeles (Fuente: [3]).

La red tendrá como entrada tantas neuronas como píxeles posea la imagen, eso si la imagen es de un solo canal de color (escala de grises). Si es una imagen a color, se triplicarán el número de neuronas, ya que se tendrán tres canales de color (RGB), siempre sin olvidarse de normalizar los valores de cada canal para adecuarlos a la red.

Una vez vista como sería la capa de entrada, se pasa a ver cómo se organizan las capas intermedias, y aquí es donde comienzan las convoluciones. Estas convoluciones tratan de tomar grupos de píxeles cercanos de la imagen y realizar un producto escalar con una pequeña matriz llamada *kernel*. Esta matriz recorre todas las neuronas de entrada y genera una nueva matriz de salida, que será la nueva capa oculta de la red, como se puede observar en la figura 10.

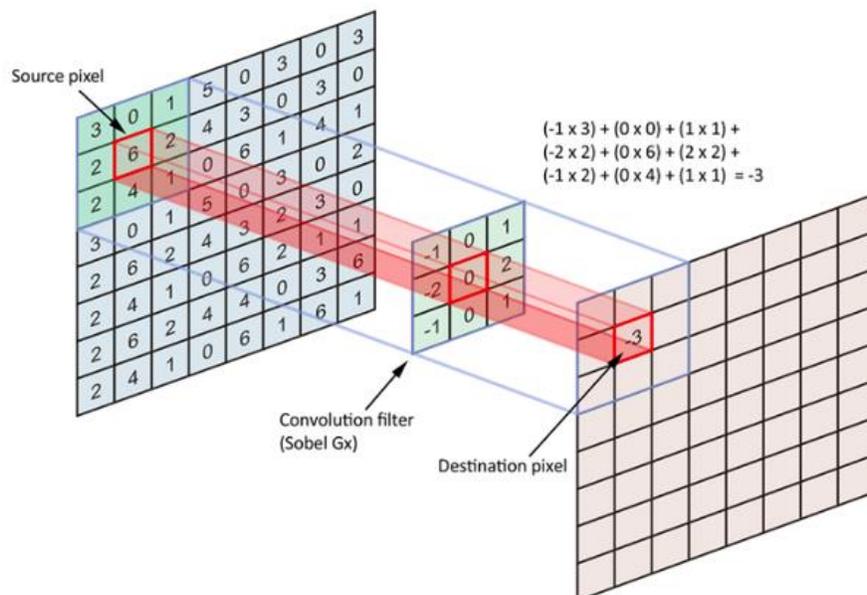


Figura 10.- Proceso de convolución (Fuente: <https://es.quora.com/C%C3%B3mo-funcionan-las-redes-neuronales-convolucionales>).

Pero no solo se aplica un kernel en cada convolución, si no que se aplicará un conjunto de ellos, a este conjunto se le llama filtros. A las matrices de salida obtenidas al aplicar estos filtros se les conoce como “feature mapping” o mapa de características, para simplificar es como si tuviéramos tantas imágenes nuevas como filtros se apliquen y estas imágenes representan ciertas características de la imagen original que después ayudarán a distinguir objetos en la imagen. Esto hace que la siguiente capa multiplique sus neuronas de manera considerable. Normalmente en estas capas, las neuronas poseen una función de activación ReLu (Rectifier Linar Unit) que consiste en la función  $f(x) = \max(0,x)$  [3].

Despues de cada convolución debemos reducir el número de neuronas antes de realizar una nueva convolución, ya que se ha visto que el número de neuronas crece exponencialmente con el número de convoluciones y si no se realiza este paso, al final quedaría una red con millones y millones de neuronas muy difícil de entrenar. Este proceso se llama “subsampling” y consiste en reducir el tamaño de las imágenes resultantes de la convolución, quedándose con las características más importantes detectadas por cada filtro. Existen diferentes tipos de subsampling, pero el más usado es el “Max Pooling”.

El proceso de Max Pooling consiste en recorrer las imágenes de características obtenidas por la convolución y dividir las en sectores según el tamaño de matriz que se configure. Una vez divididas, se toma el máximo valor presente en cada sección y se guarda en una matriz, reduciendo el tamaño de las imágenes considerablemente y con ello la cantidad de neuronas requeridas.

Una vez se realiza este paso, se pasa a realizar más convoluciones con los mismos pasos, obteniendo así nuevos mapas de características capaces de reconocer formas más complejas. Este proceso se realiza hasta tener en la salida de la convolución mapas de características pequeñas y que puedan ser procesadas.

El último paso será pasar estas matrices a una red neuronal multicapa tradicional para conseguir una clasificación o reconocimiento del objeto u objetos que hay en la imagen. Para ello, estas matrices se aplanan, es decir, dejan de tener tres dimensiones a ser un vector de valores que pasarán a las neuronas de entrada de la red tradicional multicapa, que aplicándole una función de activación llamada “Softmax” (figura 11) conecta la capa a la salida final que tendrá tantas neuronas como clases de imágenes se quieran clasificar, dando una probabilidad en cada salida de que sea ese objeto [3].

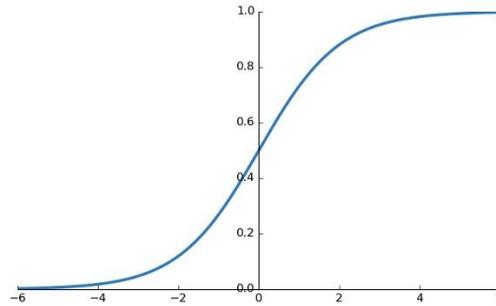


Figura 11.- Función de activación Softmax (Fuente: <https://themaverickmeerkat.com/2019-10-23-Softmax/>).

En la figura 12 se puede observar un croquis de estas redes para el caso de clasificación de imágenes de perros, gatos y pájaros.

## ARQUITECTURA DE UNA CNN

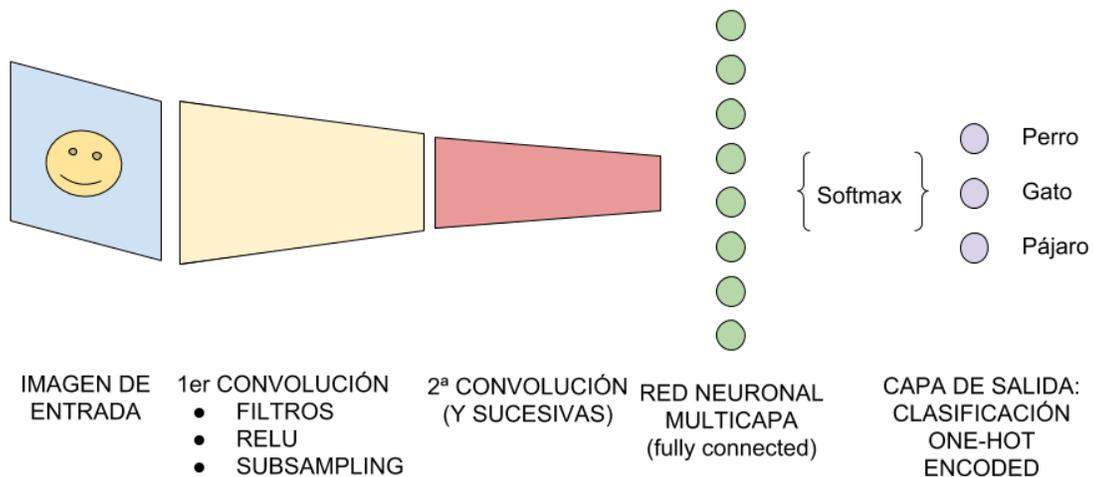


Figura 12.- Esquema de arquitectura de una red neuronal convolucional para la clasificación de imágenes (Fuente: [3]).

### 4.4 REDES GENERATIVAS ANTAGÓNICAS (GAN)

Las redes GAN, aunque son de desarrollo muy reciente (Goodfellow 2014), se han convertido en una parte muy importante en el crecimiento de la inteligencia artificial. Éstas son una herramienta muy interesante pero requiere conocimientos más avanzados en redes neuronales.

Estas redes consisten principalmente en un sistema de dos redes neuronales que compiten entre sí (de ahí la palabra antagónicas), el generador y el discriminador.

Para su funcionamiento se requiere una serie de muestras que se quieren como resultado, el generador intentará reproducir estas muestras y el discriminador se encargará de discernir si estas son muestras reales o no. Mediante este entrenamiento basado en la competición de las dos redes, el generador intentando “engañar” al discriminador, al final el generador consigue generar muestras muy similares a las reales y que sea capaz de engañar a la red discriminadora. Con esto, el generador ha sido capaz de realizar una creación artificial de los datos que se hayan utilizado [1].

Las aplicaciones de estas redes son numerosas, ya que son capaces de aprender a generar distribuciones de datos de cualquier tipo, imágenes, audio, video, datos e incluso partituras.

Aunque también existen dificultades al trabajar con estas redes, la principal es su estabilidad inconstante. Esto quiere decir que estas redes se “estancan” con facilidad en un tipo de resultados, dando lugar a que no se alcancen los resultados esperados, aunque últimamente se han desarrollado soluciones para algunos de estos problemas.

Adentrándose más en el algoritmo de las redes GAN, podemos observar algoritmos discriminatorios y generativos. Los algoritmos discriminatorios clasifican los datos de entrada, obteniendo unas características predicen una categoría a la que pertenecen estos datos (en el caso de imágenes, una red convolucional como se ha visto anteriormente). En este caso, clasifican entre si los datos son reales o falsos, es una clasificación binaria.

Los algoritmos generativos son al contrario, en lugar de predecir una categoría, predicen unas características de una categoría determinada. Esto es un claro ejemplo de aprendizaje no supervisado, donde a partir de unos datos de entrada, se genera una distribución de probabilidad de esos datos, que mediante el discriminador va cambiando hasta que se consigue “engañar” y pasar esos datos como datos reales. Todo esto se puede ver como una especie de juego de suma cero. Este funcionamiento se puede observar de manera gráfica en la figura 13.

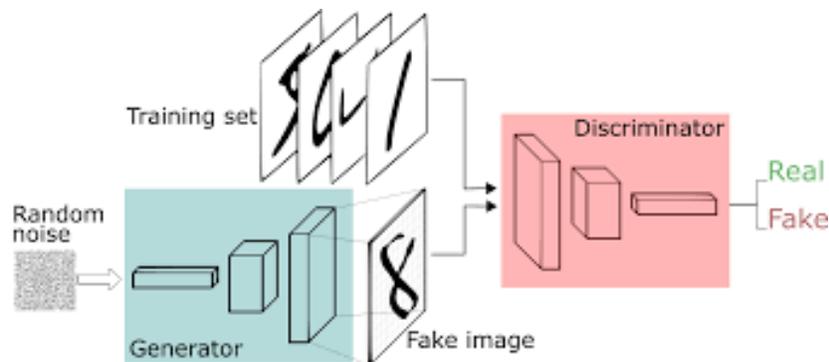


Figura 13.- Esquema de funcionamiento de una red GAN (Fuente: <https://proyectoidis.org/red-generativa-antagonica-gan/>).

Como se puede observar en la figura 13, en el caso de que se trabajen con imágenes, se requieren imágenes de un “training set” o “dataset” las cuales son imágenes reales como ejemplos de las imágenes que queremos generar.

Este trabajo se centrará en las redes GAN orientadas a la generación de imágenes, utilizando para ellos sistemas de convolución como las vistas anteriormente, con ellos aparece una de las primeras redes GAN en utilizar redes neuronales convolucionales, las redes DCGAN (Deep Convolutional Generative Adversarial Networks). Estas redes convierten 100 números aleatorios con una distribución uniforme en una imagen deseada. Consiste en capas de convolución y de desconvolución, el discriminador trabajará como una red neuronal de convolución vista anteriormente, donde las capas convolutivas generan mapas de características de la imagen a analizar y por último clasifican estas características en las categorías de real o falsa, mientras que el generador trabajará con capas desconvolutivas, es decir, de una distribución aleatoria de datos, se irán generando mapas de características de diferentes dimensiones hasta conseguir una distribución de las dimensiones adecuadas (con tres canales de color si es una imagen RGB y uno si es en escala de grises) que mostrará una imagen [9].

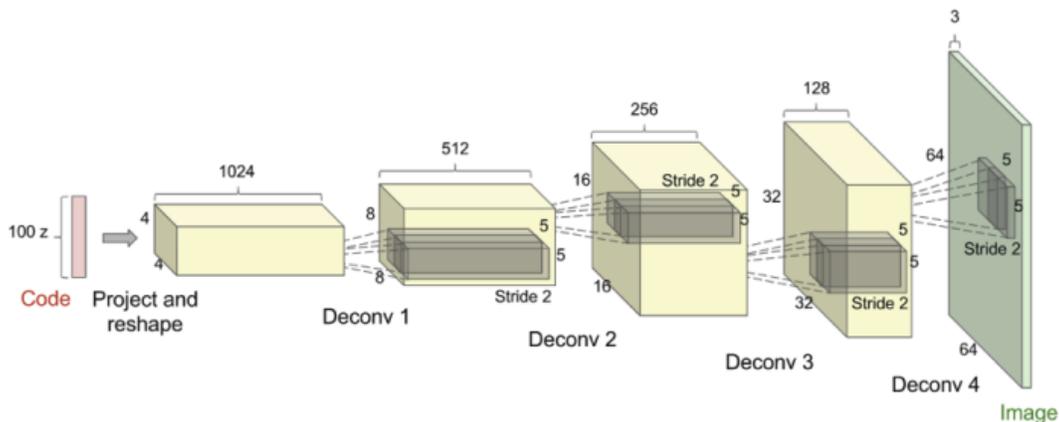


Figura 14.- Modelo desconvolucional de un generador en una red GAN (Fuente: [9]).

El aprendizaje de las dos redes se realiza por separado, en una primera instancia se entrena el discriminador con datos reales y falsos, actualizando sus pesos según la corrección de sus predicciones. El entrenamiento del generador se fija en las predicciones del discriminador en cuanto a los datos falsos, si el discriminador ha sido capaz de discernir que son falsos, se tendrá un error y con ello se modificarán los pesos de la red generador, por lo que el entrenamiento del generador depende del clasificador mientras que no ocurre al contrario, el entrenamiento del discriminador no depende del generador [7].

Algunos de los inconvenientes que pueden surgir en el uso de estas redes y que pueden ocasionar malos resultados de las mismas son [1]:

- Colapso modal: Los datos generados tienden a agruparse en un conjunto limitado de modos debido a que el generador es capaz de engañar al discriminador bloqueando los datos en un solo sin explorar los otros, en el caso de las imágenes, se generan imágenes de una misma clase o forma sin explorar otras variables.
- Convergencia: En el caso de las redes GAN no podemos saber cuándo parar el aprendizaje mediante su función de coste, por lo que la convergencia es un problema habitual.
- Calidad: Es difícil saber cuándo la calidad de los datos generados es lo suficientemente buena, normalmente el operador humano juzga esa calidad.
- Métrica: No se tiene una métrica concreta para la calidad y variedad de los resultados obtenidos.

En el desarrollo de este trabajo se verá en más detalle como construir una red GAN y los parámetros que aparecen en ella, así como se verán los problemas frecuentes que aparecen con estas redes.

## **4.5 HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO DEL TRABAJO**

En este apartado se muestra el lenguaje de programación y herramientas utilizadas para el desarrollo del trabajo.

### 4.5.1 Python.

Python es un lenguaje de programación interpretado destacable por la legibilidad de su código, es multiparadigma, soporta programación orientada a objetos, imperativa y funcional aunque en menor medida.

Pero además posee algunas características que lo hacen idóneo para la programación de redes neuronales, como son las iteraciones rápidas de datos que favorecen el desarrollo de los algoritmos o como son las amplias librerías de data science o matemáticas como son NumPy (que da soporte para el manejo de vectores y matrices multidimensionales) y ScyPy (que contiene herramientas y algoritmos matemáticos).

También existen librerías determinadas que están mas enfocadas al desarrollo de machine learning que se pasan a exponer a continuación.

#### 4.5.1.1 Scikit.

Es una librería de aprendizaje automático que incluye algoritmos de clasificación, regresión, clustering y reducción de dimensionalidad que la hace una herramienta muy útil para estructurar sistemas de análisis de datos y modelado estadístico. Contiene una gran variedad de módulos y algoritmos.

#### 4.5.1.2 OpenCV.

Es una librería de visión por computación y aprendizaje automático de código abierto. Contiene más e 2500 algoritmos para funciones como la detección de movimientos en video, reconocimiento de objetos, reconstrucción 3D, segmentación, realidad aumentada, etc.

Es multiplataforma, ofrece soporte para Windows, Linux, MacOs, Android e iOS, y es compatible con varios lenguajes de programación, siendo C y C++ el lenguaje óptimo para utilizarlo, aunque en Python ha demostrado ser también eficiente. Otros lenguajes soportados son Java, Matlab, Octave o Javascript.

#### 4.5.1.3 Tensorflow.

Es una librería de código abierto para aprendizaje automático desarrollado por Google para construir y entrenar redes neuronales en general. Principalmente proporciona una API para Python, pero también existen para Java, C++, Go, Rust, Haskell, C#, R, Julia y OCaml. Cuenta con un ecosistema flexible e integral de herramienta y recursos de la comunidad para desarrollar, compilar e implementar tecnología de aprendizaje automático.

Posee una arquitectura simple y flexible, se puede implementar prácticamente en cualquier dispositivo y compila y entrena modelos fácil e intuitivamente con ejecución inmediata.

Está diseñada para realizar cálculo de vectores (que en esta librería se denominan tensores) numéricos de manera muy eficiente, lo que lo hace idóneo para el manejo de datos en las redes neuronales.

#### 4.5.1.4 Keras

Es una biblioteca de código abierta en Python cuyo objetivo es acelerar la creación de redes neuronales, para ello utiliza una interfaz de uso intuitivo que permite acceder a varios entornos de trabajo de aprendizaje automático y desarrollarlos.

Esta proporciona bloques modulares para desarrollar modelos complejos de Deep Learning, las capas de la red neuronal se conectan entre si de manera modular simplificando en gran medida el proceso de creación de las redes neuronales. Está diseñado para las personas y no tanto para las máquinas, haciendo que las acciones del usuario se reduzcan y proporcionando feedback en caso de error.

## 5. DESARROLLO

En este apartado se presentará el desarrollo realizado para llegar a los resultados obtenidos de este trabajo, primero se presentarán los ejemplos desarrollados obtenidos de diferentes fuentes, explicando en qué consisten cada uno de ellos y para qué problema están propuestos, y después se analizará el desarrollo realizado en estos para adaptarlos al problema propuesto.

### 5.1 EJEMPLOS DE REDES GAN DESARROLLADAS EN INTERNET

En este apartado se explicará la base de software de la que se parte para una adaptación posterior al problema de estudio para la generación de imágenes de golosinas y el posterior reentrenamiento de la red clasificadora de imágenes de golosinas para comprobar su mejora con un mayor número de imágenes de cada clase.

Se muestran dos ejemplos de software de redes GAN desarrolladas en internet con pruebas de éxito en sus respectivos casos.

#### 5.1.1 Red GAN generadora de números escritos a mano MNIST.

##### 5.1.1.1 Dataset MNIST

El dataset MNIST (Modified National Institute of Standards and Technology) es una gran base de datos de dígitos manuscritos que se usan normalmente para el entrenamiento de sistemas de procesamiento de imágenes, muy comunes en el campo del *machine learning*. Contiene 60000 imágenes de entrenamiento y 10000 de test.

En este caso las imágenes son en escala de grises, por lo que sólo tendrán un canal de color, y son de resolución de 28x28 píxeles. Los dígitos manuscritos van de 0 a 9, es decir, 10 clases, y están disponibles en los dataset de keras, por lo que mediante una función de tensorflow, se pueden obtener las imágenes, en este caso solo interesan las imágenes y no sus etiquetas, por lo que no se descargan, al igual que los datos de test. Podemos ver un ejemplo de estas imágenes en la figura 15, donde se puede observar que para cada dígito hay formas diferentes.

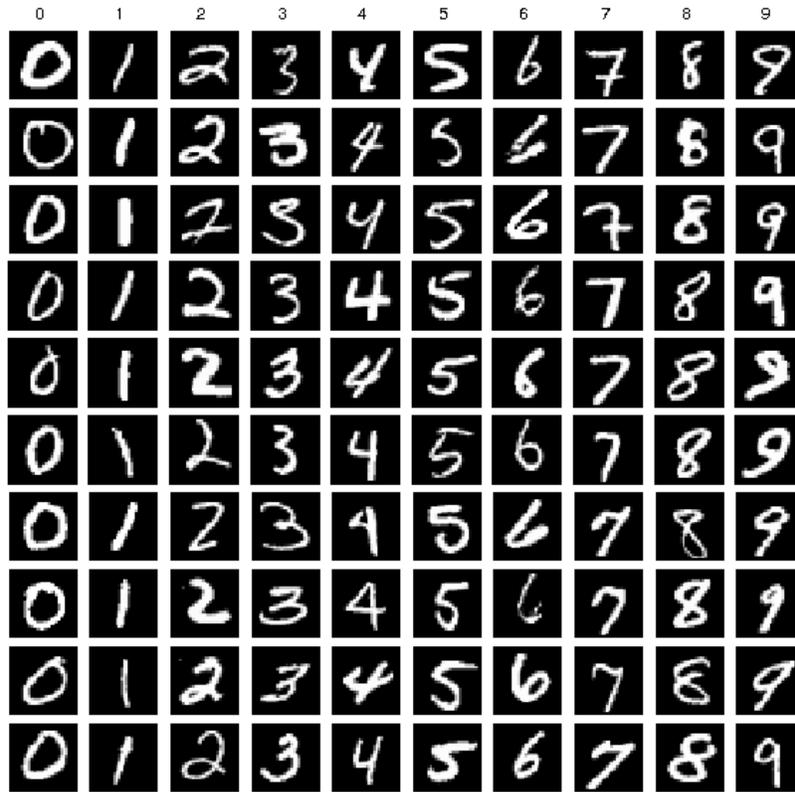


Figura 15.- Ejemplo de imágenes del dataset MNIST (Fuente: <https://deepai.org/dataset/mnist>).

Para tener preparado el dataset para la red GAN, lo primero que se deberá hacer es cargar los datos y darles un formato. Este formato será asegurarse de que las imágenes sean de 28x28x1 píxeles de resolución, tipo real y que los valores estén normalizados en el rango de [-1,1], en vez de [0,255]. Además se indica el tamaño del *buffer* de imágenes y del lote (*batch*) de imágenes que se entrenan a la vez. Una vez se realizan estas operaciones, se realiza un barajado de las imágenes y se separan en los lotes definidos [6].

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28,
1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256
# Batch and shuffle the data
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATC
H_SIZE)
```

### 5.1.1.2 Discriminador del modelo.

El discriminador del modelo es bastante simple, se basa en un clasificador de imágenes basado en redes convolucionales.

El discriminador de este modelo se encargará de determinar si la imagen analizada es real o falsa, lo cual convierte el problema en un problema de clasificación binaria, donde la entrada es la imagen y la salida es un 1 o un 0 según sea real o falsa [6].

En este caso, el discriminador desarrollado recibirá imágenes de 28x28 píxeles en escala de grises y su salida será un número entre 0 y 1 que indica la probabilidad de que la imagen sea real. Éste se forma por capas convolucionales, todas ellas de paso 2x2, lo cual produce una reducción en la resolución de la imagen hasta conseguir un vector de valores mediante una capa *Flatten*, la cual se encargará de pasar los valores de la última matriz de más baja resolución al vector de valores, y una capa *Dense* que se encargará de realizar con esos valores el último paso para la clasificación binaria.

También se utilizan las capas *Dropout* y *LeakyReLU*, la primera hace que ciertas neuronas sean ignoradas en la fase entrenamiento, esto hace que las neuronas no desarrollen co-dependencia entre ellas lo que da como resultado un sobreentrenamiento de la red y por lo tanto un peor funcionamiento, la segunda se usa como rectificador, es decir, hace que la salida este en el rango [0,X] evitando posibles errores en valores cercanos al 0 [11].

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1]))

    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

Si utilizamos la función *summary()* con este modelo obtenemos lo representado en la figura 16. En esta se observa que gracias a las capas convolucionales la imagen va bajando su resolución de 28x28 píxeles a 14x14y 7x7. Una vez se tiene una baja resolución, se utiliza la capa *Flatten* y se obtiene un solo vector de 6272 valores y que con la capa final *Dense* se obtiene un solo valor que nos dirá si la imagen es real o falsa.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 64)	1664
leaky_re_lu_3 (LeakyReLU)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	204928
leaky_re_lu_4 (LeakyReLU)	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 1)	6273
Total params: 212,865		
Trainable params: 212,865		
Non-trainable params: 0		

**Figura 16.- Resumen del modelo discriminador de la red GAN MNIST.**

Ahora hay que concretar cuál es la función de pérdida y el optimizador de este modelo, para ello se utiliza la función de pérdida de entropía cruzada (para la pérdida del generador y del discriminador), la cual es perfecta para medir el rendimiento de un modelo de clasificación binaria y nos la ofrece Keras.

Para la función de pérdida del discriminador, se desarrolla una función la cual cuantifica lo bien que distingue entre imágenes reales y falsas el discriminador. Para ello, compara las predicciones del discriminador en imágenes reales con una matriz de unos y las de imágenes falsas con una de ceros [6].

Se utiliza el optimizador proporcionado por Keras Adam.

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

### 5.1.1.3 Generador del modelo.

Este modelo se encargará de la generación de imágenes falsas de los objetos del dataset MNIST, esto se hace tomando un punto de un espacio latente como entrada y teniendo como salida una imagen en escala de grises del dataset MNIST.

Con este generador convertiremos un vector de 100 dimensiones en una matriz 2D de 28x28 con rango [-1,1]. Esto se hace mediante el enfoque de las redes

adversarias generativas convolucionales profundas que han demostrado ser efectivas.

Primero se utiliza una capa *Dense* oculta que tenga los suficientes nodos para representar una imagen en baja resolución de la imagen de salida, en este caso se usa una imagen de 7x7 píxeles en escala de grises, es decir, 49 nodos. Lo que se requiere de esta red es generar varias interpretaciones de la entrada, que es un patrón de las redes neuronales convolucionales en las cuales se tienen filtros paralelos que nos dan múltiples mapas de activación paralelos (mapas de características) con diferentes interpretaciones de la salida. Lo que se requiere en este caso es al contrario, requerimos diferentes versiones paralelas de la salida con diferentes características aprendidas que puedan comprimirse en la capa de salida en una imagen. Esto hace que la capa *Dense* de entrada tenga tantos nodos como versiones se quieren de la imagen de salida, en este caso 256.

Estas versiones se remodelan en algo parecido a una imagen mediante la capa *Reshape* con 256 mapas de características de 7x7.

Una vez se tiene esta “imagen” en baja resolución se puede pasar ahora a ir aumentando la resolución mediante capas convolucionales (capas *Conv2DTranspose*) que serán configuradas con un paso de 2x2 que cuadruplicarán el área del mapa de características de entrada. Al igual que en el caso de MNIST se recomienda usar un tamaño de núcleo (*kernel*) que sea un factor del paso para evitar patrones de tablero ajedrez que pueden aparecer en el proceso de aumento de resolución [13].

Esto se repite dos veces para llegar a la imagen de 28x28 píxeles que se quiere como salida. Cada paso, como en el caso anterior, esta seguido de una capa *LeakyReLU* para disminuir el error cometido y para finalizar una última capa convolucional (*Conv2DTranspose*) que dejará los datos preparados con forma de imagen y en el formato requerido, ésta tendrá la opción *'same' padding* (la cual nos dice que el mapa de características de la salida debe ser del mismo tamaño que el de la entrada, en este caso 28x28x1) y una función de activación *tanh* (tangente hiperbólica) para asegurar que los valores se encuentran en el rango [-1,1].

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch
size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
```

```

model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
assert model.output_shape == (None, 14, 14, 64)
model.add(layers.BatchNormalization())
model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))
assert model.output_shape == (None, 28, 28, 1)

return model

```

Si utilizamos la función `summary()` con este modelo obtenemos lo representado en la figura 17, donde se puede ver representado los pasos que se dan mediante las capas convolutivas desde la entrada vector de 100 dimensiones hasta la salida imagen de 28x28x1.

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 12544)	1254400
batch_normalization (Batch Normalization)	(None, 12544)	50176
leaky_re_lu (LeakyReLU)	(None, 12544)	0
reshape (Reshape)	(None, 7, 7, 256)	0
conv2d_transpose (Conv2DTranspose)	(None, 7, 7, 128)	819200
batch_normalization_1 (Batch Normalization)	(None, 7, 7, 128)	512
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 14, 14, 64)	204800
batch_normalization_2 (Batch Normalization)	(None, 14, 14, 64)	256
leaky_re_lu_2 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 28, 28, 1)	1600
Total params: 2,330,944		
Trainable params: 2,305,472		
Non-trainable params: 25,472		

**Figura 17.- Resumen del modelo generador de la red GAN MNIST.**

Si añadimos algo más de código, para crear el modelo y un vector de valores aleatorios de dimensión 100, podemos obtener una imagen generada que, al no estar entrenado el generador, será una imagen aleatoria completamente (figura 18).

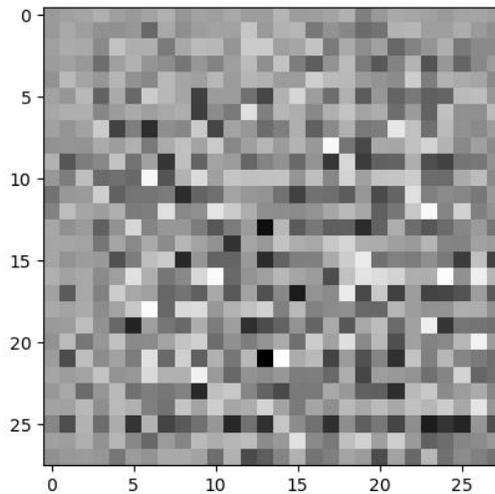


Figura 18.- Ejemplo de uso del modelo generador sin entrenar en el caso MNIST.

Ahora también hay que concretar cuál es la función de pérdida y el optimizador de este modelo, para ello, como en el caso del discriminador se utiliza la función de pérdida de entropía cruzada.

Para la función de pérdida del generador, se desarrolla una función la cual cuantifica lo bien se “engaña” al discriminador. Si el generador funciona correctamente, las imágenes falsas se clasificarán como reales, por lo que se comparará las decisiones del discriminador con imágenes generadas con una matriz de unos [6].

También se utiliza el optimizador de Keras Adam en este caso.

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

#### 5.1.1.4 Modelo GAN.

Una vez se tienen los modelos de discriminador y generador desarrollados, es hora de ver cómo entrenar el generador para obtener las imágenes falsas pero plausibles del dataset MNIST, aquí es donde entra la base de las redes antagónicas generadoras, y es el trabajo del discriminador como base del entrenamiento del generador.

Si el discriminador es bueno determinando si una imagen falsa es falsa, el generador se entrena para obtener imágenes que ‘engañen’ al discriminador, es decir, cuanto más bueno sea el discriminador en detectar una imagen falsa como falsa, el generador se actualiza más y cuando el discriminador no sea tan bueno, el generador se actualizará menos. Esta es la base de la relación adversaria de las redes GAN.

Para realizar este juego de suma-cero, se genera un nuevo modelo que contenga a los dos modelos. Este modelo recibirá de entrada los puntos aleatorios del espacio latente que pasarán al generador, la salida del generador será la entrada del discriminador, el cual los clasificará y la salida de este servirá para actualizar los parámetros del generador.

El discriminador se usará para clasificar imágenes reales (del dataset MNIST) e imágenes falsas (generadas), la pérdida calculada para cada modelo hará modificar los gradientes de cada modelo que actualizarán los modelos.

Para realizar este paso, se desarrolla una función que genera las imágenes falsas, pasa imágenes reales y falsas al discriminador y obtiene la pérdida correspondiente para cada caso, tras esto, actualiza unos gradientes para cada modelo y los aplica, actualizando los valores de los parámetros de los modelos y mejorando así su funcionamiento.

```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

        gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

Esta función servirá para cada paso de entrenamiento en cada lote (*batch*) de iteración (*epoch*) completa, añadiendo algo de código para generar y guardar las imágenes y el modelo en cada iteración, obtenemos el modelo GAN completo para el caso del dataset MNIST (código completo en el Anexo 1).

```
def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))
```

```

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix = checkpoint_prefix)

        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

# Generate after the final epoch
display.clear_output(wait=True)
generate_and_save_images(generator, epochs, seed)

```

Por último lo que se necesita es evaluar el modelo, viendo que las imágenes generadas son plausibles y para esto no hay maneras objetivas de hacerlo, por lo que será el factor humano el que determine la calidad de las imágenes generadas. Esto hace que no se puede parar el entrenamiento cuando las imágenes sean buenas, si no que se deberán ver ejemplos de estas imágenes cada cierta iteración y ver cómo van siendo los resultados. En estas iteraciones la imagen puede ir mejorando hasta que llegue a un punto donde no mejore más o incluso empiece a empeorar.

### 5.1.2 Red GAN CIFAR-10 generadora de pequeñas fotografías.

#### 5.1.2.1 Dataset CIFAR-10.

CIFAR es el acrónimo del Instituto Canadiense para la Investigación Avanzada y el dataset CIFAR-10 fue desarrollado por dicho instituto.

Este dataset está compuesto por 60000 imágenes a color de 32x32 píxeles de 10 clases de objetos (ranas, pájaros, camiones, aviones, barcos, etc.) las cuales son accesibles con una función de Keras la cual nos da la entrada y salida de los elementos de entrenamiento y de test para la red, repartiéndose 50000 y 10000 imágenes para cada conjunto respectivamente. Esta función automáticamente nos descargará el dataset comprimido y listo para usar [10].

En la figura 19 se puede observar un ejemplo de imágenes del dataset, se puede observar que la figura representada se encuentra en el medio y que las imágenes



Al igual que en el caso anterior se utilizan las capas *Dropout* y *LeakyReLU*, para evitar sobreentrenamiento y reducir errores respectivamente.

El entrenamiento del modelo se basa en minimizar la función de coste de entropía cruzada binaria, que es la más apropiada para un problema de clasificación binaria.

Este discriminador se define en una función mostrada a continuación:

```
def define_discriminator(in_shape=(32,32,3)):
    model = Sequential()
    # normal
    model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # classifier
    model.add(Flatten())
    model.add(Dropout(0.4))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=['accuracy'])
    return model
```

Se puede observar que la entrada es la imagen de 32x32 píxeles y 3 canales de color RGB.

Si utilizamos la función *summary()* con este modelo obtenemos lo representado en la figura 20. En esta se observa que gracias a las capas convolucionales la imagen va bajando su resolución de 32x32 píxeles a 16x16, 8x8 y 4x4. Una vez se tiene una baja resolución, se utiliza la capa *Flatten* y se obtiene un solo vector de 4096 valores y que con la capa final de activación sigmoide se obtiene un solo valor que nos dirá si la imagen es real o falsa.

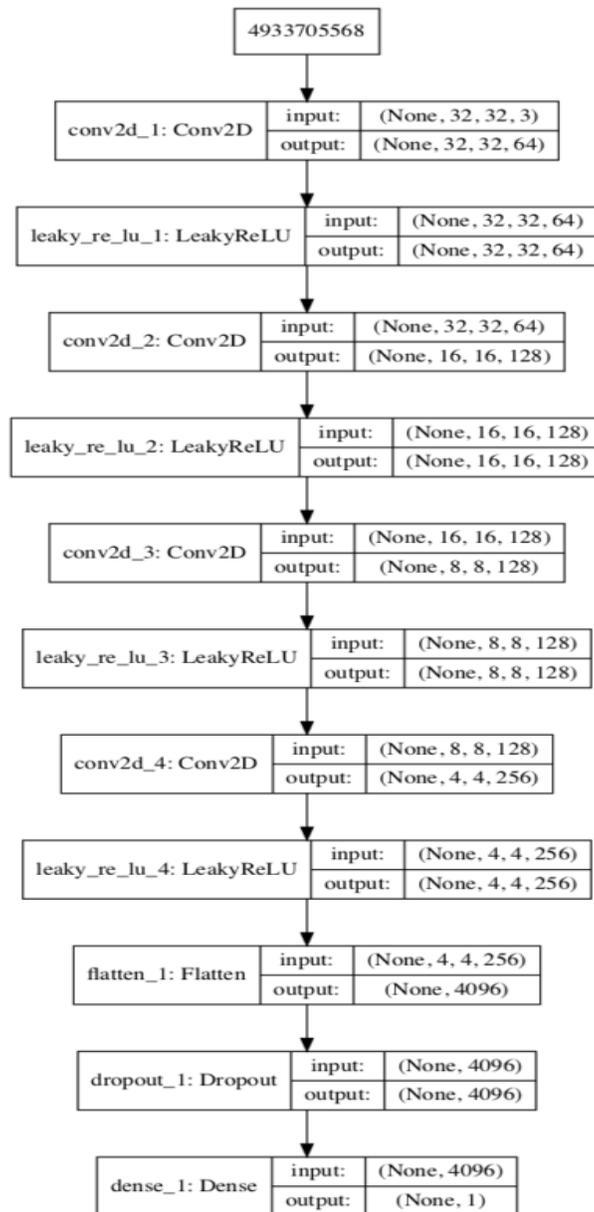


Figura 20.- Resumen del modelo discriminador de la red GAN CIFAR-10 (Fuente: [10]).

Este modelo se puede entrenar y con ello podemos observar que se trata de un modelo de clasificación binaria normal.

Primero se desarrolla una función para cargar y preparar las imágenes del dataset CIFAR-10. Estas imágenes escalarán del rango [0,255] al normalizado [-1,1] como se muestra en la función siguiente:

```

def load_real_samples():
    # load cifar10 dataset
    (trainX, _), (_, _) = load_data()
    # convert from unsigned ints to floats
    X = trainX.astype('float32')
  
```

```
# scale from [0,255] to [-1,1]
X = (X - 127.5) / 127.5
return X
```

Una vez se tiene el dataset preparado, se desarrolla otra función para obtener imágenes aleatorias de éste y una etiqueta con un 1 que indica que son imágenes reales. Este procedimiento se realiza como en el caso anterior por caso del gradiente descendente estocástico el cual lo requiere.

```
def generate_real_samples(dataset, n_samples):
    # choose random instances
    ix = randint(0, dataset.shape[0], n_samples)
    # retrieve selected images
    X = dataset[ix]
    # generate 'real' class labels (1)
    y = ones((n_samples, 1))
    return X, y
```

Se desarrolla ahora otra función para generar imágenes falsas. Al no tener el modelo del generador aún, se muestra una función que genera una imagen de 32x32 píxeles y 3 canales RGB con números aleatorios y los normaliza al rango de valores [-1,1], también se generan las etiquetas con el 0 para indicar que son imágenes falsas.

```
def generate_fake_samples(n_samples):
    # generate uniform random numbers in [0,1]
    X = rand(32 * 32 * 3 * n_samples)
    # update to have the range [-1, 1]
    X = -1 + X * 2
    # reshape into a batch of color images
    X = X.reshape((n_samples, 32, 32, 3))
    # generate 'fake' class labels (0)
    y = zeros((n_samples, 1))
    return X, y
```

La última función que se desarrolla es la función de entrenamiento del discriminador, esta se encargará de entrenar la red en un número de iteraciones pero sin pasar por todo el dataset, si no que pasará por un lote de imágenes definido en 128, donde 64 serán reales y otras 64 serán falsas. Se entrenará separadamente para imágenes reales y falsas para poder ver así la precisión de la red en cada caso.

```
def train_discriminator(model, dataset, n_iter=20, n_batch=128):
    half_batch = int(n_batch / 2)
    # manually enumerate epochs
    for i in range(n_iter):
        # get randomly selected 'real' samples
        X_real, y_real = generate_real_samples(dataset, half_batch)
        # update discriminator on real samples
        _, real_acc = model.train_on_batch(X_real, y_real)
```

```
# generate 'fake' examples
X_fake, y_fake = generate_fake_samples(half_batch)
# update discriminator on fake samples
_, fake_acc = model.train_on_batch(X_fake, y_fake)
# summarize performance
print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))
```

El resultado que obtenemos de ejecutar todo este código junto (visible en el Anexo 2) es el mostrado en la figura 21, donde se puede observar que el discriminador con apenas 20 iteraciones es capaz de distinguir perfectamente las imágenes reales de las falsas.

```
...
>16 real=100% fake=100%
>17 real=100% fake=100%
>18 real=98% fake=100%
>19 real=100% fake=100%
>20 real=100% fake=100%
```

Figura 21.- Resultado de precisión obtenida al entrenar la red discriminadora.

### 5.1.2.3 Generador del modelo.

Este modelo se encargará de la generación de imágenes falsas de los objetos del dataset CIFAR-10, esto se hace tomando un punto de un espacio latente como entrada y teniendo como salida una imagen a color del dataset CIFAR-10.

Con este generador convertiremos un vector de 100 dimensiones en una matriz 2D con  $32 \times 32 \times 3$  o 3072 valores con rango  $[-1,1]$ . Esto se hace mediante el enfoque de las redes adversarias generativas convolucionales profundas que han demostrado ser efectivas.

Primero se utiliza una capa *Dense* oculta que tenga los suficientes nodos para representar una imagen en baja resolución de la imagen de salida, en este caso se usa una imagen de  $4 \times 4$  píxeles y 3 canales de color, es decir, 48 nodos. Lo que se requiere de esta red es generar varias interpretaciones de la entrada, que es un patrón de las redes neuronales convolucionales en las cuales se tienen filtros paralelos que nos dan múltiples mapas de activación paralelos (mapas de características) con diferentes interpretaciones de la salida. Lo que se requiere en este caso es al contrario, requerimos diferentes versiones paralelas de la salida con diferentes características aprendidas que puedan comprimirse en la capa de salida en una imagen. Esto hace que la capa *Dense* de entrada tenga tantos nodos como versiones se quieren de la imagen de salida, en este caso 256 [10].

Estas versiones se remodelan en algo parecido a una imagen mediante la capa *Reshape* con 256 mapas de características de  $4 \times 4$ .

Una vez se tiene esta “imagen” en baja resolución se puede pasar ahora a ir aumentando la resolución mediante capas convolucionales (capas *Conv2DTranspose*) que serán configuradas con un paso de 2x2 que cuadruplicarán el área del mapa de características de entrada. Al igual que en el caso de MNIST se recomienda usar un tamaño de núcleo (*kernel*) que sea un factor del paso para evitar patrones de tablero ajedrez que pueden aparecer en el proceso de aumento de resolución.

Esto se repite dos veces para llegar a la imagen de 32x32 píxeles que se quiere como salida. Cada paso, como en el caso anterior, esta seguido de una capa *LeakyReLU* para disminuir el error cometido y para finalizar una capa convolucional (*Conv2D*) estándar con 3 filtros para los 3 canales de color, un núcleo de tamaño 3x3, la opción *‘same’ padding* (la cual nos dice que el mapa de características de la salida debe ser del mismo tamaño que el de la entrada, en este caso 32x32 pero con 3 canales) y una función de activación *tanh* (tangente hiperbólica) para asegurar que los valores se encuentran en el rango [-1,1].

Todo esto se muestra en la función `define_generator()` mostrada a continuación.

```
def define_generator(latent_dim):
    model = Sequential()
    # foundation for 4x4 image
    n_nodes = 256 * 4 * 4
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, 256)))
    # upsample to 8x8
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 16x16
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 32x32
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # output layer
    model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
    return model
```

Si se le añaden a este código la dimensión del espacio latente y se muestra, mediante la función `plot_model`, la estructura de la red, obtenemos lo mostrado en la figura 22.

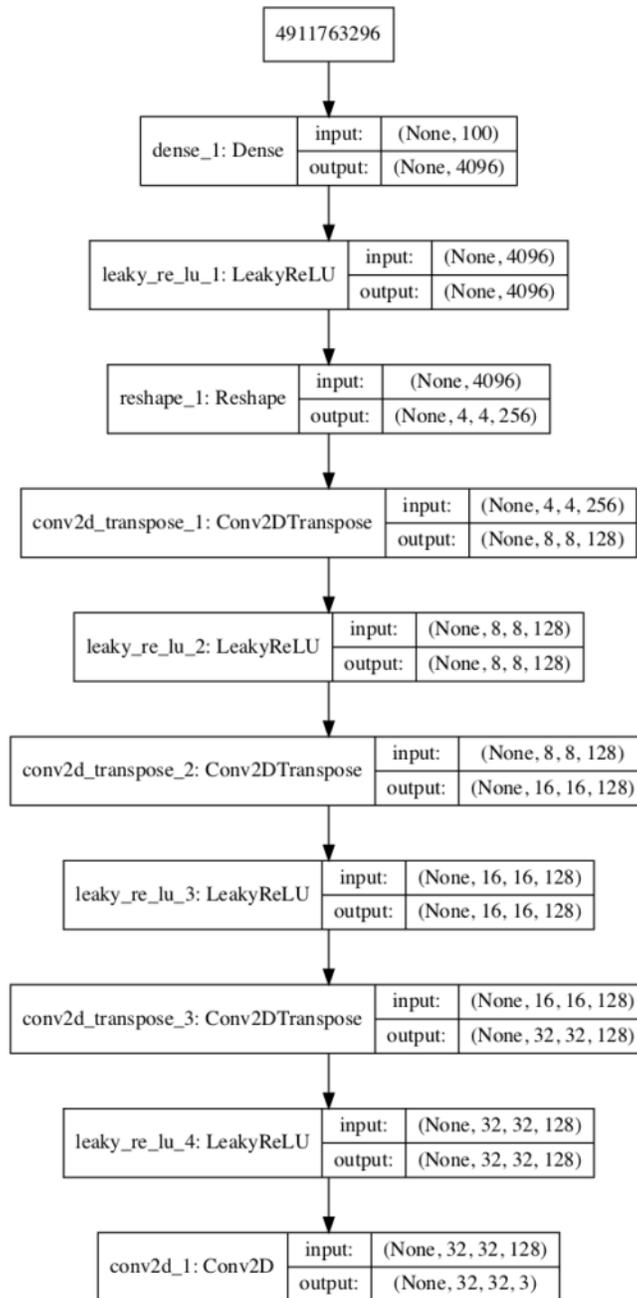


Figura 22.- Resumen del modelo generador de la red GAN CIFAR-10 (Fuente: [10]).

En esta figura se puede observar como desde un espacio latente de dimensión 100, se llega a la imagen de 32x32 píxeles a color a la salida de la red neuronal, pero este código por sí solo no puede hacer mucho, como ejemplo se procede a generar las imágenes sin entrenar el modelo.

Para ello, primero se generan puntos en el espacio latente, mediante la función *randn()* que genera matrices de números aleatorios con distribución Gaussiana. A dichos puntos se les da forma de matriz de forma que el número de muestras sean

las columnas y las filas sean 100 elementos aleatorios. Con la función `generate_latent_points()` realizamos dicho trabajo.

```
def generate_latent_points(latent_dim, n_samples):  
    # generate points in the latent space  
    x_input = randn(latent_dim * n_samples)  
    # reshape into a batch of inputs for the network  
    x_input = x_input.reshape(n_samples, latent_dim)  
    return x_input
```

Estos puntos servirán como entrada al modelo generador, con ellos podremos generar tantas imágenes como número de muestras queramos. Para ello se utiliza la función `generate_fake_samples()` mostrada a continuación.

```
def generate_fake_samples(g_model, latent_dim, n_samples):  
    # generate points in latent space  
    x_input = generate_latent_points(latent_dim, n_samples)  
    # predict outputs  
    X = g_model.predict(x_input)  
    # create 'fake' class labels (0)  
    y = zeros((n_samples, 1))  
    return X, y
```

Con un poco más de código para determinar la dimensión del espacio latente, el número de muestras y el uso de las funciones mencionadas, así como el escalado del rango de valores de los píxeles de la imagen generada de  $[-1,1]$  a  $[0,1]$ , podemos obtener imágenes generadas como muestra de que el generador funciona aunque, al no estar entrenado, esas imágenes sean aleatorias (como se muestra en la figura 23).

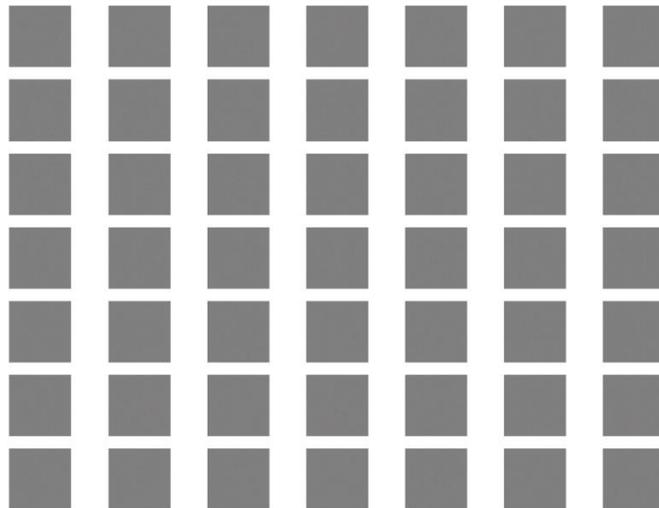


Figura 23.- Resultado obtenido de la ejecución del código del modelo del generador para la red GAN CIFAR-10 (Fuente: [10]).

El código completo de este apartado se puede consultar en el Anexo 3.

#### 5.1.2.4 Modelo GAN.

Una vez se tienen los modelos de discriminador y generador desarrollados, es hora de ver cómo entrenar el generador para obtener las imágenes falsas pero plausibles del dataset CIFAR-10. La base es la misma que en el caso del dataset MNIST, el discriminador marcará el entrenamiento del generador según sea su rendimiento.

El discriminador debe distinguir los ejemplos reales y los falsos, para que esto se realice de manera correcta, se desactivará el aprendizaje de la parte del discriminador del modelo GAN cuando se utilicen ejemplos falsos, evitando así el sobreentrenamiento en los ejemplos falsos [10].

Otro aspecto a considerar, es que se quiere que el discriminador piense que las imágenes falsas sean reales, por lo que se les pondrá la etiqueta de reales para ello.

En la función `define_gan()` se obtiene el modelo GAN basado en los modelos del generador y discriminador obtenidos anteriormente, en este caso, se realiza un modelo conjunto y se define el optimizador y una función de pérdida para el modelo.

```
def define_gan(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False
    # connect them
    model = Sequential()
    # add generator
    model.add(g_model)
    # add the discriminator
    model.add(d_model)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model
```

Si se le añaden a este código la dimensión del espacio latente y se muestra, mediante la función `plot_model`, la estructura de la red, obtenemos lo mostrado en la figura 24.

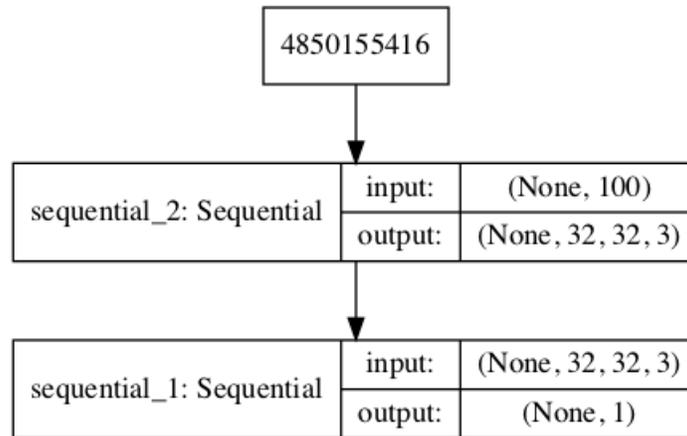


Figura 24.- Resumen del modelo de la red GAN CIFAR-10 (Fuente: [10]).

Lo que se requiere ahora es entrenar discriminador con imágenes reales y falsas y luego entrenar el generador mediante el modelo entero. Esto requiere elementos de la función de entrenamiento del discriminador y del entrenamiento de la red GAN completa.

En este caso lo que se hace es que en cada lote (*batch*) dentro de una iteración completa del dataset (*epoch*) se entrena el discriminador dos veces, una con elementos reales y otra con falsos, lo que se ha demostrado ser mejor a mezclar los elementos y entrenar una vez. Después se generan puntos aleatorios y se entrena la red GAN al completo, dejando el discriminador, como se ha mostrado antes, sin entrenar y entrenando así el generador.

Es importante monitorizar la pérdida del discriminador, debe estar entre 0,5 y 0.8 en cada lote, la del generador es menos importante y puede variar entre 0.5 y 2 o incluso mayor. Si la pérdida del discriminador cae demasiado eso indica que el generador no está funcionando bien y está generando imágenes fáciles de clasificar como falsas [10].

```

def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200,
n_batch=128):
    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    # manually enumerate epochs
    for i in range(n_epochs):
        # enumerate batches over the training set
        for j in range(bat_per_epo):
            # get randomly selected 'real' samples
            X_real, y_real = generate_real_samples(dataset, half_batch)
            # update discriminator model weights
            d_loss1, _ = d_model.train_on_batch(X_real, y_real)
            # generate 'fake' examples
            X_fake, y_fake = generate_fake_samples(g_model, latent_dim,
half_batch)
            # update discriminator model weights
            d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
  
```

```

# prepare points in latent space as input for the generator
X_gan = generate_latent_points(latent_dim, n_batch)
# create inverted labels for the fake samples
y_gan = ones((n_batch, 1))
# update the generator via the discriminator's error
g_loss = gan_model.train_on_batch(X_gan, y_gan)
# summarize loss on this batch
print('>d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
      (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))

```

Por último lo que se necesita es evaluar el modelo, viendo que las imágenes generadas son plausibles y para esto no hay maneras objetivas de hacerlo, por lo que será el factor humano el que determine la calidad de las imágenes generadas. Esto hace que no se puede parar el entrenamiento cuando las imágenes sean buenas, si no que se deberán ver ejemplos de estas imágenes cada cierta iteración y ver cómo van siendo los resultados. En estas iteraciones la imagen puede ir mejorando hasta que llegue a un punto donde no mejore más o incluso empiece a empeorar [10]. Para esto se realizan tres acciones periódicamente:

- Evaluar la precisión de clasificación del discriminador para elementos reales y falsos.
- Generar imágenes cada cierto tiempo y guardarlas en archivo para revisarlas.
- Guardar el modelo generador cada cierto tiempo.

Esto se realizará cada 10 iteraciones, para todo esto se define la función `summarize_performance()` la cual evalúa la precisión del discriminador con imágenes falsas y reales y lo imprime en pantalla, crea una representación de ejemplos de imágenes generadas y la guarda en archivo así como el modelo generador.

```

def save_plot(examples, epoch, n=7):
    # scale from [-1,1] to [0,1]
    examples = (examples + 1) / 2.0
    # plot images
    for i in range(n * n):
        # define subplot
        pyplot.subplot(n, n, 1 + i)
        # turn off axis
        pyplot.axis('off')
        # plot raw pixel data
        pyplot.imshow(examples[i])
    # save plot to file
    filename = 'generated_plot_e%03d.png' % (epoch+1)
    pyplot.savefig(filename)
    pyplot.close()

def summarize_performance(epoch, g_model, d_model, dataset, latent_dim,
n_samples=150):
    # prepare real samples
    X_real, y_real = generate_real_samples(dataset, n_samples)

```

```

# evaluate discriminator on real examples
_, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
# prepare fake examples
x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
# evaluate discriminator on fake examples
_, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
# summarize discriminator performance
print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
# save plot
save_plot(x_fake, epoch)
# save the generator model tile file
filename = 'generator_model_%03d.h5' % (epoch+1)
g_model.save(filename)

```

Añadiendo el código correspondiente para crear los modelos mediante las funciones, cargar el dataset y llamar a la función de entrenamiento tendríamos el ejemplo completo de la red GAN para imágenes del dataset CIFAR-10. Este código se puede consultar en el Anexo 4.

## 5.2 ADAPTACIÓN DE LAS REDES GAN DE EJEMPLO AL CASO DE ESTUDIO

### 5.2.1 Red GAN adaptada generadora de números escritos a mano MNIST.

#### 5.2.1.1 Preparación del dataset.

En este caso, la preparación del dataset se trata de obtener de archivo las imágenes del dataset proporcionado, con 200 imágenes por clase, de diferentes tipos de golosinas. Estas imágenes servirán como las imágenes reales para el entrenamiento del discriminador.

Lo primero será leer las imágenes, obteniendo la ruta y leyendo los directorios y las imágenes en éstos, tras esto, se construyen las etiquetas y se pasan los datos a una matriz *numpy*. Esto se pasa a la función *train\_test\_split()* la cual nos divide las matrices en subconjuntos de entrenamiento y test de manera aleatoria, estos subconjuntos se convierten en números reales (*float*) y se escalan los datos para pasar de un rango de valores de [0,255] a estar en un rango de [-1,1], apto para entrar a la red como se ha visto en el apartado 5.1.2.

```

dirname = os.path.join(os.getcwd(), 'Dataset_Golosinas/00_Beso')
imgpath = dirname + os.sep
images = []
directories = []
dircount = []
prevRoot=''
cant=0
print("leyendo imagenes de ",imgpath)
for root, dirnames, filenames in os.walk(imgpath):
    for filename in filenames:

```

```

    if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
        cant=cant+1
        filepath = os.path.join(root, filename)
        image = plt.imread(filepath)
        images.append(image)
        if prevRoot !=root:
            print(root, cant)
            prevRoot=root
            directories.append(root)
            dircount.append(cant)
            cant=0
dircount.append(cant)
dircount = dircount[1:]
dircount[0]=dircount[0]+1
print('Directorios leidos:',len(directories))
print("Imagenes en cada directorio", dircount)
print('suma Total de imagenes en subdirs:',sum(dircount))
# etiquetas
labels = []
indice = 0
for cantidad in dircount:
    for i in range(cantidad):
        labels.append(indice)
        indice = indice+1
golosinas = []
indice = 0
for directorio in directories:
    name = directorio.split(os.sep)
    golosinas.append(name[len(name)-1])
    indice = indice+1
y = np.array(labels)
X = np.array(images, dtype=np.uint8)
classes = np.unique(y)
nClasses = len(classes)
train_X, test_X, train_Y, test_Y = train_test_split(X,y, test_size=0.2)
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = (train_X-127.5)/127.5
test_X = (test_X-127.5)/127.5
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)

```

### 5.2.1.2 Discriminador adaptado del modelo.

En este caso, la adaptación del discriminador se basará en añadir dos capas convolucionales al discriminador ya desarrollado anteriormente, ya que las imágenes de golosinas sólo se diferencian de las imágenes del dataset MNIST en el tamaño y un los canales de color. Las imágenes del dataset MNIST son de un tamaño de 28x28 píxeles en escala de grises y las del dataset de golosinas de 64x64 píxeles en RGB, lo que hace que se requieran más capas convolucionales para ir disminuyendo la resolución hasta una resolución lo suficientemente baja como para poder tomar un vector y mediante la capa *Dense* final, obtener una probabilidad de que la imagen sea real o no.

```

def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                             input_shape=[64, 64, 3]))

    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(512, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model

```

Podemos ver la estructura de la red en la figura 25, donde se observa las diferentes capas y como se va obteniendo una “imagen” en menor resolución en cada capa convolutiva desde los 64x64 a la salida entre 0 y 1.

```

Model: "sequential_1"

```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 64)	4864
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 64)	0
dropout (Dropout)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 16, 16, 128)	204928
leaky_re_lu_5 (LeakyReLU)	(None, 16, 16, 128)	0
dropout_1 (Dropout)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 8, 8, 256)	819456
leaky_re_lu_6 (LeakyReLU)	(None, 8, 8, 256)	0
dropout_2 (Dropout)	(None, 8, 8, 256)	0
conv2d_3 (Conv2D)	(None, 4, 4, 512)	3277312
leaky_re_lu_7 (LeakyReLU)	(None, 4, 4, 512)	0
dropout_3 (Dropout)	(None, 4, 4, 512)	0
flatten (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 1)	8193

```

Total params: 4,314,753
Trainable params: 4,314,753
Non-trainable params: 0

```

**Figura 24.- Resumen del modelo discriminador adaptado de la red GAN MNIST.**

### 5.2.1.3 Generador adaptado del modelo.

En el generador se tendrá que realizar un cambio de estructura y de parámetros de las capas, ya que el generador MNIST está diseñado para generar imágenes de 28x28 píxeles en escala de grises (es decir, un canal de color), y en este caso no solo se requiere aumentar la resolución a 64x64 píxeles, si no que se requieren 3 canales de color, para ello, se requieren más capas y parámetros.

Como en el caso de los dígitos manuscritos MNIST, se debe partir de una “imagen” o matriz de baja resolución pero con suficientes valores como para luego ir aumentando la resolución hasta obtener una imagen de la resolución requerida. Para ello, y como en el caso de los datos MNIST, se hace uso de una capa *Dense* a la entrada con suficientes nodos como para formar una “imagen” en baja resolución, en este caso de 4x4 con 1024 valores. Esta “imagen” se pasa por cuatro capas convolucionales que van aumentando la resolución y disminuyendo el número de valores en la matriz. En la última capa convolucional se establece un tres como parámetro de número de filtros de la capa, por lo que tendremos una salida con 3 canales de color para la imagen, obteniendo finalmente la imagen 64x64 RGB que se requiere.

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(4*4*1024, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((4,4,1024)))
    assert model.output_shape == (None, 4,4,1024)
    model.add(layers.Conv2DTranspose(512, (5,5), strides=(2,2), padding='same',
use_bias=False))
    assert model.output_shape == (None, 8,8,512)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(256, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
    assert model.output_shape == (None, 16, 16, 256)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
    assert model.output_shape == (None, 32, 32, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(3, (5,5), strides=(2,2), padding='same',
use_bias=False, activation='sigmoid'))
    assert model.output_shape == (None, 64, 64, 3)
    return model
```

Se puede observar la evolución de los datos que pasan por el modelo generador en la figura 26.

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16384)	1638400
batch_normalization (Batch Normalization)	(None, 16384)	65536
leaky_re_lu (LeakyReLU)	(None, 16384)	0
reshape (Reshape)	(None, 4, 4, 1024)	0
conv2d_transpose (Conv2DTranspose)	(None, 8, 8, 512)	13107200
batch_normalization_1 (Batch Normalization)	(None, 8, 8, 512)	2048
leaky_re_lu_1 (LeakyReLU)	(None, 8, 8, 512)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 16, 16, 256)	3276800
batch_normalization_2 (Batch Normalization)	(None, 16, 16, 256)	1024
leaky_re_lu_2 (LeakyReLU)	(None, 16, 16, 256)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 32, 32, 128)	819200
batch_normalization_3 (Batch Normalization)	(None, 32, 32, 128)	512
leaky_re_lu_3 (LeakyReLU)	(None, 32, 32, 128)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 64, 64, 3)	9600
Total params: 18,920,320		
Trainable params: 18,885,760		
Non-trainable params: 34,560		

**Figura 25.- Resumen del modelo generador adaptado de la red GAN MNIST.**

El resto de funciones, como son la definición de las pérdidas de generador y discriminador, optimizadores, funciones de entrenamiento y guardado de modelo e imágenes son exactamente iguales a los del ejemplo con el dataset MNIST, al ejecutar el código completo podremos ir obteniendo los resultados del entrenamiento en archivo, se puede consultar el código completo en el Anexo 5.

## 5.2.2 Red GAN adaptada CIFAR-10 generadora de pequeñas fotografías.

### 5.2.2.1 Preparación del dataset.

En este caso, la preparación del dataset es prácticamente igual a la del ejemplo de los dígitos manuscritos MNIST, con un par de modificaciones para el correcto funcionamiento del ejemplo. Estas imágenes servirán como las imágenes reales para el entrenamiento del discriminador.

Lo primero será leer las imágenes como en el caso anterior, obteniendo la ruta y leyendo los directorios y las imágenes en éstos, tras esto, se construyen las etiquetas y se pasan los datos a una matriz *numpy*. Esto se pasa a la función *train\_test\_split()* la cual nos divide las matrices en subconjuntos de entrenamiento y test de manera aleatoria, estos subconjuntos se convierten en números reales (*float*) y se escalan los datos para pasar de un rango de valores de [0,255] a estar en un rango de [-1,1], apto para entrar a la red como se ha visto en el apartado 5.1.2.

```

def load_real_samples():
    dirname = os.path.join(os.getcwd(), 'Dataset_Golosinas')
    imgpath = dirname + os.sep
    images = []
    directories = []
    dircount = []
    prevRoot = ''
    cant = 0
    print("leyendo imagenes de ", imgpath)
    for root, dirnames, filenames in os.walk(imgpath):
        for filename in filenames:
            if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
                cant = cant + 1
                filepath = os.path.join(root, filename)
                image = plt.imread(filepath)
                images.append(image)
                if prevRoot != root:
                    print(root, cant)
                    prevRoot = root
                    directories.append(root)
                    dircount.append(cant)
                    cant = 0
        dircount.append(cant)
        dircount = dircount[1:]
        dircount[0] = dircount[0] + 1
    print('Directorios leidos:', len(directories))
    print("Imágenes en cada directorio", dircount)
    print('suma Total de imágenes en subdirs:', sum(dircount))
    # etiquetas
    labels = []
    indice = 0
    for cantidad in dircount:
        for i in range(cantidad):
            labels.append(indice)
            indice = indice + 1
    golosinas = []
    indice = 0
    for directorio in directories:
        name = directorio.split(os.sep)
        golosinas.append(name[len(name) - 1])
        indice = indice + 1
    y = np.array(labels)
    X = np.array(images, dtype=np.uint8)
    classes = np.unique(y)
    nClasses = len(classes)
    train_X, test_X, train_Y, test_Y = train_test_split(X, y, test_size=0.2)
    #convert from unsigned ints to floats
    test_X = test_X.astype('float32')
    X = train_X.astype('float32')
    # scale from [0,255] to [-1,1]
    X = (X - 127.5) / 127.5
    return X

```

### 5.2.2.2 Discriminador adaptado del modelo.

En este caso, la adaptación del discriminador es más sencilla, ya que las imágenes de golosinas sólo se diferencian de las imágenes del dataset CIFAR-10 en el tamaño, que es el doble. Las imágenes del dataset CIFAR-10 son de un tamaño de 32x32 píxeles en RGB y las del dataset de golosinas de 64x64 píxeles en RGB, lo que hace que simplemente habrá que adaptar el discriminador para aceptar imágenes más grandes y eso se traduce en añadir una capa más de convolución en el modelo, por lo que se cambiará la forma de los datos de entrada y se añadirá una capa más, quedando el código mostrado a continuación.

```
def define_discriminator(in_shape=(64,64,3)):
    model = Sequential()
    # normal
    model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(256, (3, 3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # classifier
    model.add(Flatten())
    model.add(Dropout(0.4))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=['accuracy'])
    return model
```

Se puede observar que se ha añadido una capa más al final de las capas convolutivas con su correspondiente capa *LeakyReLU* para la corrección de errores.

La función de generar los muestras reales se mantiene igual que en el caso del dataset CIFAR-10, obteniendo imágenes aleatorias del nuevo dataset y poniendo la etiqueta que indica que son imágenes reales.

### 5.2.2.3 Generador adaptado del modelo.

El caso del generador es bastante similar al del discriminador, el generador desarrollado para el dataset CIFAR-10 generaba imágenes de 32x32 píxeles RGB, en este caso, añadiendo una capa más de convolución para aumentar la resolución de esta imagen a el doble, obteniendo así la imagen de 64x64 píxeles RGB del dataset de las golosinas.

```
def define_generator(latent_dim):
    model = Sequential()
    # foundation for 4x4 image
    n_nodes = 256 * 4 * 4
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, 256)))
    # upsample to 8x8
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 16x16
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 32x32
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 64x64
    model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # output layer
    model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
    return model
```

En este caso se observa, como en el caso anterior, que se ha añadido dicha capa convolucional y su correspondiente capa *LeakyReLU*, por lo que vemos que la adaptación es bastante sencilla.

En cuanto a la función que genera las muestras falsas, no cambia con respecto al caso del dataset CIFAR-10, se generan unos puntos del espacio latente aleatorios, con la misma dimensión (100 en este caso) y se pasan al generador, tras esto se les coloca la etiqueta que indica que son datos falsos.

Con estos cambios, el modelo está preparado para ser entrenado, el resto de funciones, como son la combinación de generador y discriminador para realizar el modelo GAN, la función de entrenamiento, de evaluación del discriminador y el guardado del modelo generador son exactamente iguales a las del ejemplo del dataset CIFAR-10. Más tarde se estudiarán los diferentes resultados de aplicar este código al dataset de golosinas y diferentes variantes.

El código al completo del modelo GAN para el dataset de las golosinas, se puede consultar en el Anexo 6, donde se pueden ver las dependencias utilizadas así como el resto de funciones que no se han modificado conforme al ejemplo anterior.

### 5.2.3 Generador de imágenes para entrenamiento de la red clasificadora basado en la red GAN CIFAR-10.

Una vez se tiene el modelo GAN entrenado, se tienen archivos guardados de los modelos del generador para las diferentes iteraciones o variaciones del GAN que se hayan hecho. Esto es muy útil, ya que lo que se busca en el trabajo es la generación masiva de imágenes plausibles para el entrenamiento de la red clasificadora de golosinas, por lo que teniendo en archivo el modelo del generador que se haya visto que genere imágenes plausibles, se podrá cargar el modelo y usarlo como generador masivo de imágenes.

Esto requiere un código que cargue dicho modelo, genere puntos aleatorios de un espacio latente como en el caso del modelo GAN y guarde las imágenes generadas de forma individual en archivo para poder usarlas luego en el entrenamiento del clasificador. Este código se muestra a continuación.

```
# example of Loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot
from skimage import io

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)
    return x_input

# Load model
model = load_model('GAN_Dedo/generator_model_300.h5')
# generate images
latent_points = generate_latent_points(100, 1000)
# generate images
X = model.predict(latent_points)
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
# save images
for i in range(1000):
    io.imsave('GAN_Dedo/Dedo_Generado_%04d.jpeg' % (i+1), X[i,:,:,:])
```

Se puede observar que el código genera 1000 muestras de puntos en el espacio latente de dimensión 100, como en el caso del GAN completo, por lo que tendremos 1000 imágenes diferentes que generaremos cargando el modelo del generador y pasando dichos puntos por éste. Con estas imágenes, que se encuentran en un rango de valores de [-1,1], se escalan al rango de [0,1] y se guardan en archivo con el nombre requerido.

Para generar imágenes de cada clase, se requiere tener un modelo generador de cada clase y, por lo tanto, una red GAN entrenada para cada clase, por ello, no se utilizará la GAN genérica con todo el dataset, si no que se irá entrenando con cada clase y guardando los resultados en diferentes directorios, teniendo así modelos generadores para cada clase. Aunque se puede entrenar con todas las clases a la vez, se ha comprobado que se obtienen mejores resultados entrenando para cada clase.

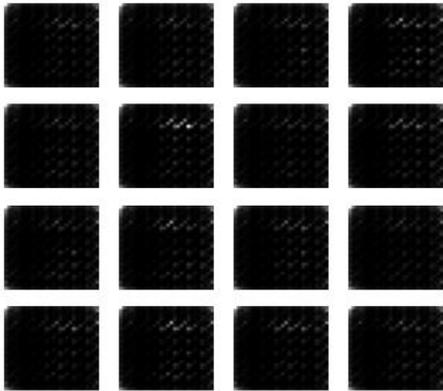
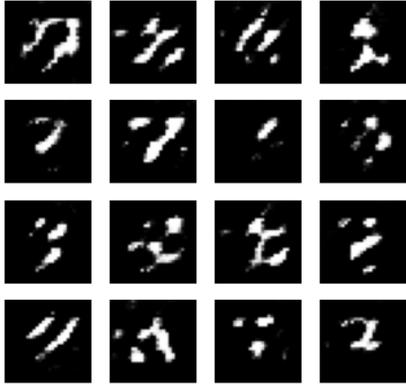
## 6. RESULTADOS

En este apartado se mostrarán los resultados obtenidos para cada software desarrollado en el apartado anterior, además de los resultados obtenidos de entrenar una red clasificadora desarrollada en otro trabajo con imágenes generadas por la red GAN que mejor resultados presente. La validez de los resultados de las redes se basarán en el factor humano, es decir, será el humano el que determine si las imágenes falsas pueden pasar por reales o no.

### 6.1 REDES GAN DE EJEMPLO DESARROLLADAS EN INTERNET

#### 6.1.1 Red GAN generadora de números escritos a mano MNIST.

En este caso, al ejecutar el código presente en el Anexo 1, obtenemos unas figuras que contienen 16 imágenes de cifras manuscritas, estas figuras son el resultado del generador entrenado en cada iteración. En este caso, el código realiza 50 iteraciones donde se entrena tanto el discriminador como el generador, ofreciéndonos 50 figuras, una por iteración, con las que podemos observar la evolución de las imágenes y como éstas se van pareciendo más a las imágenes originales. En la tabla 2, podemos observar esta evolución en diferentes etapas.

Iteración	Figura
1	
10	

20	
30	
40	
50	

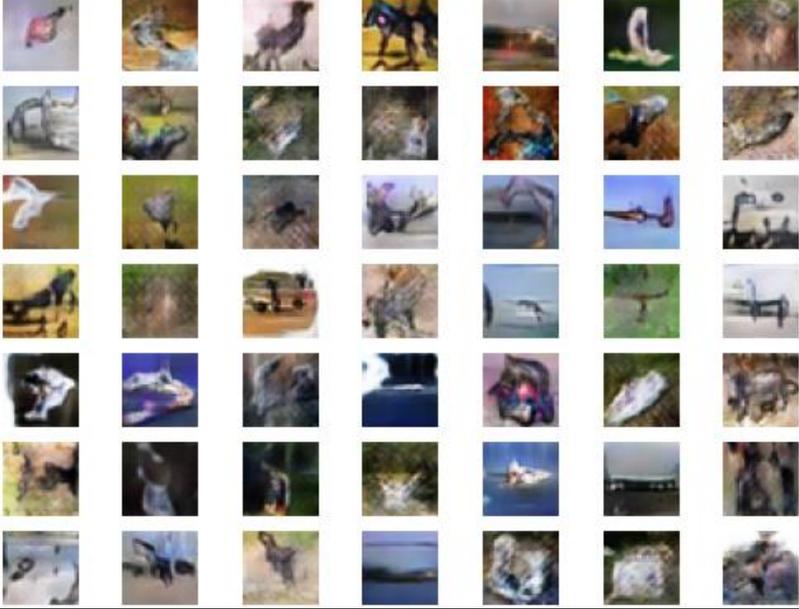
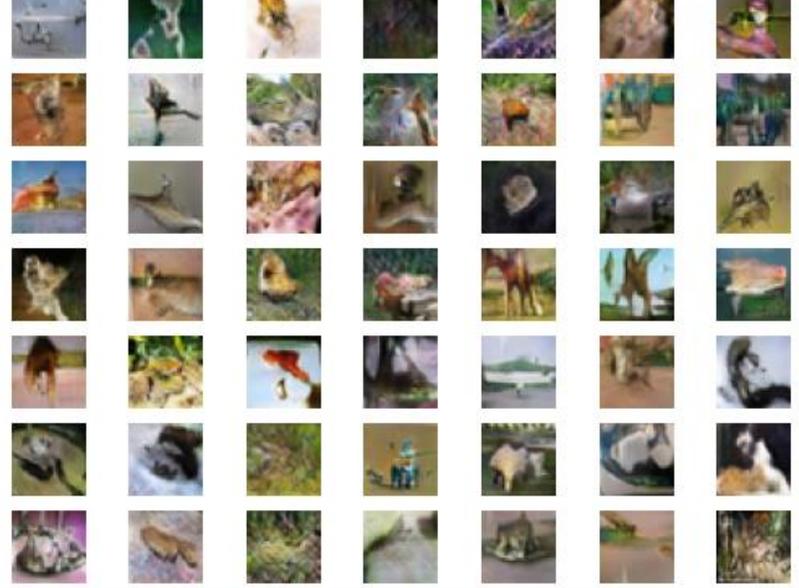
**Tabla 2.- Evolución de las imágenes generadas del dataset MNIST con respecto a la iteración del entrenamiento con la red GAN MNIST.**

El tiempo de entrenamiento obtenido es de unas 5.5 horas aproximadamente.

6.1.2 Red GAN CIFAR-10 generadora de pequeñas fotografías.

Al ejecutar el código de ejemplo (presente en el Anexo 4), obtenemos unas figuras que contienen 49 imágenes generadas relativas al dataset CIFAR-10, estas figuras son el resultado del generador entrenado en cada iteración. En este caso, el código realiza 200 iteraciones donde se entrena tanto el discriminador como el generador, ofreciéndonos 20 figuras, una por cada 10 iteraciones, con las que podemos observar la evolución de las imágenes y como éstas se van pareciendo más a las imágenes originales. En la tabla 3, podemos observar esta evolución en diferentes etapas.

Iteración	Figura							
10								
								
								
								
								
								
								

<p>50</p>	
<p>100</p>	

150	
200	

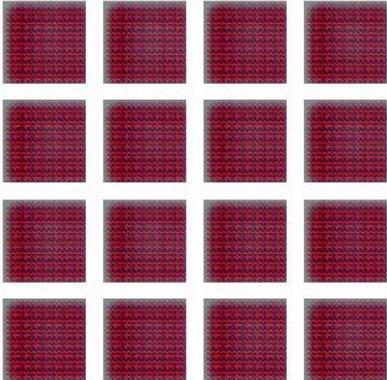
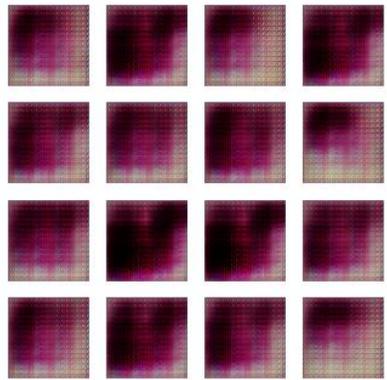
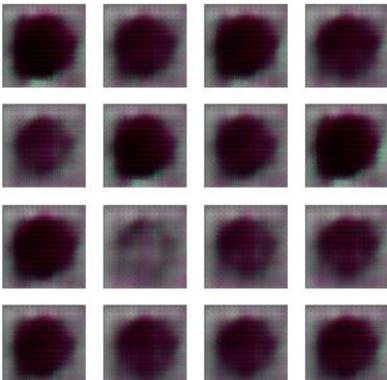
Tabla 3.- Evolución de las imágenes generadas del dataset CIFAR-10 con respecto a la iteración del entrenamiento con la red GAN CIFAR-10.

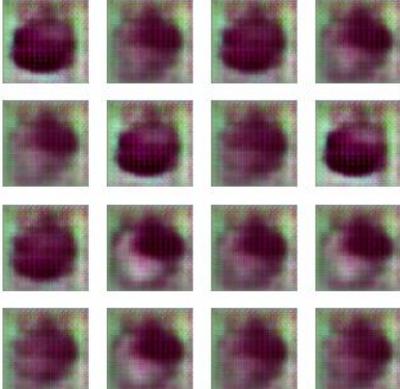
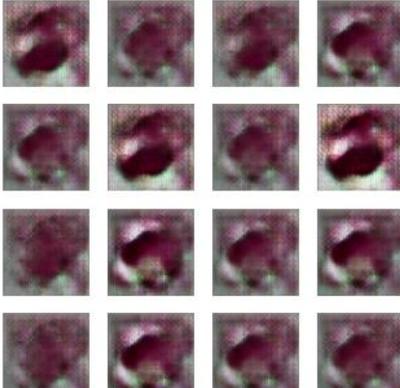
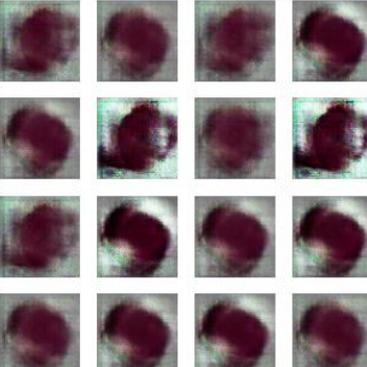
El tiempo de entrenamiento ha sido bastante largo, de aproximadamente 58.5 horas.

## 6.2 ADAPTACIÓN DE LAS REDES GAN DE EJEMPLO AL CASO DE ESTUDIO PARA LE GENERACIÓN DE IMÁGENES

### 6.2.1 Red GAN adaptada generadora de números escritos a mano MNIST.

En este caso, se hace una prueba de entrenamiento por clases, escogiendo la clase de imágenes de golosinas tipo “Beso” y entrenando la red con las imágenes de dicho tipo, observando los resultados después de 100 iteraciones, recogiendo esta evolución en la tabla 4, donde se pueden observar las figuras generadas con 16 imágenes generadas de la clase “Beso”.

Iteración	Figura
1	
5	
10	

25	
50	
75	
100	

**Tabla 4.- Evolución de las imágenes generadas del dataset golosinas de clase “Beso” con respecto a la iteración del entrenamiento con la red GAN MNIST adaptada.**

El tiempo de entrenamiento obtenido para una clase es de unas 25 horas.

### 6.2.2 Red GAN adaptada CIFAR-10 generadora de pequeñas fotografías.

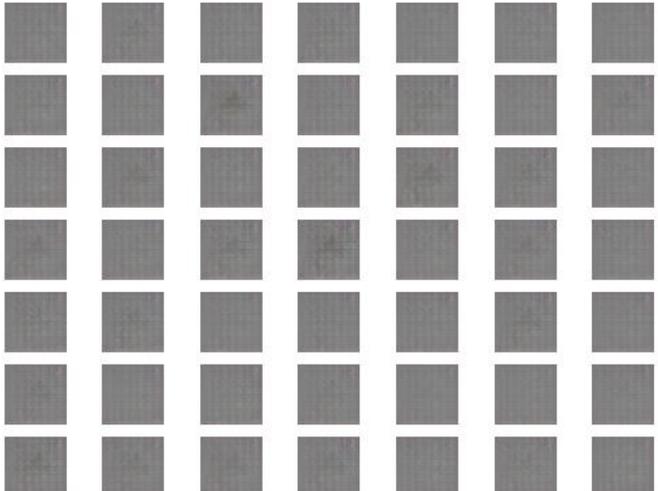
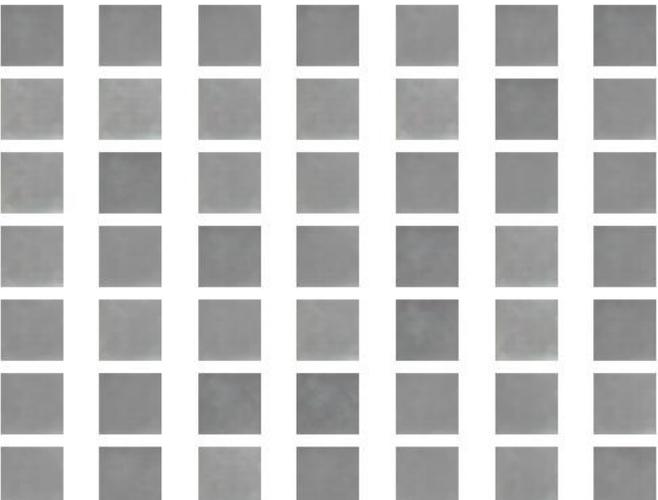
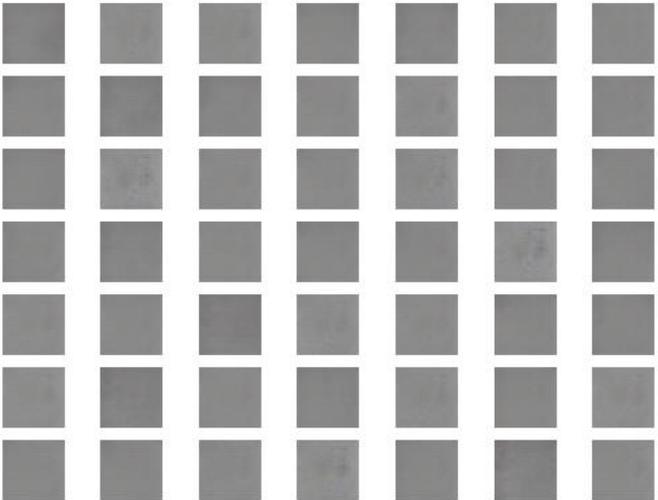
Este apartado, al igual que en el caso de la red GAN MNIST, se separará en dos, los resultados obtenidos de entrenar la red teniendo en cuenta todas las clases, y si se entrena clase a clase.

#### 6.2.2.1 Entrenamiento con todas las clases.

Si se entrena la red GAN CIFAR-10 adaptada a las imágenes de del dataset de golosinas teniendo en cuenta todas las clases, realizando 300 iteraciones obtenemos la evolución marcada en la tabla 5.

Iteraciones	Figura							
20								
								
								
								
								
								
								
								
60								
								
								
								
								
								
								
								

100	
140	
180	

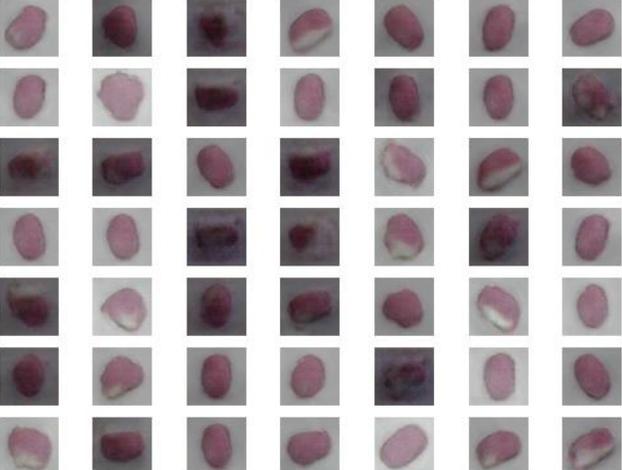
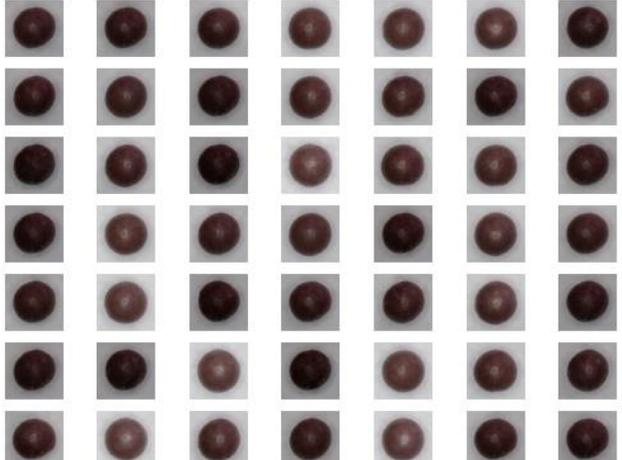
220	
260	
300	

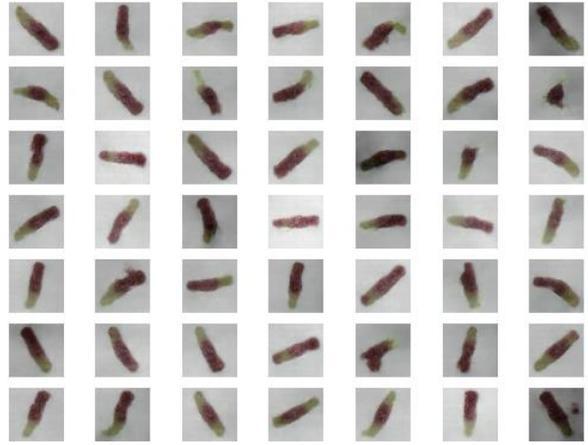
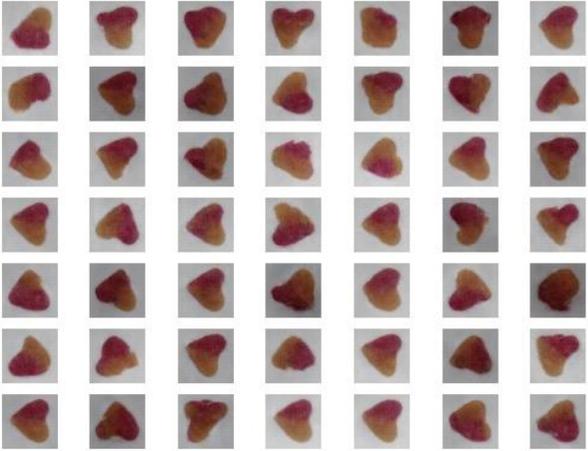
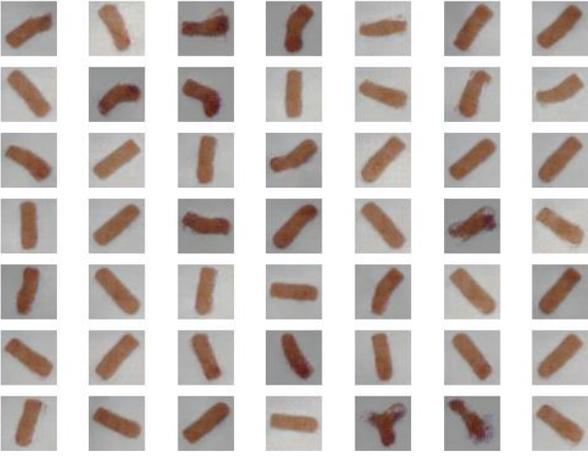
**Tabla 5.- Evolución de las imágenes generadas del dataset de golosinas con respecto a la iteración del entrenamiento con la red CIFAR-10 adaptada.**

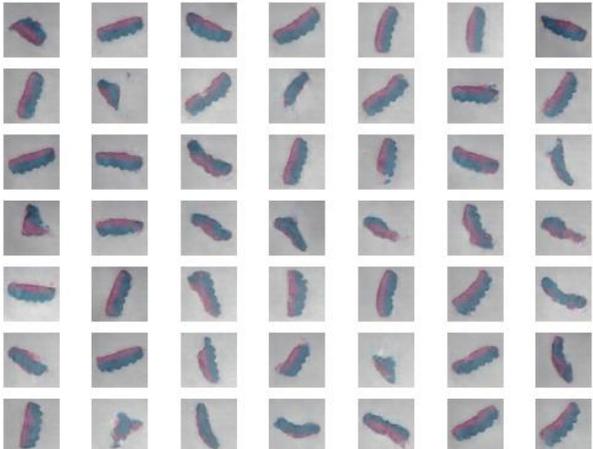
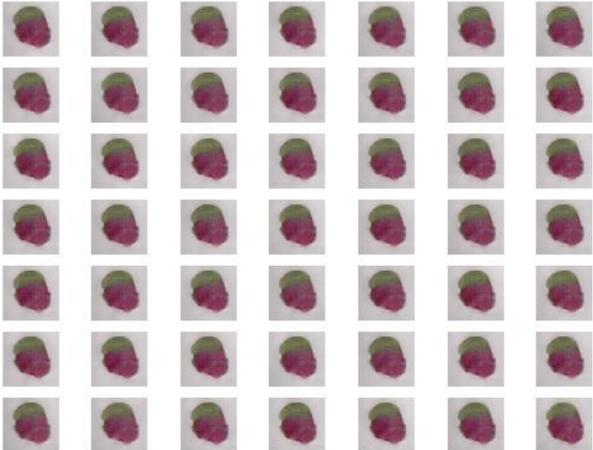
El tiempo de entrenamiento obtenido es de unas 16.25 horas.

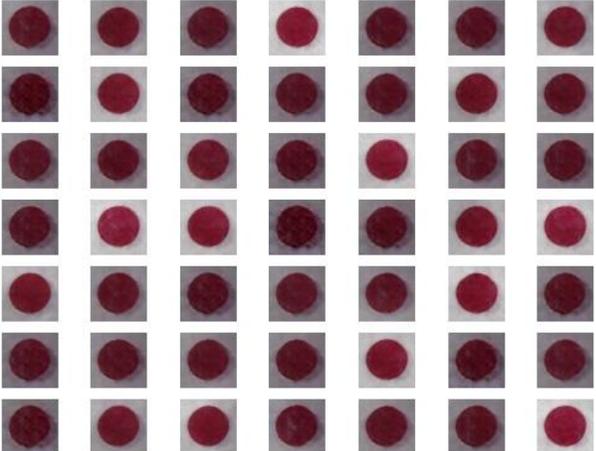
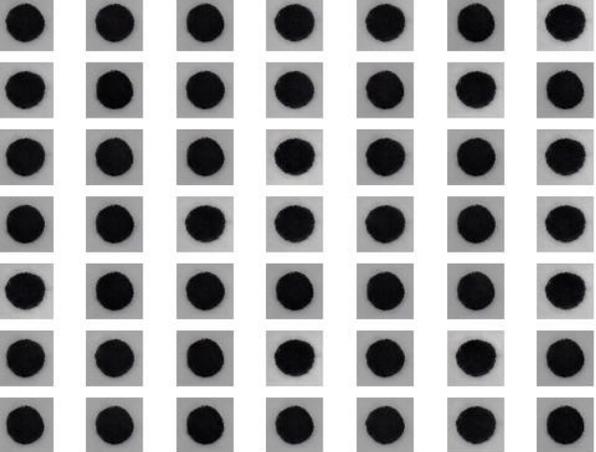
### 6.2.2.2 Entrenamiento por clases.

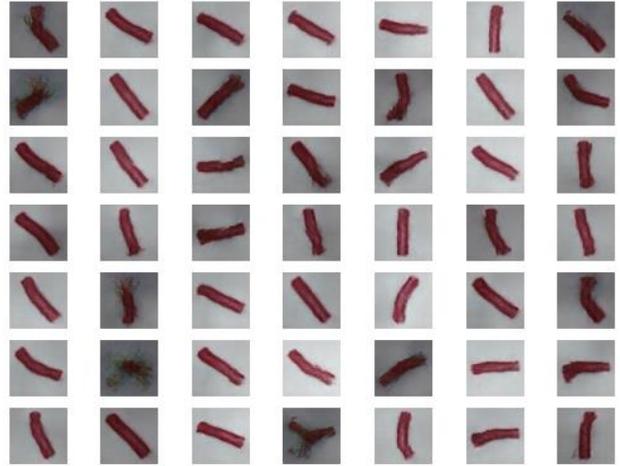
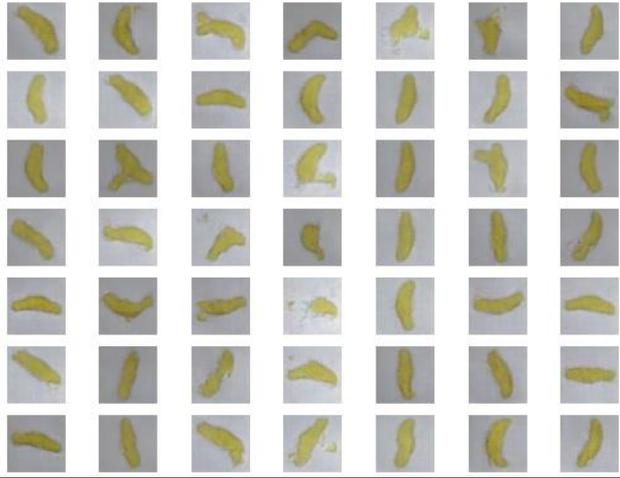
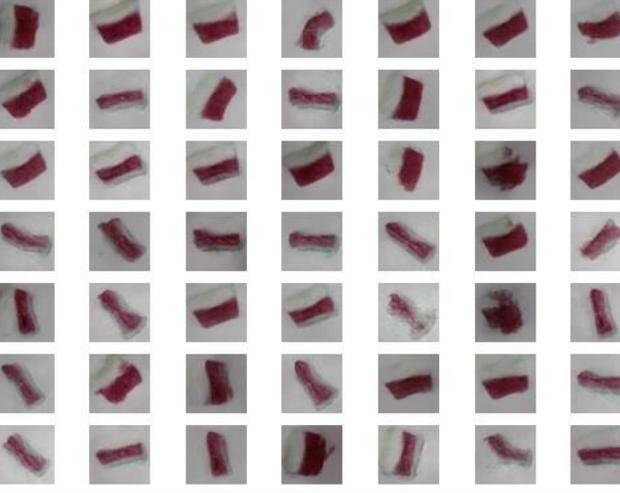
En vista de los resultados anteriores, es decir, entrenando la red teniendo en cuenta todas las clases y generar así todas las clases de una, se decide entrenar un modelo GAN para cada clase, generando así imágenes de cada clase y viendo si se mejoran los resultados. El resultado obtenido para cada clase, después de 300 iteraciones para cada clase es el mostrado en la tabla 6.

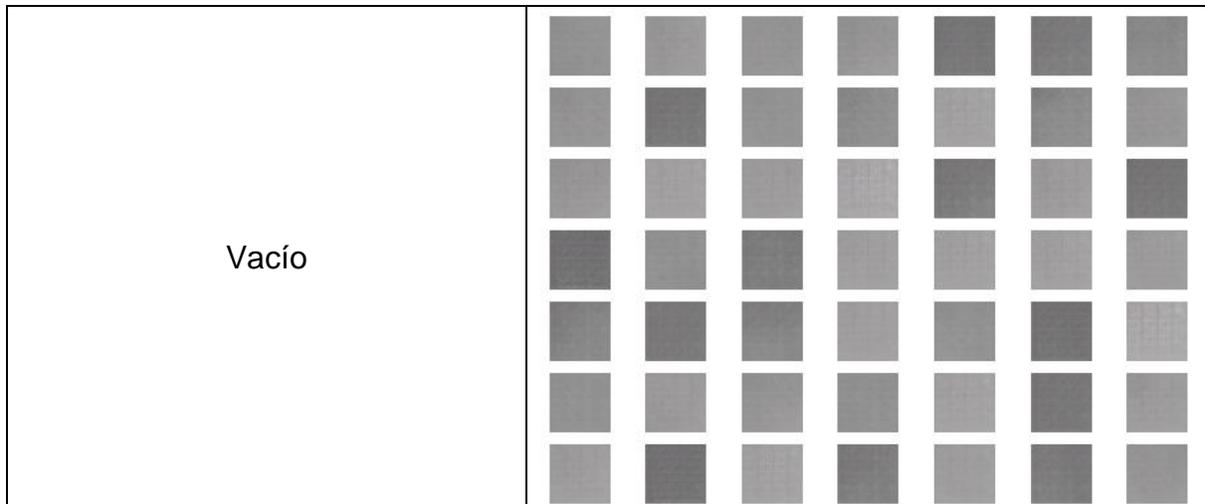
Clase	Figura
Beso	
Bola	

<p>Cola</p>	
<p>Corazón</p>	
<p>Dedo</p>	

<p>Dentadura</p>	
<p>Fresa</p>	
<p>Melón</p>	

<p>Moneda</p>	
<p>Mora</p>	
<p>Nube</p>	

<p>Palo</p>	
<p>Plátano</p>	
<p>Sandía</p>	



**Tabla 6.- Imágenes generadas obtenidas entrenando la red GAN CIFAR-10 adaptada para cada clase después de 300 iteraciones.**

El tiempo de entrenamiento obtenido es de unas 12.5 horas.

### **6.3 ENTRENAMIENTO DE LA RED CLASIFICADORA CON IMÁGENES GENERADAS**

En este apartado se mostrarán los resultados obtenidos de precisión, pérdida y tiempo de entrenamiento de la red clasificadora de imágenes de golosinas (presente en el Anexo 7) en diferentes casos de entrenamiento según el número de clases con las que se esté entrenando.

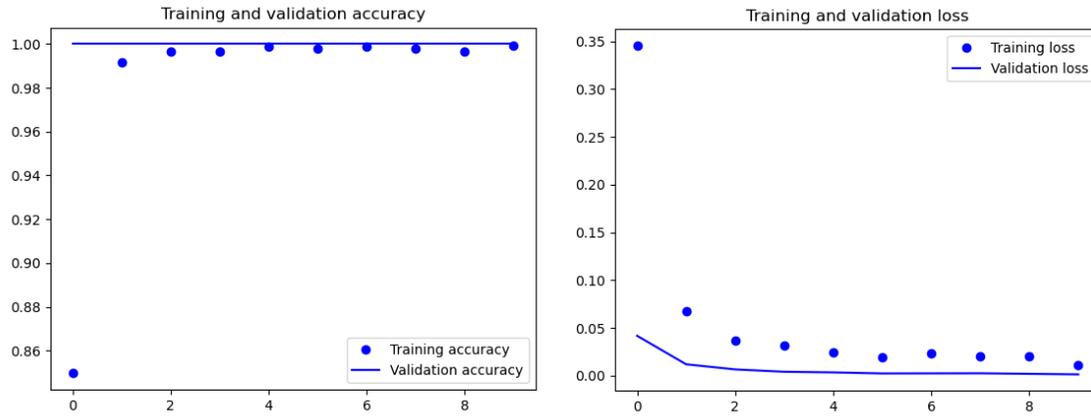
#### **6.3.1 Entrenamiento con 2 clases**

Resultados obtenidos utilizando las 2 clases con 10 iteraciones.

La figura 27 muestra las gráficas con la evolución con el número de iteraciones de las funciones de pérdida del conjunto de entrenamiento (loss) y del conjunto de validación (val\_loss) y de la función de precisión o acierto del conjunto de entrenamiento (acc) y del de validación (val\_acc).

Clases	Iteraciones	Imágenes reales por clase	Imágenes generadas por clase	Cantidad de imágenes procesadas a la vez
2	10	200	1000	64

**Tabla 7.- Ficha del experimento para 2 clases.**



**Figura 26.- Entrenamiento con 2 clases.**

La tabla 8 resume los resultados obtenidos para 2 clases.

Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
99.93 %	100 %	1.15 %	0.14 %	207

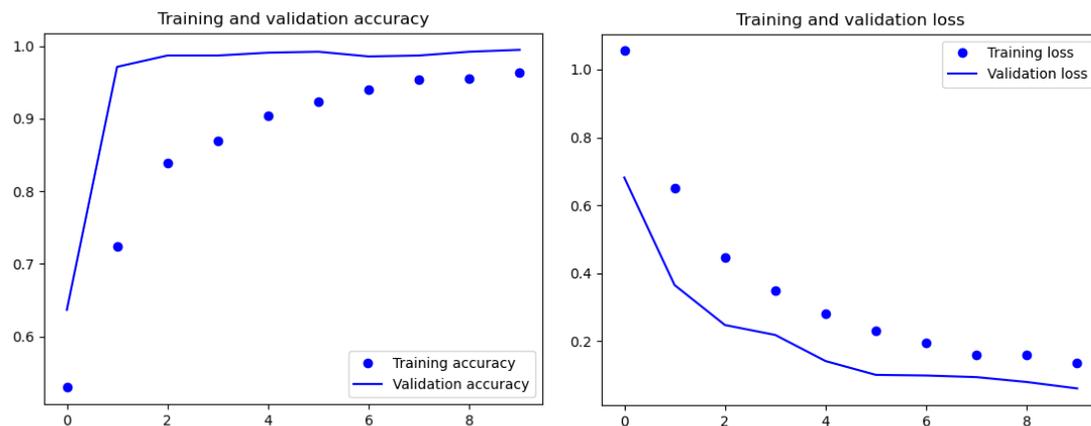
**Tabla 8.- Resultados del entrenamiento con 2 clases.**

### 6.3.2 Entrenamiento con 4 clases

La figura 28 muestra las gráficas con los resultados obtenidos utilizando las 4 clases con 10 iteraciones.

Clases	Iteraciones	Imágenes reales por clase	Imágenes generadas por clase	Cantidad de imágenes procesadas a la vez
4	10	200	1000	64

**Tabla 9.- Ficha del experimento para 4 clases.**



**Figura 27.- Entrenamiento con 4 clases.**

La tabla 10 resume los resultados obtenidos para 4 clases.

Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
96.39 %	99.48 %	13.41 %	6.01 %	416

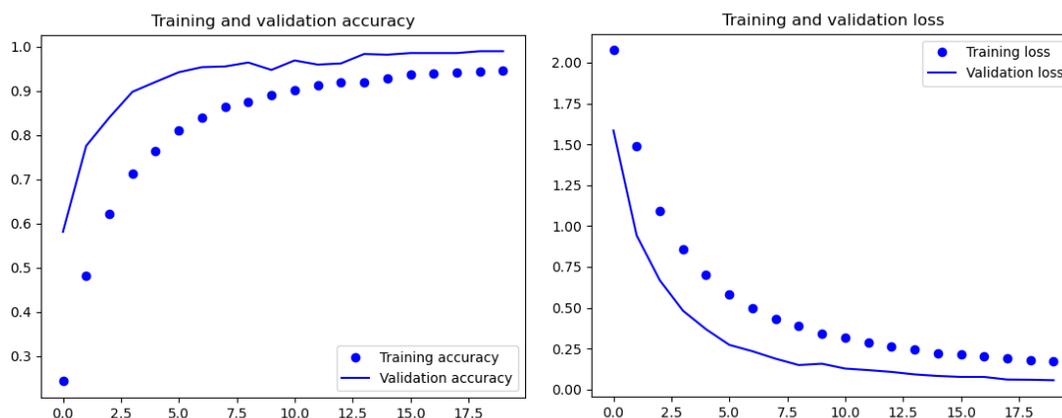
**Tabla 10.- Resultados del entrenamiento con 4 clases.**

### 6.3.3 Entrenamiento con 8 clases

La figura 29 muestra las gráficas con los resultados obtenidos utilizando las 8 clases con 15 iteraciones.

Clases	Iteraciones	Imágenes reales por clase	Imágenes generadas por clase	Cantidad de imágenes procesadas a la vez
8	15	200	1000	64

**Tabla 11.- Ficha del experimento para 8 clases.**



**Figura 28.- Entrenamiento con 8 clases.**

La tabla 12 resume los resultados obtenidos para 8 clases.

Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
93.12 %	98.49 %	22.15 %	8.91 %	1190

**Tabla 12.- Resultados del entrenamiento con 8 clases.**

### 6.3.4 Entrenamiento con 10 clases

La figura 30 muestra las gráficas con los resultados obtenidos utilizando las 10 clases con 20 iteraciones.

Clases	Iteraciones	Imágenes reales por clase	Imágenes generadas por clase	Cantidad de imágenes procesadas a la vez
10	20	200	1000	64

Tabla 13.- Ficha del experimento para 10 clases.

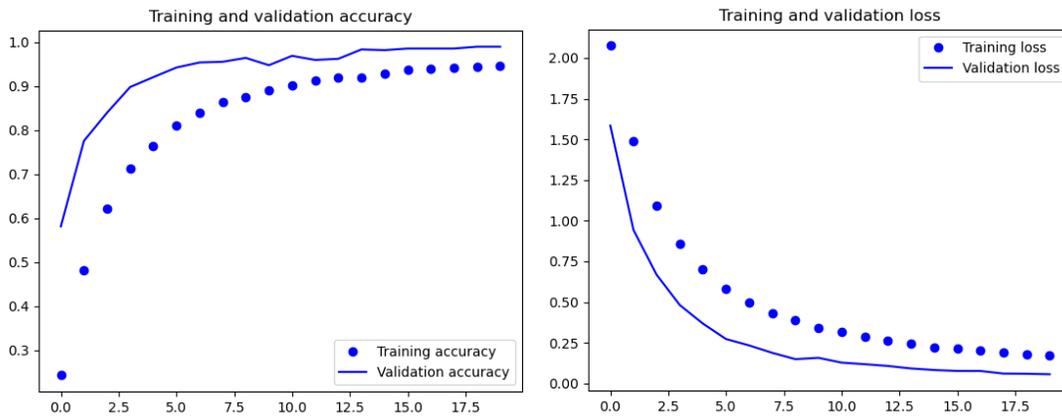


Figura 29.- Entrenamiento con 10 clases.

La tabla 14 resume los resultados obtenidos para 10 clases.

Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
93.10 %	98.49 %	22.44 %	9.96 %	1802

Tabla 14.- Resultados del entrenamiento con 10 clases.

### 6.3.5 Entrenamiento con 12 clases

La figura 31 muestra las gráficas con los resultados obtenidos utilizando las 12 clases con 40 iteraciones.

Clases	Iteraciones	Imágenes reales por clase	Imágenes generadas por clase	Cantidad de imágenes procesadas a la vez
12	40	200	1000	64

Tabla 15.- Ficha del experimento para 12 clases.

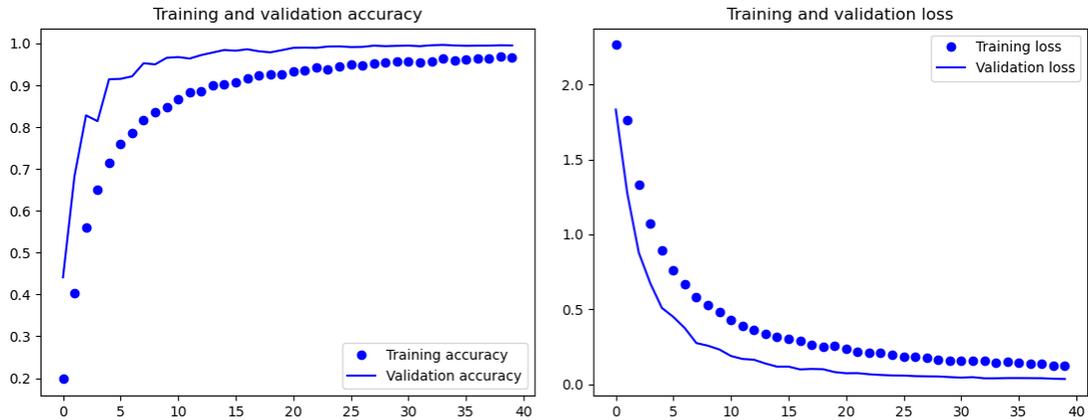


Figura 30.- Entrenamiento con 12 clases.

La tabla 16 resume los resultados obtenidos para 12 clases.

Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
97.16 %	99.35 %	11.36 %	3.61 %	4308

Tabla 16.- Resultados del entrenamiento con 12 clases.

### 6.3.6 Entrenamiento con 15 clases

La figura 32 muestra las gráficas con los resultados obtenidos utilizando las 15 clases del problema y con 40 iteraciones.

Clases	Iteraciones	Imágenes reales por clase	Imágenes generadas por clase	Cantidad de imágenes procesadas a la vez
15	40	200	1000	64

Tabla 17.- Ficha del experimento para 15 clases.

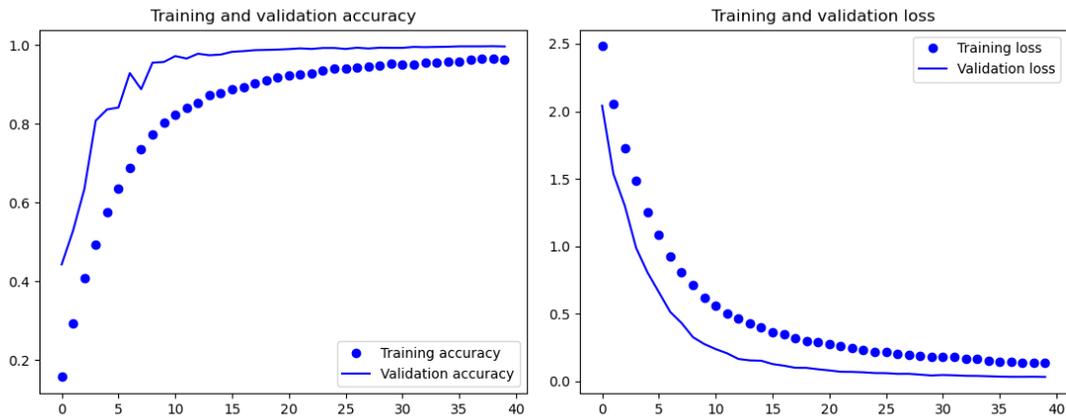


Figura 31.- Entrenamiento con 15 clases.

La tabla 18 resume los resultados obtenidos para todas las clases del problema.

Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
96.38 %	99.62 %	13.33 %	3.11 %	5521

**Tabla 18.- Resultados del entrenamiento con 15 clases.**

## 7. ANALISIS Y DISCUSION DE RESULTADOS

En este apartado se analizarán los resultados mostrados en el apartado anterior, observando si las imágenes generadas por las distintas redes son plausibles o no, que red es mejor para utilizarla de generadora de imágenes para el caso de estudio en base a la calidad de las imágenes y los tiempos de entrenamiento y una vez decidido esto, cómo es la mejora de la red clasificadora utilizando dichas imágenes generadas.

### 7.1 REDES GAN DE EJEMPLO DESARROLLADAS EN INTERNET

#### 7.1.1 Red GAN generadora de números escritos a mano MNIST.

En la tabla 2, se puede observar como las imágenes generadas van desde prácticamente un fondo negro con algunos patrones de blanco (patrones de tablero de ajedrez) en la primera iteración, hasta algo bastante parecido a los dígitos, aunque con defectos, en la iteración número 50. En dicha evolución, se puede observar como las primeras iteraciones estos patrones de blanco van buscando un patrón para intentar engañar al discriminador, hasta que, poco a poco, van formando formas que recuerdan a los dígitos manuscritos del dataset MNIST. Una vez se alcanza algo bastante parecido, las imágenes generadas se estabilizan, realizando pocos cambios en cada iteración, incluso llegando a un punto en el que incluso las imágenes comienzan a empeorar, esto queda reflejado si comparamos las imágenes generadas en la iteración 50 con las generadas en la iteración 47 (figura 33), donde se puede observar que algunos dígitos han cambiado quedando menos reales que anteriormente.

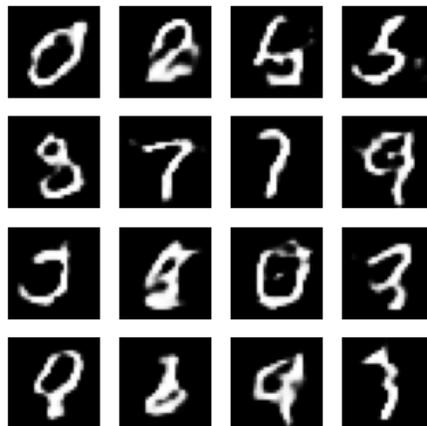


Figura 32.- Imágenes generadas en la iteración 47 del entrenamiento de la red GAN con dataset MNIST.

En general, con 50 iteraciones se puede observar que los resultados obtenidos son bastante buenos, las imágenes generadas son bastante parecidas a las del dataset MNIST y vemos que hay variedad de dígitos en la generación, sin que

haya problemas de “estancamientos” en el entrenamiento en patrones o puntos que provoquen que las imágenes generadas se parezcan poco a las reales.

El tiempo de entrenamiento no es muy lento comparado con otros casos, aproximadamente 5.5 horas, se verá posteriormente que en los demás casos este tiempo se multiplica llegando a ser extremadamente lento.

#### 7.1.1 Red GAN CIFAR-10 generadora de pequeñas fotografías.

Si se observa la evolución reflejada en la tabla 3, con 10 iteraciones ya se tiene algunos resultados, aunque en muy baja calidad, pudiendo diferenciar un fondo y un objeto en las imágenes, aunque se distinguen patrones de tablero de ajedrez y defectos que provocan que resulte muy obvio que la imagen es falsa o generada. Conforme se avanza en el entrenamiento, se puede observar que las imágenes van siendo más plausibles, con 100 iteraciones, se distinguen las imágenes muy parecidas a las del dataset CIFAR-10 pero sin distinguir muy claramente de que clases son. El modelo se mantiene prácticamente estable desde dicha iteración, mejorando muy poco y generando imágenes muy similares, pero que son bastante parecidas a las imágenes del dataset CIFAR-10. Esto hace pensar que, por la similitud del tipo de imágenes, de baja resolución pero con 3 canales de color como las del caso de estudio, se obtengan buenos resultados cuando se adapte la red GAN a las imágenes del caso del dataset de golosinas.

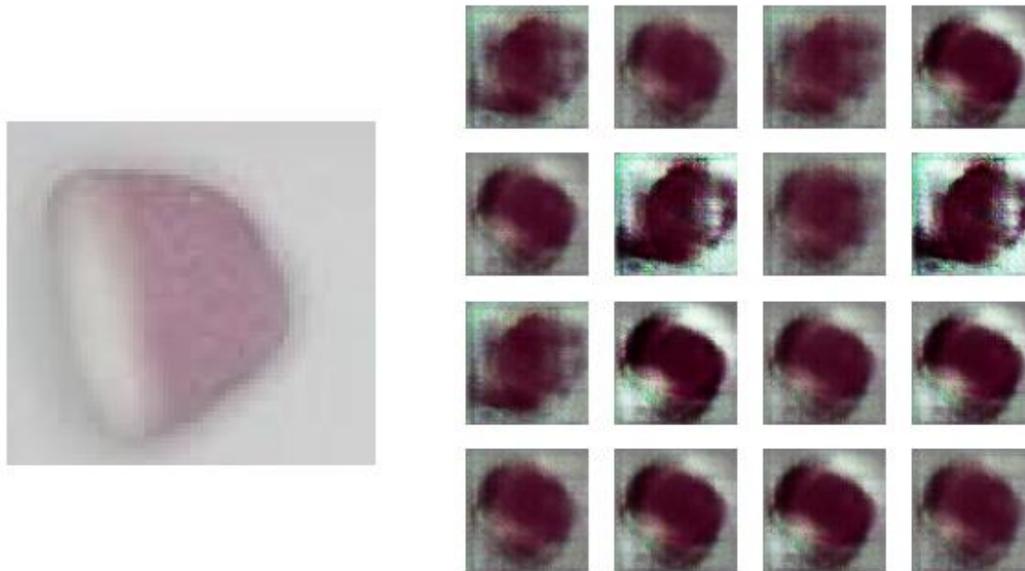
El entrenamiento es muy lento debido al gran número de imágenes y el proceso convolutivo (unas 58.5 horas), en el caso de estudio, el número de imágenes para entrenar es mucho menor, por lo que se espera que este tiempo se reduzca drásticamente.

## 7.2 ADAPTACIÓN DE LAS REDES GAN DE EJEMPLO AL CASO DE ESTUDIO PARA LA GENERACIÓN DE IMÁGENES

#### 7.2.1 Red GAN adaptada generadora de números escritos a mano MNIST.

En este caso, se prueba a entrenar la red para una sola clase, la clase “Beso” del dataset de golosinas. En la tabla 4, se puede observar la evolución de las imágenes generadas conforme avanza el entrenamiento.

Se puede observar que en las primeras iteraciones el algoritmo intenta generar diferentes fondos y patrones, generando imágenes claramente falsas, conforme avanza el entrenamiento y en relativamente pocas iteraciones, se puede observar que la red es capaz de generar un fondo y un objeto rojo en el centro acercándose poco a poco a la imagen real de la clase “Beso”, pero aún con muy mala calidad, esta imagen va mejorando con las iteraciones pero llega a un punto, sobre la iteración 20, en la que esta mejora se estabiliza, generando imágenes muy parecidas o incluso peores en las siguientes iteraciones hasta la iteración 100.



**Figura 33.- Comparación entre imagen real y generadas por la red GAN MNIST adaptada de la clase "Beso" del dataset golosinas.**

Como se puede apreciar en la figura 34, estos resultados nos muestran que esta red no es capaz de generar imágenes mínimamente plausibles del dataset de golosinas, ya que si no es capaz de generar imágenes para una sola clase, no lo será para el conjunto de todas.

Además, el proceso de entrenamiento en este caso es muy lento, unas 25 horas para una sola clase, lo que, extrapolando, serían 375 horas para las 15 clases. El tiempo de entrenamiento y los malos resultados para una clase, hacen que se descarte este caso para el generador de imágenes falsas que se utilizará para completar el dataset de golosinas en la red clasificadora.

## 7.2.2 Red GAN adaptada CIFAR-10 generadora de pequeñas fotografías.

### 7.2.2.1 Entrenamiento con todas las clases.

En la evolución presente en la tabla 5, se puede observar como en las primeras iteraciones, la red va aprendiendo a generar imágenes del dataset cada vez más plausibles, hasta llegar a imágenes bastante parecidas entorno a la iteración número 100. Se estabiliza en dicha iteración unas cuantas iteraciones más pero llega a un punto en el cual las imágenes empiezan a ser iguales entre sí, dejando el espacio vacío y dando una imagen de gris continuo. Esto puede ser debido a que, existiendo la clase "Vacío" dentro del dataset, la cual son imágenes del fondo establecido para todas las clases, un fondo gris, la red encuentre mayor facilidad en generar un fondo gris, sin patrones de figuras encima, y así engañar al discriminador con mayor facilidad, pero esto hace que no exista variedad en las imágenes generadas, generando solamente imágenes de una clase y no de todas las clases involucradas en el entrenamiento.

### 7.2.2.2 Entrenamiento por clases.

En la tabla 6, donde se pueden observar los resultados tras 300 iteraciones de la red GAN CIFAR-10 adaptada entrenada para cada clase, podemos observar que los resultados son muy buenos, las imágenes generadas son muy parecidas a las del dataset original, haciendo que en algunos casos sea imposible para el humano diferenciarlas de las imágenes reales. Es cierto que en algunos casos las imágenes generadas presentan anomalías, haciendo que sea muy obvio que no son reales, pero es un porcentaje pequeño y asumible. También se puede observar que en el caso de la clase “Fresa” la red se estanca en un tipo de imágenes, generando siempre el mismo tipo que, aunque plausible, carece de variedad.

En general las imágenes generadas son muy parecidas a las del dataset, con variedad y sin defectos, lo que hace que este método, es decir, la red GAN CIFAR-10 adaptada y entrenada para cada clase, sea la elegida para su uso posterior en la red generadora de imágenes, la cual generará imágenes falsas que se mezclarán con las reales del dataset para el entrenamiento de la red clasificadora y poder así comprobar si se produce una mejora de la precisión de la clasificación o no.

Debido al gran número de clases y al tener que entrenar la red GAN para cada una de ellas, el entrenamiento de la red para una sola clase tiene un tiempo de entrenamiento más rápido que para todas las clases juntas (unos 3000 segundos por clase), al tener que entrenar 15 redes GAN para cada clase (unas 12.5 horas), el entrenamiento es más rápido que entrenando la red para todas las clases (unas 16.25 horas). Se puede apreciar que estos datos confirman la reducción de tiempo en el entrenamiento de esta red en comparación con la red entrenada con imágenes del dataset CIFAR-10, aunque se hayan aumentado los procesos convolutivos, el número de imágenes es mucho menor, reduciendo considerablemente el tiempo de entrenamiento.

## 7.3 ENTRENAMIENTO DE LA RED CLASIFICADORA CON IMÁGENES GENERADAS

### 7.3.1 Entrenamiento con diferente número de clases.

En los diferentes experimentos con los diferentes números de clases, se puede observar que el resultado obtenido es bueno, la precisión de la red en la clasificación siempre está por encima del 98%.

Las curvas precisión con respecto a las iteraciones muestran que la tendencia es buena y en todos los experimentos se llega al punto de estancamiento, donde mayor número de iteraciones no suponen una mejora reseñable de la precisión de la red.

El único resultado que empeora es el tiempo de entrenamiento, que al tener un mayor número de imágenes, aumenta de manera considerable conforme aumentan las clases tenidas en cuenta, esta tendencia se puede observar en la figura 35.



Figura 34.- Gráfica de tendencia del tiempo de entrenamiento conforme a las clases tenidas en cuenta.

Se puede observar que el tiempo aumenta muy considerablemente según las clases que se involucren en el entrenamiento, en el caso de la red entrenada sin imágenes generadas, el tiempo aumenta pero la tendencia tiene una pendiente pequeña. Cuando se entrena la red con las imágenes generadas, podemos observar que la pendiente es mucho mayor, ya que se introducen 1000 imágenes más por clase, disparando el tiempo de entrenamiento.

### 7.3.2 Comparativa de resultados.

En este apartado se comparan y analizan los resultados de entrenar la red clasificadora de imágenes de golosinas del trabajo base con y sin imágenes generadas.

Casos	Entrenamiento	Training accuracy	Validation accuracy	Training loss	Validation loss	Time (seconds)
2 clases	Sin imágenes GAN	99.61 %	100 %	2.05 %	0.025 %	22
	Con imágenes GAN	99.93 %	100 %	1.15 %	0.14 %	207
4 clases	Sin imágenes GAN	97.46 %	98.44 %	5.21 %	1.71 %	41
	Con imágenes GAN	96.39 %	99.48 %	13.41 %	6.01 %	416

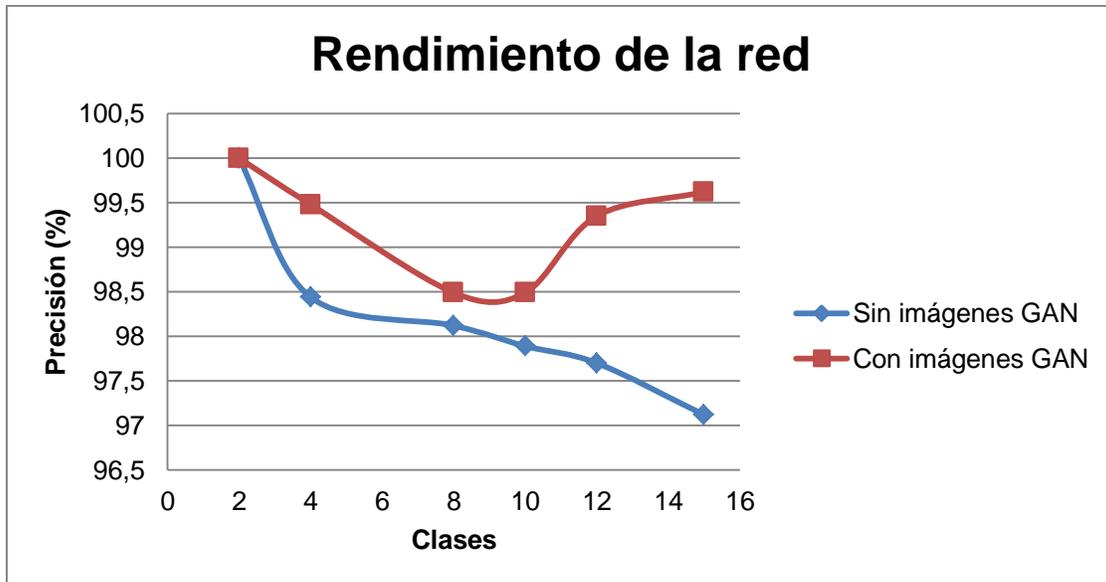
8 clases	Sin imágenes GAN	97.28 %	98.12 %	8.22 %	1.23 %	62
	Con imágenes GAN	93.12 %	98.49 %	22.15 %	8.91 %	1190
10 clases	Sin imágenes GAN	96.21 %	97.89 %	9.24 %	1.56 %	71
	Con imágenes GAN	93.10 %	98.49 %	22.44 %	9.96 %	1802
12 clases	Sin imágenes GAN	96.03 %	97.70 %	10.38 %	1.89 %	84
	Con imágenes GAN	97.16 %	99.35 %	11.36 %	3.61 %	4308
15 clases	Sin imágenes GAN	96.01 %	97.12 %	12.21 %	2.19 %	101
	Con imágenes GAN	96.38 %	99.62 %	13.33 %	3.11 %	5521

**Tabla 19.- Tabla comparativa de resultados de la red clasificadora con y sin imágenes generadas.**

En la tabla 19 se pueden comparar los resultados de entrenar la red clasificadora sin y con imágenes generadas por la red GAN, en cada caso, se han añadido 1000 imágenes generadas a las 200 reales de cada clase a clasificar. Como se puede observar en la tabla, la mejora en los resultados se observa en los datos de validación, siendo los datos de entrenamiento igual o incluso peor a los entrenamientos sin imágenes generadas.

El resultado es positivo, ya que se demuestra una mejora en general en la validación y test, que son las medidas que realmente nos dan una evaluación del rendimiento de la red neuronal, aunque el aumento de tiempo de entrenamiento es bastante considerable, ya que el volumen de imágenes a procesar es mucho mayor.

Podemos ver las comparativas de este rendimiento en la figura 36, donde se puede observar gráficamente la mejora del entrenamiento con imágenes generadas.



**Figura 35.- Comparativa gráfica de datos de validación de la red entrenada con y sin imágenes generadas.**

En esta gráfica se hace evidente la mejora del entrenamiento, como se puede ver, al tener más imágenes para entrenar la red, podemos hacer que al aumentar las clases no disminuya su rendimiento, si no que se obtiene una mejora de éste, llegando al 99.62 % de acierto incluso con las 15 clases, e incluso cambiando la tendencia descendente en ascendente.

Vemos que la curva de precisión de la red entrenada con imágenes generadas siempre está por encima de la obtenida por sin dichas imágenes, utilizando solo las imágenes reales, lo cual ya es un buen indicador de que al tener mayor número de imágenes, obtenemos un mejor rendimiento de la red clasificadora.

## 8. CONCLUSIONES

En este apartado se comentarán las conclusiones alcanzadas al realizar este trabajo, tanto en el desarrollo como en los resultados del mismo.

Lo primero que se ha podido probar en este trabajo es que existen diferentes arquitecturas de redes GAN en el mundo y que, mediante unas cuantas modificaciones, se puede obtener una red GAN adaptada a la aplicación que se requiera obteniendo resultados dispares. En este trabajo, con el estudio de dos redes GAN desarrolladas en internet y mediante las modificaciones realizadas se obtienen dos resultados bastante dispares, uno negativo y otro bastante positivo.

Con las adaptaciones incorporadas a una de las redes GAN, se obtienen imágenes plausibles para el problema expuesto de imágenes de golosinas, las cuales se utilizan para el estudio de mejora de la precisión de la red clasificadora de imágenes de golosinas desarrollada en el trabajo fin de grado del que se apoya este trabajo.

Los resultados del estudio de mejora de la precisión de dicha red clasificadora son positivos, se ha probado que el uso de un mayor número de imágenes (en este caso generadas) mejora la precisión de la red en general.

En cuanto a las conclusiones más específicas, se puede decir:

- Se han estudiado dos tipos de redes GAN con ejemplos de éxito en internet, comprobando sus resultados.
- Se han modificado satisfactoriamente estas redes GAN adaptándolas al problema propuesto, obteniendo imágenes que en uno de los casos han sido plausibles para el problema propuesto.
- Con dicha red GAN se ha generado una gran cantidad de imágenes falsas pero plausibles para el estudio de la mejora de la precisión de la red clasificadora.
- Se ha reentrenado la red clasificadora de imágenes de golosinas y se ha realizado un estudio comparativa con los resultados expuestos en el trabajo fin de grado donde se desarrolla dicha red.

Como se ha podido observar en este trabajo, la mejora de la red clasificadora no es muy notable, debido a que la red ya tenía una elevada precisión, pero en redes donde la precisión no sea tan alta o se tengan pocas imágenes para el entrenamiento, la incorporación de esta red GAN puede suponer una mejora considerable.

Se ha comprobado que el tiempo requerido para el entrenamiento de estas redes es muy dispar, por lo que en ciertas ocasiones puede resultar un ahorro de tiempo

y recursos y en otros puede que tome menos tiempo y recursos obtener más imágenes reales en vez de generar estas imágenes. Aunque la principal ventaja de estas redes GAN es que se pueden generar el número de imágenes que el usuario quiera y con bastante diversidad.

En este trabajo se han estudiado las redes GAN para la generación de imágenes, pero es posible utilizar estas redes para la generación de datos de cualquier tipo, abriendo un abanico de posibilidades enorme en cuanto a casuística y pudiéndose adaptar a casi cualquier problema como se ha visto.

## 9. RECOMENDACIONES Y TRABAJOS FUTUROS

Son observaciones de la parte práctica sobre aspectos por mejorar si alguien decide en un futuro realizar un avance, comparación, rectificación o rebatimiento al tema estudiado, así como una propuesta de los posibles trabajos futuros que complementarían o profundizarían este desarrollo.

En este apartado se exponen algunas mejoras y posibles trabajos que surgen a partir del desarrollo de este trabajo, teniendo en cuenta que esta ya es un trabajo surgido de otro trabajo fin de grado desarrollado anteriormente.

- Aunque el resultado obtenido del uso de estas redes desarrolladas es positivo, se podría estudiar el uso de otra arquitectura de red GAN estudiando la mejora en tiempos de entrenamiento y calidad de las imágenes generadas.
- El uso de estas redes GAN para la generación de otro tipo de datos, como pueden ser datos para modelado, ingeniería de control, robótica o reconocimiento de patrones por ejemplo.
- Estudiar la mejora de los resultados obtenidos en cuanto a calidad de imágenes y tiempo de entrenamiento con otros lenguajes de programación adecuados para redes neuronales como son R, Lisp, Java, etc.
- Estudio de las anomalías surgidas en el entrenamiento y resultados de las redes GAN como es el “estancamiento” de la calidad de las imágenes o el sobreentrenamiento de la red y posibles soluciones.

## 10. REFERENCIAS BIBLIOGRÁFICAS

- [1] Raj, B., 2019. *Todo Lo Que Necesitas Saber Sobre Las GAN: Redes Generativas Antagónicas – Puentes Digitales*. [online] Puentesdigitales.com. Available at: <<https://puentesdigitales.com/2019/04/05/todo-lo-que-necesitas-saber-sobre-las-gan-redes-generativas-antagonicas/>> [Accessed 15 March 2020].
- [2] Gujar, S., 2018. *Gans In Tensorflow (Part II)*. [online] Medium. Available at: <<https://medium.com/@sanketgujar95/gans-in-tensorflow-261649d4f18d>> [Accessed 11 April 2020].
- [3] Bagnato, J., 2018. *¿Cómo Funcionan Las Convolutional Neural Networks? Visión Por Ordenador*. [online] Aprende Machine Learning. Available at: <<https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/>> [Accessed 16 April 2020].
- [4] Bagnato, J., 2018. *Clasificación De Imágenes En Python*. [online] Aprende Machine Learning. Available at: <<https://www.aprendemachinelearning.com/clasificacion-de-imagenes-en-python/>> [Accessed 26 April 2020].
- [5] Martinez Heras, J., 2020. *Redes Neuronales Generativas Adversarias (Gans) - Iartificial.Net*. [online] IArtificial.net. Available at: <<https://www.iartificial.net/redes-neuronales-generativas-adversarias-gans/>> [Accessed 4 May 2020].
- [6] TensorFlow. 2020. *Deep Convolutional Generative Adversarial Network | Tensorflow Core*. [online] Available at: <<https://www.tensorflow.org/tutorials/generative/dcgan>> [Accessed 4 May 2020].
- [7] Torres, J., 2019. *Generative Adversarial Networks - Jordi TORRES.AI*. [online] Jordi TORRES.AI. Available at: <<https://torres.ai/generative-adversarial-networks/>> [Accessed 10 June 2020].
- [8] Ndoye, M., 2019. *Developing A DCGAN Model In Tensorflow 2.0*. [online] Medium. Available at: <<https://towardsdatascience.com/developing-a-dcgan-model-in-tensorflow-2-0-396bc1a101b2>> [Accessed 19 June 2020].
- [9] Pandey, P., 2018. *Deep Generative Models*. [online] Medium. Available at: <<https://towardsdatascience.com/deep-generative-models-25ab2821afd3>> [Accessed 23 June 2020].

- [10] Brownlee, J., 2019. *How To Develop A GAN To Generate CIFAR10 Small Color Photographs*. [online] Machine Learning Mastery. Available at: <<https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-cifar-10-small-object-photographs-from-scratch/>> [Accessed 25 June 2020].
- [11] Budhiraja, A., 2016. *Learning Less To Learn Better—Dropout In (Deep) Machine Learning*. [online] Medium. Available at: <<https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5>> [Accessed 30 June 2020].
- [12] Goodfellow, I., 2016. *NIPS 2016 Tutorial: Generative Adversarial Networks*.
- [13] ODENA, A., DUMOULIN, V. and OLAH, C., 2016. *Deconvolution And Checkerboard Artifacts*. [online] distill. Available at: <<https://distill.pub/2016/deconv-checkerboard/>> [Accessed 9 July 2020].
- [14] PASCUAL ESTAPÉ, J.A., 2019. *Inteligencia artificial: qué es, cómo funciona y para qué se está utilizando*. [online] ComputerHoy. Available at: <<https://computerhoy.com/reportajes/tecnologia/inteligencia-artificial-469917/>> [Accessed 22 October 2020].
- [15] VILLANUEVA GARCÍA, J.D., 2020. *Redes neuronales desde cero*. [online] IArtificial.net . Available at: <<https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion/>> [Accessed 22 October 2020].
- [15] *Conceptos básicos sobre redes neuronales*, s.f.. [online] Grupo.Us . Available at: <<http://grupo.us.es/gtocom/pid/pid10/RedesNeuronales.htm#modeloneurona/>> [Accessed 23 October 2020].
- [16] *Redes neuronales desde cero*. 2019. [online] biSmart . Available at: <<https://blog.bismart.com/es/diferencia-machine-learning-deep-learning>> [Accessed 23 October 2020].

## 11. ANEXOS

Los anexos expuestos serán todos códigos relativos a las diferentes redes y programas utilizados para el trabajo.

### 11.1 ANEXO 1. CÓDIGO RELATIVO A LA RED GAN MNIST COMPLETA (GAN\_MNIST.PY)

```
import tensorflow as tf
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display

(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28,
1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
BUFFER_SIZE = 60000
BATCH_SIZE = 256
# Batch and shuffle the data
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATC
H_SIZE)

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch
size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
```

```

use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model

generator = make_generator_model()
generator.summary()
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')

def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1]))

    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
discriminator = make_discriminator_model()
discriminator.summary()

# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                  discriminator_optimizer=discriminator_optimizer,
                                  generator=generator,
                                  discriminator=discriminator)

EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

```

```

# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate, noise_dim])

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

        gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))

def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix = checkpoint_prefix)

```

```

print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

# Generate after the final epoch
display.clear_output(wait=True)
generate_and_save_images(generator, epochs, seed)

train(train_dataset, EPOCHS)
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

```

## 11.2 ANEXO 2. CÓDIGO RELATIVO AL DISCRIMINADOR DE LA RED CIFAR-10 (CIFAR\_10\_DISCRIMINATOR.PY)

```

# example of training the discriminator model on real and random cifar10 images
from numpy import expand_dims
from numpy import ones
from numpy import zeros
from numpy.random import rand
from numpy.random import randint
from keras.datasets.cifar10 import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model

# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
    model = Sequential()
    # normal
    model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # classifier
    model.add(Flatten())
    model.add(Dropout(0.4))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=['accuracy'])
    return model

# Load and prepare cifar10 training images

```

```

def load_real_samples():
    # load cifar10 dataset
    (trainX, _), (_, _) = load_data()
    # convert from unsigned ints to floats
    X = trainX.astype('float32')
    # scale from [0,255] to [-1,1]
    X = (X - 127.5) / 127.5
    return X

# select real samples
def generate_real_samples(dataset, n_samples):
    # choose random instances
    ix = randint(0, dataset.shape[0], n_samples)
    # retrieve selected images
    X = dataset[ix]
    # generate 'real' class labels (1)
    y = ones((n_samples, 1))
    return X, y

# generate n fake samples with class labels
def generate_fake_samples(n_samples):
    # generate uniform random numbers in [0,1]
    X = rand(32 * 32 * 3 * n_samples)
    # update to have the range [-1, 1]
    X = -1 + X * 2
    # reshape into a batch of color images
    X = X.reshape((n_samples, 32, 32, 3))
    # generate 'fake' class labels (0)
    y = zeros((n_samples, 1))
    return X, y

# train the discriminator model
def train_discriminator(model, dataset, n_iter=20, n_batch=128):
    half_batch = int(n_batch / 2)
    # manually enumerate epochs
    for i in range(n_iter):
        # get randomly selected 'real' samples
        X_real, y_real = generate_real_samples(dataset, half_batch)
        # update discriminator on real samples
        _, real_acc = model.train_on_batch(X_real, y_real)
        # generate 'fake' examples
        X_fake, y_fake = generate_fake_samples(half_batch)
        # update discriminator on fake samples
        _, fake_acc = model.train_on_batch(X_fake, y_fake)
        # summarize performance
        print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# define model
model = define_discriminator()
# load image data
dataset = load_real_samples()
# fit the model
train_discriminator(model, dataset)

```

### 11.3 ANEXO 3. CÓDIGO RELATIVO AL GENERADOR DE LA RED CIFAR-10 (CIFAR\_10\_GENERATOR.PY)

```
# example of defining and using the generator model
from numpy import zeros
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from matplotlib import pyplot

# define the standalone generator model
def define_generator(latent_dim):
    model = Sequential()
    # foundation for 4x4 image
    n_nodes = 256 * 4 * 4
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, 256)))
    # upsample to 8x8
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 16x16
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 32x32
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # output layer
    model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
    return model

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)
    return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)
    # predict outputs
    X = g_model.predict(x_input)
    # create 'fake' class labels (0)
    y = zeros((n_samples, 1))
    return X, y
```

```

# size of the latent space
latent_dim = 100
# define the discriminator model
model = define_generator(latent_dim)
# generate samples
n_samples = 49
X, _ = generate_fake_samples(model, latent_dim, n_samples)
# scale pixel values from [-1,1] to [0,1]
X = (X + 1) / 2.0
# plot the generated samples
for i in range(n_samples):
    # define subplot
    pyplot.subplot(7, 7, 1 + i)
    # turn off axis labels
    pyplot.axis('off')
    # plot single image
    pyplot.imshow(X[i])
# show the figure
pyplot.show()

```

#### 11.4 ANEXO 4. CÓDIGO RELATIVO A LA RED GAN CIFAR-10 COMPLETA (CIFAR\_10\_GAN.PY)

```

# example of a dcgan on cifar10
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy.random import randn
from numpy.random import randint
from keras.datasets.cifar10 import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(32,32,3)):
    model = Sequential()
    # normal
    model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))

```

```

model.add(LeakyReLU(alpha=0.2))
# downsample
model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
# classifier
model.add(Flatten())
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=['accuracy'])
return model

# define the standalone generator model
def define_generator(latent_dim):
    model = Sequential()
    # foundation for 4x4 image
    n_nodes = 256 * 4 * 4
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, 256)))
    # upsample to 8x8
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 16x16
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 32x32
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # output layer
    model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
    return model

# define the combined generator and discriminator model, for updating the
generator
def define_gan(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False
    # connect them
    model = Sequential()
    # add generator
    model.add(g_model)
    # add the discriminator
    model.add(d_model)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model

# Load and prepare cifar10 training images
def load_real_samples():
    # load cifar10 dataset
    (trainX, _), (_, _) = load_data()

```

```

# convert from unsigned ints to floats
X = trainX.astype('float32')
# scale from [0,255] to [-1,1]
X = (X - 127.5) / 127.5
return X

# select real samples
def generate_real_samples(dataset, n_samples):
    # choose random instances
    ix = randint(0, dataset.shape[0], n_samples)
    # retrieve selected images
    X = dataset[ix]
    # generate 'real' class labels (1)
    y = ones((n_samples, 1))
    return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)
    return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)
    # predict outputs
    X = g_model.predict(x_input)
    # create 'fake' class labels (0)
    y = zeros((n_samples, 1))
    return X, y

# create and save a plot of generated images
def save_plot(examples, epoch, n=7):
    # scale from [-1,1] to [0,1]
    examples = (examples + 1) / 2.0
    # plot images
    for i in range(n * n):
        # define subplot
        pyplot.subplot(n, n, 1 + i)
        # turn off axis
        pyplot.axis('off')
        # plot raw pixel data
        pyplot.imshow(examples[i])
    # save plot to file
    filename = 'generated_plot_CIFAR_e%03d.png' % (epoch+1)
    pyplot.savefig(filename)
    pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim,
n_samples=150):
    # prepare real samples

```

```

X_real, y_real = generate_real_samples(dataset, n_samples)
# evaluate discriminator on real examples
_, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
# prepare fake examples
x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
# evaluate discriminator on fake examples
_, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
# summarize discriminator performance
print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
# save plot
save_plot(x_fake, epoch)
# save the generator model tile file
filename = 'generator_model_CIFAR_%03d.h5' % (epoch+1)
g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200,
n_batch=128):
    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    # manually enumerate epochs
    for i in range(n_epochs):
        # enumerate batches over the training set
        for j in range(bat_per_epo):
            # get randomly selected 'real' samples
            X_real, y_real = generate_real_samples(dataset, half_batch)
            # update discriminator model weights
            d_loss1, _ = d_model.train_on_batch(X_real, y_real)
            # generate 'fake' examples
            X_fake, y_fake = generate_fake_samples(g_model, latent_dim,
half_batch)
            # update discriminator model weights
            d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
            # prepare points in latent space as input for the generator
            X_gan = generate_latent_points(latent_dim, n_batch)
            # create inverted labels for the fake samples
            y_gan = ones((n_batch, 1))
            # update the generator via the discriminator's error
            g_loss = gan_model.train_on_batch(X_gan, y_gan)
            # summarize loss on this batch
            print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
                (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
            # evaluate the model performance, sometimes
            if (i+1) % 10 == 0:
                summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data

```

```

dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)

```

## 11.5 ANEXO 5. CÓDIGO RELATIVO A LA RED GAN MNIST ADAPTADA PARA EL CASO DE ESTUDIO DE IMÁGENES DE (GAN\_MNIST\_CHUCHES.PY)

```

import glob
import os
import re
import time

import PIL
import imageio
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from IPython import display
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers

# Preparación y carga de datasets
dirname = os.path.join(os.getcwd(), 'Dataset_Golosinas')
imgpath = dirname + os.sep
images = []
directories = []
dircount = []
prevRoot=''
cant=0
print("leyendo imagenes de ",imgpath)
for root, dirnames, filenames in os.walk(imgpath):
    for filename in filenames:
        if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
            cant=cant+1
            filepath = os.path.join(root, filename)
            image = plt.imread(filepath)
            images.append(image)
            if prevRoot !=root:
                print(root, cant)
                prevRoot=root
                directories.append(root)
                dircount.append(cant)
                cant=0
dircount.append(cant)
dircount = dircount[1:]
dircount[0]=dircount[0]+1
print('Directorios leidos:',len(directories))
print("Imagenes en cada directorio", dircount)
print('suma Total de imagenes en subdirs:',sum(dircount))
# etiquetas
labels = []
indice = 0
for cantidad in dircount:

```

```

    for i in range(cantidad):
        labels.append(indice)
        indice = indice+1
golosinas = []
indice = 0
for directorio in directories:
    name = directorio.split(os.sep)
    golosinas.append(name[len(name)-1])
    indice = indice+1
y = np.array(labels)
X = np.array(images, dtype=np.uint8)
classes = np.unique(y)
nClasses = len(classes)
train_X, test_X, train_Y, test_Y = train_test_split(X,y, test_size=0.2)
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = (train_X-127.5)/127.5
test_X = (test_X-127.5)/127.5
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)

train_X, valid_X, train_label, valid_label = train_test_split(train_X,
train_Y_one_hot, test_size=0.2, random_state=13)
BUFFER_SIZE = 60000
BATCH_SIZE = 64

# Creación de Los modelos (ambos con KERAS secuencial)

# Generador
# Usa capas tf.keras.layers.Conv2DTranspose para producir una imagen desde una
semilla. Empiza con una Dense para tomar la semilla
# como entrada hasta obtener una imagen de 64x64x3. La activación es LeakyRelY
excepto la salida que es tanh
# Modelo DCGAN
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(4*4*1024, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((4,4,1024)))
    assert model.output_shape == (None, 4,4,1024)
    model.add(layers.Conv2DTranspose(512,(5,5),strides=(2,2), padding='same',
use_bias=False))
    assert model.output_shape == (None, 8,8,512)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(256,(5, 5), strides=(2, 2), padding='same',
use_bias=False))
    assert model.output_shape == (None, 16, 16, 256)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2),
padding='same', use_bias=False))
    assert model.output_shape == (None, 32, 32, 128)
    model.add(layers.BatchNormalization())

```

```

    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(3, (5,5), strides=(2,2), padding='same',
use_bias= False, activation='sigmoid'))
    assert model.output_shape == (None, 64, 64, 3)
    return model

# Discriminador
# Clasificador de imágenes basado en redes convolucionales
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
input_shape=[64, 64, 3]))

    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(512, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model

generator = make_generator_model()
generator.summary()

discriminator = make_discriminator_model()
discriminator.summary()

# Definimos la función de coste y los optimizadores
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
# Función de coste del discriminador
# Compara las predicciones con imágenes reales con una matriz de 1s, y las
predicciones con imágenes generadas con una de 0s
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

# Función de coste del generador
# El discriminador debe clasificar las imágenes generadas como reales (1). Se
comparan las decisiones del discriminador
# con las imágenes generadas con una matriz de 1s
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

```

```

# Los optimizadores son diferentes ya que entrenaremos las 2 redes por separado
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
# Guardamos los modelos para poder restaurarlos posteriormente
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
discriminator_optimizer=discriminator_optimizer,
generator=generator,
discriminator=discriminator)
# Se define el entrenamiento
EPOCHS = 100
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.uniform(shape=[num_examples_to_generate, noise_dim], minval=0.,
maxval=1.)
# El entrenamiento empieza con el generador que recibe una semilla random como
entrada, ésta produce una imagen. El discriminador
# se usa para clasificar imágenes reales (del dataset) e imágenes generadas. El
coste se calcula para cada modelo y los gradientes se usan
# para entrenar el generador y discriminador.
@tf.function
def train_step(images):
    noise = tf.random.uniform(shape=[BATCH_SIZE, noise_dim], minval=0.,
maxval=1.)

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
        gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
        generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()
        for image_batch in dataset:
            image_batch = image_batch.reshape(1, 64, 64, 3)
            train_step(image_batch)
            display.clear_output(wait=True)
            generate_and_save_images(generator, epoch + 1, seed)
            # Guardar el modelo cada 15 iteraciones
            if (epoch+1)%10 == 0:
                checkpoint.save(file_prefix=checkpoint_prefix)
                print('Time for epoch {} is {} sec'.format(epoch+1, time.time()-start))
            display.clear_output(wait=True)
            generate_and_save_images(generator, epoch, seed)

```

```

# Función de generar y guardar imagenes
def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    fig = plt.figure(figsize=(4,4))
    for i in range(predictions.shape[0]):
        plt.subplot(4,4,i+1)
        plt.imshow(predictions[i, :, :, :])
        plt.axis('off')

plt.savefig('Imágenes_generadas/Completo/image_at_epoch_{:04d}.png'.format(epoch
))
    #plt.show()

# Entrenamiento del modelo
train(train_X, EPOCHS)

```

## 11.6 ANEXO 6. CÓDIGO RELATIVO A LA RED GAN CIFAR-10 ADAPTADA PARA EL CASO DE IMÁGENES DE (GAN\_CIFAR\_10\_CHUCHES.PY)

```

# dcgan on cifar10 for xuxes
import os
import re
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.cifar10 import load_data
from keras.optimizers import Adam
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from matplotlib import pyplot

# define the standalone discriminator model
def define_discriminator(in_shape=(64,64,3)):
    model = Sequential()
    # normal
    model.add(Conv2D(64, (3,3), padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # downsample
    model.add(Conv2D(128, (3,3), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))

```

```

# downsample
model.add(Conv2D(256, (3,3), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
# downsample
model.add(Conv2D(256, (3, 3), strides=(2, 2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
# classifier
model.add(Flatten())
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt,
metrics=['accuracy'])
return model

# define the standalone generator model
def define_generator(latent_dim):
    model = Sequential()
    # foundation for 4x4 image
    n_nodes = 256 * 4 * 4
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, 256)))
    # upsample to 8x8
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 16x16
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 32x32
    model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # upsample to 64x64
    model.add(Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    # output layer
    model.add(Conv2D(3, (3,3), activation='tanh', padding='same'))
    return model

# define the combined generator and discriminator model, for updating the
generator
def define_gan(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False
    # connect them
    model = Sequential()
    # add generator
    model.add(g_model)
    # add the discriminator
    model.add(d_model)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model

```

```

# Load and prepare xuxes training images
def load_real_samples():
    dirname = os.path.join(os.getcwd(), 'Dataset_golosinas')
    imgpath = dirname + os.sep
    images = []
    directories = []
    dircount = []
    prevRoot = ''
    cant = 0
    print("leyendo imagenes de ", imgpath)
    for root, dirnames, filenames in os.walk(imgpath):
        for filename in filenames:
            if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
                cant = cant + 1
                filepath = os.path.join(root, filename)
                image = plt.imread(filepath)
                images.append(image)
                if prevRoot != root:
                    print(root, cant)
                    prevRoot = root
                    directories.append(root)
                    dircount.append(cant)
                    cant = 0
    dircount.append(cant)
    dircount = dircount[1:]
    dircount[0] = dircount[0] + 1
    print('Directorios leidos:', len(directories))
    print("Imagenes en cada directorio", dircount)
    print('suma Total de imagenes en subdirs:', sum(dircount))
    # etiquetas
    labels = []
    indice = 0
    for cantidad in dircount:
        for i in range(cantidad):
            labels.append(indice)
            indice = indice + 1
    golosinas = []
    indice = 0
    for directorio in directories:
        name = directorio.split(os.sep)
        golosinas.append(name[len(name) - 1])
        indice = indice + 1
    y = np.array(labels)
    X = np.array(images, dtype=np.uint8)
    classes = np.unique(y)
    nClasses = len(classes)
    train_X, test_X, train_Y, test_Y = train_test_split(X, y, test_size=0.2)
    #convert from unsigned ints to floats
    test_X = test_X.astype('float32')
    X = train_X.astype('float32')
    # scale from [0,255] to [-1,1]
    X = (X - 127.5) / 127.5
    return X

```

```

# select real samples
def generate_real_samples(dataset, n_samples):
    # choose random instances
    ix = randint(0, dataset.shape[0], n_samples)
    # retrieve selected images
    X = dataset[ix]
    # generate 'real' class labels (1)
    y = ones((n_samples, 1))
    return X, y

# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)
    return x_input

# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)
    # predict outputs
    X = g_model.predict(x_input)
    # create 'fake' class labels (0)
    y = zeros((n_samples, 1))
    return X, y

# create and save a plot of generated images
def save_plot(examples, epoch, n=7):
    # scale from [-1,1] to [0,1]
    examples = (examples + 1) / 2.0
    # plot images
    for i in range(n * n):
        # define subplot
        pyplot.subplot(n, n, 1 + i)
        # turn off axis
        pyplot.axis('off')
        # plot raw pixel data
        pyplot.imshow(examples[i])
    # save plot to file
    filename = 'generated_plot_e%03d.png' % (epoch+1)
    pyplot.savefig(filename)
    pyplot.close()

# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim,
n_samples=150):
    # prepare real samples
    X_real, y_real = generate_real_samples(dataset, n_samples)
    # evaluate discriminator on real examples
    _, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
    # prepare fake examples
    x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
    # evaluate discriminator on fake examples

```

```

_, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
# summarize discriminator performance
print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))
# save plot
save_plot(x_fake, epoch)
# save the generator model tile file
filename = 'generator_model_%03d.h5' % (epoch+1)
g_model.save(filename)

# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=200,
n_batch=16):
    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    # manually enumerate epochs
    for i in range(n_epochs):
        # enumerate batches over the training set
        for j in range(bat_per_epo):
            # get randomly selected 'real' samples
            X_real, y_real = generate_real_samples(dataset, half_batch)
            # update discriminator model weights
            d_loss1, _ = d_model.train_on_batch(X_real, y_real)
            # generate 'fake' examples
            X_fake, y_fake = generate_fake_samples(g_model, latent_dim,
half_batch)
            # update discriminator model weights
            d_loss2, _ = d_model.train_on_batch(X_fake, y_fake)
            # prepare points in latent space as input for the generator
            X_gan = generate_latent_points(latent_dim, n_batch)
            # create inverted labels for the fake samples
            y_gan = ones((n_batch, 1))
            # update the generator via the discriminator's error
            g_loss = gan_model.train_on_batch(X_gan, y_gan)
            # summarize loss on this batch
            print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
                (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))
            # evaluate the model performance, sometimes
            if (i+1) % 20 == 0:
                summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)

```

## 11.7 ANEXO 7. CÓDIGO RELATIVO A LA RED CLASIFICADORA DE IMÁGENES DE (CLASIFICADOR\_CON\_GAN.PY)

```
import numpy as np
import os
import re
import getopt
import sys
import ast
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import keras
from keras.utils import to_categorical
from keras.models import Sequential, Input, Model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
from keras.layers.advanced_activations import LeakyReLU
from keras import backend as K
import cv2

# prevent Tensorflow memory Leakage
K.clear_session()

dirname = os.path.join(os.getcwd(), 'Dataset_Golosinas_GAN/clases')
imgpath = dirname + os.sep
images = []
directories = []
dircount = []
prevRoot=''
cant=0
print("leyendo imagenes de ",imgpath)
for root, dirnames, filenames in os.walk(imgpath):
    for filename in filenames:
        if re.search("\.(jpg|jpeg|png|bmp|tiff)$", filename):
            cant=cant+1
            filepath = os.path.join(root, filename)
            image = plt.imread(filepath)
            images.append(image)
            #X[cant-1, :, :, :] = image
            if prevRoot !=root:
                print(root, cant)
                prevRoot=root
                directories.append(root)
                dircount.append(cant)
                cant=0
    dircount.append(cant)
dircount = dircount[1:]
dircount[0]=dircount[0]+1
print('Directorios leidos:',len(directories))
print("Imágenes en cada directorio", dircount)
print('suma Total de imágenes en subdirs:',sum(dircount))
# etiquetas
labels = []
```

```

indice = 0
for cantidad in dircount:
    for i in range(cantidad):
        labels.append(indice)
        indice = indice+1
golosinas = []
indice = 0
for directorio in directories:
    name = directorio.split(os.sep)
    golosinas.append(name[len(name)-1])
    indice = indice+1
y = np.array(labels)
X = np.array(images, dtype=np.uint8)
classes = np.unique(y)
nClasses = len(classes)
train_X, test_X, train_Y, test_Y = train_test_split(X,y, test_size=0.2)
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X/255.
test_X = test_X/255.
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)

train_X, valid_X, train_label, valid_label = train_test_split(train_X,
train_Y_one_hot, test_size=0.2, random_state=13)
# nivel de frecuencia de aprendizaje
INIT_LR = 1e-3
#Cantidad de iteraciones completas al conjunto de imagenes de entrenamiento
epochs = 15
#Cantidad de imagenes que se toman a la vez en memoria
batch_size = 64
# modelo y capas
chuche_model = Sequential()
chuche_model.add(Conv2D(16, kernel_size=(3, 3), activation='linear',
padding='same', input_shape=(64, 64, 3)))
chuche_model.add(Conv2D(32, kernel_size=(3, 3), activation='linear', padding=
'same', input_shape=(64, 64, 3)))
chuche_model.add(Conv2D(64, kernel_size=(3,3), activation='linear', padding=
'same',input_shape=(64, 64, 3)))

chuche_model.add(LeakyReLU(alpha =0.1))
chuche_model.add(MaxPooling2D((2,2), padding= 'same'))
chuche_model.add(Dropout(0.5))

chuche_model.add(Flatten())
chuche_model.add(Dense(64, activation='linear'))
chuche_model.add(LeakyReLU(alpha=0.1))
chuche_model.add(Dropout(0.5))
chuche_model.add(Dense(nClasses, activation='softmax'))

chuche_model.summary()
chuche_model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adagrad(lr=INIT_LR, decay=INIT_LR/100), metrics
=['accuracy'])

```

```

chuche_train = chuche_model.fit(train_X, train_label, batch_size=batch_size,
epochs=epochs, verbose=1, validation_data=(valid_X, valid_label))

chuche_model.save("modelo_chuche_GAN.h5py")

#Evaluacion del modelo
test_eval = chuche_model.evaluate(test_X, test_Y_one_hot, verbose=1)

accuracy = chuche_train.history['accuracy']
val_accuracy = chuche_train.history['val_accuracy']
loss = chuche_train.history['loss']
val_loss = chuche_train.history['val_loss']
epochs = range(len(accuracy))

plt.plot(epochs, accuracy, 'bo', label='Training accuracy')
plt.plot(epochs, val_accuracy, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()

```