Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA DE SEGOVIA

Grado en Ingeniería Informática de Servicios y Aplicaciones

Generación de elementos mediante técnicas de Deep Learning

Alumno: Manuel Martínez Martínez

Tutor: José Vicente Álvarez Bravo

Generación de elementos mediante técnicas de Deep Learning

Manuel Martínez Martínez

Dedicado a mis padres:
Agustín y María Jesús,
mis hermanos:
José Antonio, Santiago,
Pilar, Juan e Inés,
y a todos mis amigos.

Resumen

En los últimos 10 años hemos visto grandes avances en el campo de la inteligencia artificial. Más recientemente, aplicaciones comerciales utilizando redes generativas, como son *DALL-E 2* o *Chat GPT* han mostrado el potencial de las herramientas de análisis del lenguaje natural y la generación de imágenes. Sin embargo, estas aplicaciones emergentes no serían posibles sin el desarrollo previo de las redes generativas adversariales, las primeras redes capaces de generar objetos nuevos extrapolando distribuciones de probabilidad de conjuntos de datos.

Para entender mejor el estado en el que se encuentra la tecnología en la actualidad, este trabajo presenta las redes precursoras de los métodos de atención y *Transformers*, utilizados en la actualidad para la generación de contenido, tanto escrito como visual. En este trabajo podremos observar el funcionamiento teórico de las redes generativas adversariales y los resultados que produjeron las primeras implementaciones de las mismas, y compararlos con la actualidad. Esto demostrará lo mucho que se ha avanzado en poco tiempo (alrededor de 9 años), además de ser una buena introducción para todo aquel que quiera entender el funcionamiento de la generación de elementos mediante técnicas de aprendizaje automático profundo.

El código implementado se puede encontrar en:
https://github.com/mapacheAgridulce/TFG-Generacion-elementos-DL.git

Palabras clave: aprendizaje automático, aprendizaje profundo, redes neuronales generativas, redes generativas antagónicas, redes generativas adversariales, *GAN, CGAN, DCGAN, LAPGAN.*

Abstract

The last 10 years have seen great advances in the field of artificial intelligence. More recently, commercial applications using generative networks, such as *DALL-E 2* or *Chat GPT* have shown the potential of natural language processing tools and image generation. However, these emerging applications would not be possible without the prior development of generative adversarial networks, the first networks capable of generating novel objects by extracting probability distributions from dataset information.

To better understand the current state of the art, this paper presents the precursor networks of attention methods and Transformers, currently used for content generation, both written and visual. In this paper we will be able to observe the theoretical operation of generative adversarial networks and the results produced by early implementations of them, and compare them with today. This will demonstrate how much progress has been made in a short time (about 9 years), and it will be a good introduction for anyone who wants to understand the way to item generation using deep machine learning techniques.

The implemented code can be found at:
https://github.com/mapacheAgridulce/TFG-Generacion-elementos-DL.git.

Keywords: *Machine Learning, Deep Learning, generative neural networks, generative adversarial networks, GAN, CGAN, DCGAN, LAPGAN.*

Índice general

Índice de figuras

Índice de cuadros

Siglas y abreviaturas

- Autocodificadores (*Autoencoders*): *AE*.
- Autocodificadores variacionales (*Variational Autoencoders*): *VAE*.
- Grandes modelos del lenguaje (*Large Language Models*): *LLM*.
- Inteligencia artificial: IA.
- Inteligencia artificial de propósito general (*Artificial General Intelligence*): *AGI*.
- Máquinas de vector soporte (*Support Vector Machine*): *SVM*.
- Modelos generativos preentrenados (*Generative Pre-Trained Transformer*): *GPT*.
- Procesamiento del lenguaje natural (*Natural Language Processing*): *NLP*.
- Redes convolucionales (*Convolutional Neural Networks*): *CNN*.
- Redes generativas adversariales (*Generative Adversarial Networks*): *GAN*.
- Redes generativas adversariales condicionales (*Conditional GAN*): *CGAN*.
- Redes generativas adversariales Laplacianas (*Laplacian GAN*): *LAPGAN*.
- Redes neuronales: RRNN.
- Redes recurrentes (*Recurrent Neural Network*): *RNN*.
- Redes recurrentes con umbral (*Gated Recurrent Unit*): *GRU*.
- Redes recurrentes de memoria a corto plazo (*Long Short-Term Memory*): *LSTM*.

Parte I

Descripción del proyecto

Capítulo 1

Planteamiento del proyecto

Comentario inicial: El código implementado se puede encontrar en el repositorio de *GitHub*:
https://github.com/mapacheAgridulce/TFG-Generacion-elementos-DL.git.

1.1. Masificación de la IA

En plena era de la información, es natural encontrar noticias, libros, artículos científicos y profesionales e incluso opiniones contrapuestas sobre cualquier avance tecnológico, todo gracias a Internet. Aparte de que la información nunca haya sido tan accesible como lo es hoy en día, la Red ha permitido a miles de millones de usuarios compartir información en diversos formatos: vídeos, imágenes, texto, audio... Esto ha aumentado grandiosamente la cantidad de datos de que se disponen y ha ayudado al desarrollo científico fuertemente, pues en la actualidad es mucho más sencillo divulgar el conocimiento a través de páginas web, blogs, canales de *YouTube* y muchos medios más.

Internet se ha convertido en un vehículo de información a gran escala y de forma masiva, información que es codificada y transmitida, además de disponible para la gran mayoría del público. La gran mayoría de esta información tiene un formato predefinido y, aunque se representen de distintas formas, en lo que al código se refiere, nos referiremos a toda esta información como datos. De entre todos estos datos, prestaremos atención a un tipo muy específico, que son las imágenes. Desde el inicio de la humanidad, una manera que hemos tenido de expresarnos como especie ha sido el dibujo, el arte, y más recientemente la fotografía e imágenes digitales. En la actualidad, uno de los formatos de dato más utilizados son las imágenes, ya sea para transmitir sentimientos o emociones, o para simplemente mostrar a nuestros amigos y conocidos el estado actual de nuestra vida.

A su vez, las ramas, más populares recientemente, de *Big Data* y *Machine Learning* aprovechan la accesibilidad a los datos disponibles en esta Red para cumplir con objetivos de análisis masivos de datos, clasificación de objetos por clases, reconocimiento de objetos en imágenes y mucho más. En 2014 se presentó el artículo [16], que rompía con algunos de los esquemas establecidos en el campo de la IA hasta el momento, y actualmente lo sigue haciendo; hablamos de las redes generativas. Estas redes, para la mayoría de la gente,

son un misterio y nos acercan a la ciencia ficción, pues son capaces de generar texto como si habláramos con ellas, o imágenes bajo petición, e incluso algunas son capaces de hacer ambas cosas a la vez. Alrededor de Octubre de 2022 se ha podido observar un creciente interés en estas redes generativas, más específicamente, aquellas que son capaces de extrapolar características de los datos u objetos con los que se trabaja y generar nuevos objetos similares (como lo son *DALL-E 2* o *chatGPT*). Se podría pensar que hemos llegado a un punto en el que verdaderamente hemos infundido de creatividad y libertad a nuestros agentes inteligentes, pero, como veremos en este proyecto, aún queda mucho avance por hacer, y estos agentes tienen margen de mejora.

El comienzo de estas redes se basa en un único formato, en el que la red generativa muestra el nuevo contenido, pero no interpretando el lenguaje natural, y generando acorde a esa petición, más cercana al usuario de a pie, contenido "a la carta". Actualmente esta barrera parece superada hace mucho tiempo por el paradigma de la multimodalidad, ahora en boca de todos. Esta sensación viene, sencillamente, de la velocidad a la que se han comenzado a desarrollar nuevas e imponentes herramientas como lo son *GPT-4* de *OpenAI* o su implementación, anunciada en una conferencia online y con información disponible en [1], que pretende revolucionar la manera de enfocar el trabajo reduciendo la carga de trabajo mediante un asistente automático. Estas nuevas herramientas utilizan el aprendizaje multimodal, como ya hemos mencionado, del cual hablaremos como potenciación de las redes generativas y se puede obtener más información leyendo [5] o [2].

Centrémonos de nuevo en lo que acontecía antes de estas redes multimodales. La base que alimenta estas nuevas herramientas, las redes generativas adversariales, presentadas en [16] fueron un descubrimiento fascinante que en tan solo 9 años ha avanzado a pasos agigantados. Estas redes son el pilar fundamental de los nuevos paradigmas de aprendizaje para los sistemas inteligentes, además de los *LLM*, de las nuevas herramientas inteligentes. Nuestro objetivo principal será presentar los conceptos de estas bases, tanto teórica como prácticamente, pues entendiendo su funcionamiento podremos entender como funcionan muchos de los agentes inteligentes disponibles en la Red actualmente.

1.2. Objetivos del trabajo

Como ya hemos mencionado en la sección anterior, la imagen es uno de los datos predominantes en la vida de los usuarios de la Internet, ya sea a través de portales web, redes sociales o comunidades *online*. Es entonces lógico que uno de los campos de la IA que más haya sorprendido al público sea la generación de imágenes. Esto se debe a que sus resultados son, no sólo vistosos, si no, en ocasiones, bastante "humanos", pues pareciera que no hayan podido ser producidos por una máquina.

Como ya se ha dicho, todas las nuevas herramientas generativas parten de la misma base, las redes generativas, comúnmente llamadas *GAN*. Estas son redes neuronales profundas que permiten generar nuevo contenido tras realizar su entrenamiento con un *dataset* predefinido. El contenido generado estará íntimamente relacionado con el conjun-

to de datos con el que alimentemos la red, y dependiendo de las técnicas aplicadas sobre el modelo generativo, podremos asegurar flexibilidad, comprensión del tipo de datos y generación de datos de ese tipo, robustez, y en algunos casos mayor simplicidad a la hora de realizar el entrenamiento.

El objetivo de este trabajo es abordar las *GAN*, tanto en el marco teórico, con un enfoque estadístico basado en la inferencia, como en el marco práctico, mostrando distintos ejemplos de redes generativas y su funcionamiento sobre un conjunto de datos. Siendo más específicos, este proyecto espera lograr los siguientes objetivos:

- Introducir los fundamentos teóricos para entender las *GAN*.
- Mostrar las diferencias y similitudes de estas redes con las técnicas de *Deep Learning* anteriores a ellas.
- Explicar en detalle la arquitectura de tres de las primeras redes generativas: la *DCGAN*, la *CGAN* y la *LAPGAN*.
- Implementar ejemplos de estos modelos generativos y mostrar sus resultados.
- Detallar la situación actual de este tipo de redes y sus implementaciones.
- Obtener conclusiones sobre el funcionamiento de los modelos actuales a partir de lo redactado en este trabajo.

Expliquemos un poco más en detalle estos objetivos. El primero de ellos es completamente necesario, pues las redes generativas plantearon un nuevo marco de entrenamiento para los algoritmos de aprendizaje automático. El conocimiento teórico sobre el funcionamiento de estas redes es necesario para poder comprender su funcionamiento completamente, dado que la red va más allá del funcionamiento de la misma. Este objetivo tiene como alcance explicar cual es el razonamiento que se siguió para implementar estas redes, cómo se han de entrenar, cuáles serán los mecanismos de evaluación de los resultados y finalmente los resultados que producen tras entrenarlas.

También está dentro del alcance del trabajo dar un contexto sobre el funcionamiento de las redes neuronales anteriores a las *GAN* y mostrar el estado del arte de las redes actuales y la evolución de estas redes generativas durante la última década. Principalmente, estas redes supusieron un pequeño punto de inflexión en la forma de entrenar agentes inteligentes y de producir resultados, pues consiguen extrapolar características de los datos y producir salidas similares. El punto de inflexión fue principalmente el cambio del reconocimiento de objetos y segmentación en imágenes a la producción de las mismas. De forma secundaria, se tratará de mostrar cómo las redes actuales superan con creces a estas primeras, sus precursoras debido a nuevas técnicas, ligeramente más modernas, que permiten observar estas mejoras.

Además de lo anterior, también se mostrará la estructura por capas que siguen las redes *DCGAN*, *CGAN* y *LAPGAN*, además de implementarlas. Con esto se pretende dar un enfoque práctico del funcionamiento y entrenamiento de estas redes, con el *datasets*

MNIST y *CIFAR10*. Aún así, pese a que el apartado teórico se limita al estudio y no requiere más que horas de recolección, lectura y redacción de documentos, se presenta tras esta parte el problema principal de un desarrollo así planteado: el exigente coste computacional exigido por el entrenamiento de estas redes. Una de las maneras en que se ha puede reducir el coste sería mediante la ejecución con librerías adaptadas y ya optimizadas, y la provisión de recursos en la nube mediante *Google Colaboratory* o *Microsoft Azure*, del cual tenemos 100 $ disponibles gracias al servicio *Github Student Developer Pack*. Con esto pretendemos mostrar, también, algunas de las alternativas al alcance de los alumnos para realizar tareas que puedan requerir otros enfoques más allá de la programación y prueba en local. Todas estas herramientas se introducirán al final del trabajo en los manuales de usuario.

Finalmente, compararemos los modelos actuales y se justificará su papel en el desarrollo de las herramientas de moda actuales.

1.3. Herramientas utilizadas

Esta sección servirá para introducir las herramientas que se han utilizado en el desarrollo de este trabajo, tanto para la redacción como para la implementación de las redes generativas.

Para la redacción de la memoria se han utilizado tres herramientas principalmente: LaTeX, dentro de *Visual Studio Code*, *Microsoft Excel* y *Microsoft Teams*.

LaTeXes, como podemos ver de [33], un sistema de preparación de documentos para una escritura de alta calidad, tanto para trabajos técnicos como para documentos científicos, lo cual describe esta memoria de trabajo. El sistema se centra en olvidarse del diseño y centrarse en el contenido de los documentos. Esto se consigue mediante el etiquetado de los elementos gráficos, y el procesamiento de las distintas funciones para dar formato al texto, de este modo, no es necesario perder el tiempo en cambiar el tamaño de la fuente, u otras cualidades del diseño del documento, para obtener un diseño aceptable. Si se desea saber más, se puede visitar la web [39], que presenta tutoriales del uso, además del almacenamiento de documentos LaTeXen la nube, para su fácil acceso. Por otra parte, se presentará en A la manera en que se puede trabajar en local con LaTeXen *VS Code*.

La segunda de estas herramientas, *Excel*, se ha utilizado como herramienta de diseño de cronogramas para realizar los diagramas de Gantt que han guiado la planificación del trabajo, como se podrá ver en 2.2. *Excel*, es una herramienta del entorno *Microsoft 365*, desarrollada por *Microsoft* para el análisis de datos, pero que con el tiempo se ha convertido en una herramienta útil en el campo del diseño de cronogramas (aunque existe *Microsoft Project*) y otras gráficas y organigramas de distinto tipo. Se dispone de información más detallada de esta herramienta en [27] y los foros de *Microsoft* para las herramientas de ofimática de *Office 365*.

Por otra parte, las herramientas importantes en la implementación de las redes generativas son librerías e *IDE*s del lenguaje *Python*. Como entorno de desarrollo para las redes se ha escogido utilizar *Spyder* junto con *Anaconda Navigator*, herramientas que

se han utilizado a lo largo de la carrera para el desarrollo de *scripts* de *Python*, y con las que el estudiante ya está familiarizado. Además, ha sido necesario instalar las librerías de *TensorFlow*, *PyTorch* que permiten el desarrollo de redes neuronales de forma sencilla, definiendo las capas que se van a utilizar y conectándolas en forma de modelo. Adicionalmente, también se ha instalado *CUDA* para poder aprovechar la potencia gráfica del ordenador en el que se entrenan las redes generativas, y de este modo obtener entrenamientos más veloces de las redes.

La última herramienta destacable para el buen desarrollo de la memoria ha sido *Microsoft Teams*. Esta herramienta permite programar reuniones, subir archivos y mantener conversaciones vía *chat*, y esto se ha utilizado para mantener informado de los avances al tutor del trabajo, y para realizar reuniones sobre el estado del trabajo, como veremos en 2.2.

Veamos con más profundidad en que consisten estas librerías, pues son las herramientas principales de desarrollo del trabajo.

Comenzaremos explicando que son *TensorFlow* y *Keras*, dos de las librerías más utilizadas para el desarrollo e implementación de modelos de IA. *TensorFlow* es una librería de código abierto con soporte para múltiples lenguajes, *Python* entre ellos. Completando la utilidad de *TensorFlow* se utiliza la librería *Keras*, que está destinada al desarrollo de modelos de aprendizaje automático, y actúa como interfaz de *TensorFlow*. Esta librería simplifica el desarrollo de redes profundas y soporta redes convolucionales, que utilizaremos para el desarrollo de dos de las tres redes que se presentan en el trabajo.

Por otra parte, también utilizaremos *PyTorch*, otra librería para la implementación de redes neuronales. *PyTorch* ofrece cualidades similares a las de *TensorFlow*: posee una buena integración con *Python*, tiene buena documentación sobre su funcionamiento y la manera de producir redes es similar, pues se basa en generar un grafo que define el paso de los datos por la red neuronal.

Finalmente, se ha utilizado *CUDA*, que aparte de estar adaptada como librería para *Python*, es un conjunto de herramientas con un único objetivo: paralelizar el procesamiento en tarjetas de *NVIDIA*, lo cual acelera el proceso de entrenamiento de RRNN de manera significativa.

Para la instalación de estas librerías y los *IDE*s necesarios para replicar los experimentos de este trabajo también se puede ver A.

1.4. Formato de la memoria

Dado que este trabajo no es tan similar al estilo de memoria de un proyecto de aplicación empresarial o web, se ha modificado ligeramente el orden y estilo general de los capítulos, para adaptarlo más a un trabajo de alcance teórico-práctico. De este modo, dividimos la memoria en tres partes con distintos capítulos y secciones, dividiendo el trabajo en:

- La descripción del trabajo, que se conforma de la motivación, la planificación temporal y económica, el planteamiento previo de conceptos teóricos y la explicación

teórica principal del proyecto.

- La solución propuesta, donde presentaremos las arquitecturas e implementaciones realizadas, además de introducir los conjuntos de datos con los que trabajaremos y los resultados del trabajo.

- Los apéndices que contendrán los manuales y la información para repetir los experimentos, así como la bibliografía utilizada para su desarrollo.

Si recorremos las secciones de estas tres partes veremos que el orden del trabajo pretende ir incrementando la dificultad de los conceptos progresivamente, después de haber detallado lo necesario para su buena planificación y desarrollo. Una vez se tengan todos los conceptos desarrollados, se implementarán, dando lugar a la parte práctica del proyecto. El trabajo pretende terminar realizando conclusiones y explicando los procesos y elecciones de implementación tomadas.

Primero se plasmará la planificación temporal y económica del trabajo, así como la metodología seguida para el desarrollo de las redes que se utilizarán. Tras ello, se introducirán conceptos básicos relacionados con las redes neuronales que serán necesarios para comprender mejor el trabajo, de modo que los conceptos principales no resulten extraños para el lector nuevo y refresquen la memoria de los lectores con conocimiento sobre lo redactado. Aparte, también se introducirán las últimas técnicas de entrenamiento de redes generativas y se mencionarán recursos que ampliarán los conceptos de las redes que se presentarán.

Tras esto se presentarán y explicarán las diferentes soluciones propuesta al problema de generar contenido a partir de un conjunto de datos previo, sean del tipo que sean. En primer lugar se realizará un pequeño análisis del problema y una explicación teórica sobre el funcionamiento de las *GAN* sobre el papel. En segundo lugar se introducirán los conjuntos de datos para el entrenamiento y se detallará como se han clasificado y tratado las imágenes que servirán de alimento para la red generativa. Finalmente se mostrarán el proceso de entrenamiento de la red y los resultados obtenidos, y se obtendrán conclusiones sobre los resultados obtenidos y el proceso general del proyecto.

A partir de ese momento tendremos los apéndices, orientados a proveer de manuales de instalación para reproducir el trabajo y dar contenido adicional con el que ampliar el conocimiento sobre las redes generativas y se recopila la bibliografía utilizada para el desarrollo de este trabajo.

Capítulo 2

Planificación

Este siguiente capítulo pretende reflejar tres cosas:

1. La metodología que se ha seguido en el desarrollo y redacción del trabajo.

2. La planificación y el presupuesto que se han realizado a priori con técnicas vistas a lo largo de la carrera.

3. La comparación a posteriori entre los objetivos logrados y la planificación y presupuestos realizados.

Lo primero que haremos será explicar las peculiaridades de este trabajo y la metodología que se ha seguido, dando razones para la elección de esta. Tras esto, se presentarán la planificación temporal y el presupuesto, para, finalmente, mostrar la comparación entre tiempo real y planificado.

2.1. Metodología de trabajo

Como ya hemos mencionado en la sección 1.4, la estructura de este proyecto está más centrada en la tarea de investigación e implementación que en el desarrollo de una aplicación empresarial o web; por eso mismo, se ha escogido una metodología de trabajo basada en el modelo en cascada iterativo, que procederemos a explicar ahora.

El modelo en cascada es un modelo lineal que divide los procesos de desarrollo en fases sucesivas. Estas fases se ejecutarían sólo una vez, pero al realizar un método en cascada iterativo repetiremos alguna de ellas, como veremos más adelante en la figura 2.1.

La propuesta de modelo en cascada para el ciclo de vida del desarrollo de una herramienta *software* sigue las siguientes fases:

- Análisis: Esta es la primera fase del ciclo de vida de la aplicación, donde se realiza un estudio de viabilidad, con el objetivo de determinar costes, rentabilidad y factibilidad del proyecto. Esto nos sirve para poder obtener las condiciones que han de cumplirse para tener un desarrollo exitoso, el plan a seguir para la ejecución del proyecto y la

estimación del presupuesto. También se definen los requisitos del proyecto *software* que se va a realizar, incluyendo descripciones detalladas de las tareas y subtareas.

- Diseño: La segunda fase del ciclo de vida del proyecto se centra en la formulación de la solución al problema planteado. Generalmente, es en esta fase en la que se diseña la arquitectura *software* a seguir, así como un plan detallado de la solución, donde se detallan los componentes y su funcionamiento.

- Implementación: Durante la tercera fase del desarrollo se ejecuta el plan diseñado en la fase anterior. Ejecutar este plan incluye las tareas de programación, búsqueda de errores y pruebas unitarias, con el propósito de traducir el diseño del proyecto al lenguaje de programación. Cada componente de la aplicación final se programa por separado y se prueba unitariamente, para unificar las distintas partes en una solución completa.

- Prueba: En esta fase del proyecto, realizamos la integración del sistema programado en el entorno seleccionado para su ejecución. Además, realizamos las pruebas de aceptación del sistema, lo que nos permite determinar si el *software* entregado cumple con las exigencias definidas en la fase de análisis.

- Mantenimiento: Finalmente, en la quinta fase, se realizan tareas de monitorización y autorización de uso para la solución implementada y probada. En ocasiones esto también implica realizar mejoras sobre lo entregado y la solución de errores imprevistos en las fases anteriores.

Sin embargo, estas fases sirven estrictamente para el desarrollo de herramientas *software*, así que se han adaptado al trabajo, pues para un Trabajo Fin de Grado centrado en la investigación, recolección de fuentes e implementación de una serie de modelos no servirían.

Utilizando esta manera de desarrollar como esqueleto, procedemos a presentar la metodología planteada para el desarrollo de este trabajo. Igual que en el caso del desarrollo en cascada, utilizaremos cinco fases diferenciadas para cumplir con los objetivos planteados en la sección 1.2:

- Análisis: El proceso de análisis se realiza sobre todo el documento, además de sobre las implementaciones que se realizarán de redes generativas. El análisis comienza antes de iniciar el trabajo, de modo que se pueda obtener un plan conjunto para la redacción del proyecto y la investigación de fuentes para el mismo, además de un plan para la implementación y entrenamiento de los modelos de IA que se ejecutarán.

- Diseño: Esta etapa se sustituye por el planteamiento de un modelo de memoria estructurado para el trabajo, además de la búsqueda sobre las arquitecturas de red generativa que se implementarán, de modo que únicamente sea necesario programar la arquitectura que las define, adaptándolas al *dataset* que se vaya a utilizar.

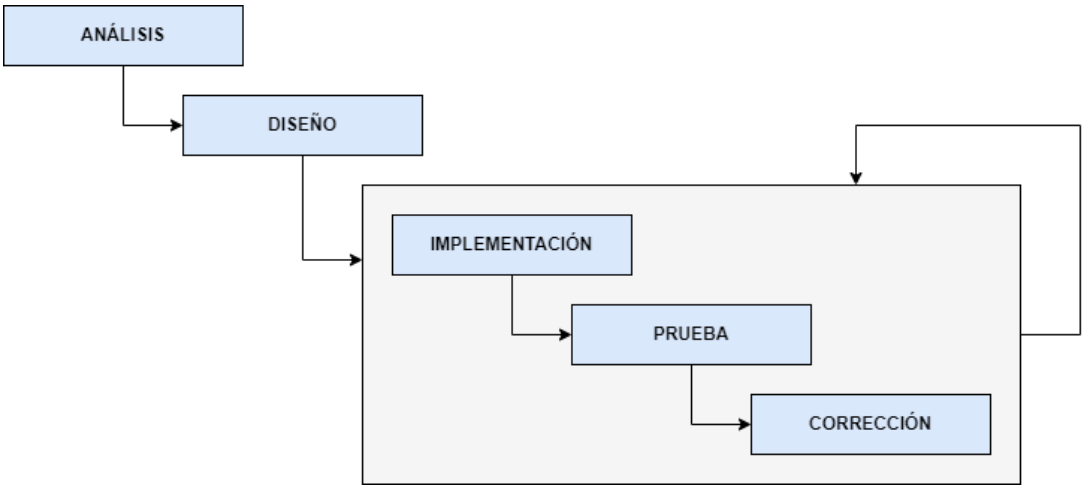


Figura 2.1: Método de cascada adaptado

- Implementación: La implementación se transforma en una fase de doble tarea, pues no se centrará solo en el desarrollo del *software* necesario, si no que se incluye como implementación la redacción e investigación de cada apartado de la memoria que se presenta.
- Prueba: En la etapa de pruebas nos centraremos en localizar erratas, estudiar la cohesión de los apartados redactados y en el entrenamiento y prueba de los modelos GAN programados.
- Corrección: En esta etapa realizaremos correcciones ortográficas sobre la memoria, añadiremos cohesión entre distintos apartados en caso de ser necesario y modificaremos el código programado en caso de que no se obtengan los resultados esperados.

Además, la adaptación del modelo busca, como se puede ver en la imagen 2.1, mediante la iteración de estas tres últimas fases, una mejora iterativa de los modelos programados y de la memoria redactada. De este modo se consigue una mejora progresiva sobre un esqueleto bien formado, tanto en lo referido a los modelos como en lo referido a la redacción de la teoría sobre los mismos.

La manera de proceder será la siguiente, primero se realizará la identificación y redacción de los requisitos y tareas, con su correspondiente división en subtareas. Con los requisitos detallados, realizaremos una serie de diagramas de Gantt que nos permitirán seguir cronológicamente el desempeño de las tareas, con un apartado de comentarios para observar los impedimentos y problemas que se han generado en cada iteración y resolverlos en la siguiente.

Una vez terminadas estas tareas de análisis y diseño, procederemos a comenzar con la redacción de la memoria y redes y las pruebas y correcciones necesarias. Estas fases se repetirán de forma sucesiva durante los meses de desarrollo del proyecto, de modo que inicialmente bastará con tener un esqueleto con el que trabajar, para sucesivamente refinar el mismo hasta obtener un resultado final bien redactado y completamente funcional.

Para saber que tareas hay que realizar, concretamos una lista que especifica los apartados del trabajo y las tareas que pertenecen a cada uno de los apartados. La lista es la siguiente:

1. Planteamiento del trabajo:
    - Investigación sobre la masificación de la IA.
    - Análisis del objetivo del trabajo.
    - Redacción de las secciones: introducción al trabajo.
    - Redacción de las secciones: objetivos del trabajo.
    - Redacción de las secciones: herramientas de desarrollo.
    - Redacción de las secciones: formato de la memoria.
2. Planificación:
    - Análisis temporal del trabajo.
    - Análisis del presupuesto del trabajo.
    - Repaso de las distintas metodologías.
    - Recolección de información para la comparativa.
    - Redacción de las secciones: metodología utilizada.
    - Redacción de las secciones: planificación temporal.
    - Redacción de las secciones: presupuesto.
    - Redacción de las secciones: comparativa entre realidad y planes.
3. Conceptos previos:
    - Repaso de conceptos básicos: *ML*, *DL*, tipos de aprendizaje.
    - Investigación sobre el estado del arte.
    - Investigación sobre el funcionamiento de las *CNN*.
    - Redacción de las secciones: introducción mediante el estado del arte.
    - Redacción de las secciones: conceptos básicos *ML*, *DL*, tipos de aprendizaje y *CNN*.
4. Redes generativas:
    - Investigación sobre el funcionamiento de las *GAN*.
    - Investigación sobre la implementación de las *CGAN*.
    - Investigación sobre la implementación de las *LAPGAN*.
    - Investigación sobre la implementación de las *DCGAN*.

- Redacción de las secciones: funcionamiento del entrenamiento adversarial.
- Redacción de las secciones: *CGAN*.
- Redacción de las secciones: *LAPGAN*.
- Redacción de las secciones: *DCGAN*.

5. Arquitectura e implementación:
   - Redacción de las secciones: arquitectura de una *CGAN*.
   - Redacción de las secciones: arquitectura de una *LAPGAN*.
   - Redacción de las secciones: arquitectura de una *DCGAN*.
   - Implementación de una red *CGAN* en *Python*.
   - Implementación de una red *LAPGAN* en *Python*.
   - Implementación de una red *DCGAN* en *Python*.
6. *Dataset* utilizados:
   - Recolección de información sobre *MNIST*.
   - Recolección de información sobre *CIFAR10*.
   - Redacción de las secciones: *datasets* utilizados y sus peculiaridades.
7. Entrenamiento y resultados:
   - Recolección de datos del entrenamiento de la *CGAN* implementada.
   - Recolección de datos del entrenamiento de la *LAPGAN* implementada.
   - Recolección de datos del entrenamiento de la *DCGAN* implementada.
   - Redacción de las secciones: resultados de una *CGAN*.
   - Redacción de las secciones: resultados de una *LAPGAN*.
   - Redacción de las secciones: resultados de una *DCGAN*.
8. Conclusiones:
   - Redacción de las secciones: resumen del trabajo en inglés y español.
   - Redacción de las secciones: competencias obtenidas.
   - Redacciones de las secciones: objetivos cumplidos e impedimentos encontrados.
9. Apéndices:
   - Redacción de las secciones: manual de instalación y ejecución.

Sin embargo, aunque la disposición de las tareas se a ordenado por apartados, no se seguirá este orden para redactar el trabajo e implementar las redes, por ejemplo, no sería un planteamiento real redactar el apartado sobre las herramientas utilizadas para desarrollar la solución antes de haber implementado las redes.

2.2. Planificación temporal

Lo primero que se ha realizado para poder llevar a cabo la planificación temporal del proyecto ha sido calcular el número de horas diarias disponibles para las tareas de implementación, así como las horas estimadas de trabajo para la asignatura Trabajo Fin de Grado.

La asignatura consta de 12 créditos *ECTS* que equivalen a 300 horas de trabajo (25 horas de trabajo por crédito). Son estas 300 horas disponibles sobre las que generaremos la planificación del trabajo, que abarcará los meses de Abril y Mayo, dado que se estiman 5 horas diarias dedicadas a las tareas de investigación, redacción e implementación de los elementos del proyecto. Si se tienen en cuenta ese número de horas diarias, en total se obtienen 60 días hábiles para el desarrollo del proyecto, es decir, dos meses (aproximadamente) de ciclo de vida.

Para el desarrollo del trabajo no se contemplan vacaciones ni días libres, de modo que se incluyen los festivos y los fines de semana como días laborables también. No tomar días festivos ni de descanso sirve para mantener un buen ritmo de trabajo y no caer en retrasos y llevar a término correctamente el desarrollo del proyecto. Aunque no se consideran días de descanso, debido a causas personales, los días 13/05/2023 y 20/05/2023 no se podrán realizar tareas de desarrollo, de modo que se tomarán los primeros días de Junio para terminar las tareas que se hayan retrasado, estimando que no serán necesarios más de 10 días de Junio como margen para la resolución de tareas desplazadas por motivos personales. Esto no se reflejará en los diagramas, pero aquellas tareas que queden desplazadas serán las que se destinen a estos días.

Con el objetivo de llevar a cabo la planificación temporal se han realizado dos diagramas de Gantt, uno para cada mes, en los que se incluyen las distintas tareas y subtareas necesarias para llevar a término el proyecto y las soluciones propuestas. En estos diagramas se sigue el mismo esquema:

- Cada tarea y subtarea relacionadas tienen un identificador y una breve descripción en forma abreviada (que servirá de título).

- Solo hay un empleado disponible, ya sea para las tareas de redacción, investigación o implementación, que es el alumno que realiza el trabajo.

- Se establecen una fecha de inicio y de fin para cada tarea, y las subtareas se distribuyen dentro de ese marco temporal, pues resulta más sencillo fijar una fecha de inicio y una fecha límite que asignar puntos de esfuerzo a cada tarea y calcular el número de horas adecuado para cada punto de esfuerzo.

- Cada tarea y subtarea pueden tener distintos estados de entre la siguiente selección: “Programada”, “Iniciada”, “En progreso”, “Terminada” o “Retrasada”. Todas las tareas pasarán por los cuatro primeros estados de forma secuencial, pasando por el quinto en caso de retrasos respecto a la previsión realizada. Este último estado sirve para, como veremos a posteriori, recibir *feedback* sobre el retraso y poder realizar

la comparativa entre lo planificado y la realidad. También tendrá será de utilidad para las reuniones desplazadas.

- Para cada uno de los meses se plantean dos reuniones para el seguimiento del desempeño, una a mediados de mes y otra a finales, con una fecha flexible y dependiente d ela disponibilidad de alumno y tutor.

Estos diagramas se revisan a diario para poder realizar el seguimiento de las tareas, y se les añade comentarios a las tareas en las últimas celdas, de modo que así exista retroalimentación y se pueda realizar la comparación entre el planteamiento y el desempeño del trabajo. De este modo es, también, más sencillo identificar retrasos, tareas complejas y partes del trabajo que requieren de mayor esfuerzo o un enfoque distinto.

Para asignar el marco temporal a las distintas tareas, se ha estimado la fecha de inicio y de final de cada una, suponiendo que se pueden trabajar 4 horas los días de entre semana y 8 los fines de semana, que se aproxima a una media de 5 horas por día cada semana. Las tareas se dividen para la redacción de la memoria y el entrenamiento de las redes a lo largo del mes.

Como el desarrollo de la memoria consiste en un refinado sucesivo de lo escrito, se dedicará la mayor parte del tiempo a esto, pues la implementación de los modelos, una vez realizada, sólo necesita de tiempo para el entrenamiento, que se puede realizar en segundo plano. La estimación temporal de las tareas de redacción se ha realizado procurando analizar la complejidad de cada una de las partes de esta memoria. A su vez, la estimación temporal del entrenamiento e implementación se asigna a una tarea continua de implementación de los modelos y entrenamiento que se realiza a lo largo de todo el mes.

Además de utilizar los diagramas para llevar el seguimiento de las tareas, se ha utilizado *Microsoft Teams* como herramienta de seguimiento, así como un repositorio de *GitHub* para el almacenamiento de un respaldo para el código, la memoria y el seguimiento del progreso del trabajo.

Veamos la planificación teórica de los dos meses.

Comencemos con la planificación de las tareas para Abril. Dado que es el mes con las tareas de análisis y diseño, los días iniciales del proyecto se dedican a la estructura de la memoria y la investigación sobre las implementaciones de distintas redes generativas, concluyendo que aquellas que se van a desarrollar son la *CGAN*, la *LAPGAN* y la *DCGAN*.

Tras el análisis de la situación y cómo enfocar el trabajo, los días siguientes se dedican a estimar la planificación temporal y considerar la metodología de desarrollo del proyecto.

Con esto terminado, se asignan las siguientes tareas para el primer mes:

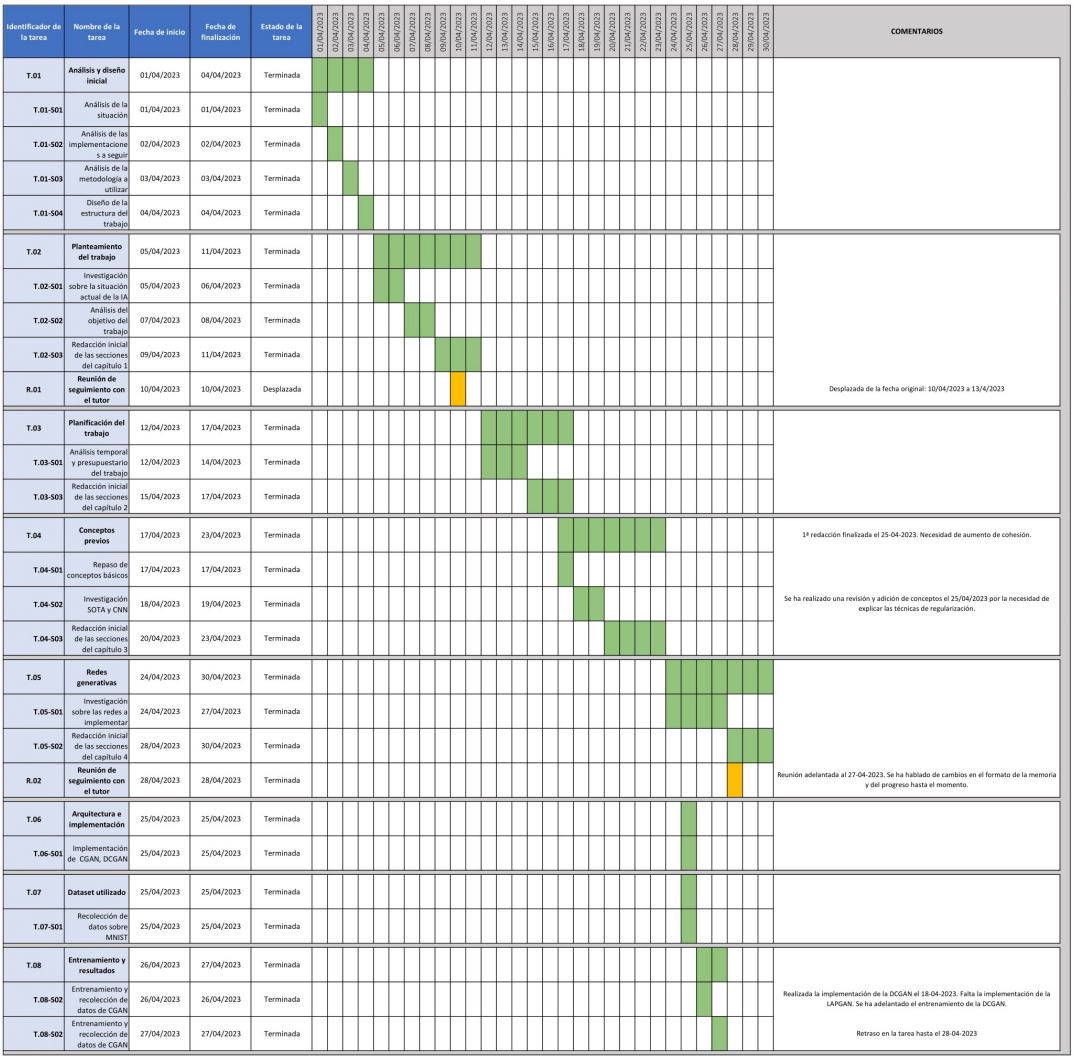
1. Planteamiento del trabajo:
    - Investigación sobre la masificación de la IA.
    - Análisis del objetivo del trabajo.
    - Redacción de las secciones del capítulo 1

2. Planificación:
    - Análisis temporal del trabajo.
    - Análisis del presupuesto del trabajo.
    - Repaso de las distintas metodologías.
    - Redacción de las secciones del capítulo 2.
3. Conceptos previos:
    - Repaso de conceptos básicos: *ML*, *DL*, tipos de aprendizaje.
    - Investigación sobre el estado del arte.
    - Investigación sobre el funcionamiento de las *CNN*.
    - Redacción de las secciones del capítulo 3.
4. Redes generativas:
    - Investigación sobre el funcionamiento de las *GAN*.
    - Investigación sobre la implementación de las *CGAN*.
    - Investigación sobre la implementación de las *LAPGAN*.
    - Investigación sobre la implementación de las *DCGAN*.
    - Redacción de las secciones del capítulo 4.
5. Arquitectura e implementación:
    - Implementación de una red *CGAN* en *Python*.
    - Implementación de una red *DCGAN* en *Python*.
6. *Dataset* utilizados:
    - Recolección de información sobre *MNIST*.
7. Entrenamiento y resultados:
    - Recolección de datos del entrenamiento de la *CGAN* implementada.
    - Recolección de datos del entrenamiento de la *DCGAN* implementada.

Podemos ver la distribución de las tareas en el diagrama de Gantt dedicado a este mes (figura 2.2).

Este diagrama incluye también el tiempo dedicado al análisis y diseño del esqueleto de la memoria y del análisis y diseño de los diagramas temporales, que se realizan los primeros días del mes de Abril.

Presentamos ahora la lista de tareas para el mes de Mayo:



Identificador de la tarea	Nombre de la tarea	Fecha de inicio	Fecha de finalización	Estado de la tarea	COMENTARIOS
T.01	**Análisis y diseño inicial**	01/04/2023	04/04/2023	Terminada	
T.01-S01	Análisis de la situación	01/04/2023	01/04/2023	Terminada	
T.01-S02	Análisis de las implementaciones a seguir	02/04/2023	02/04/2023	Terminada	
T.01-S03	Análisis de la metodología a utilizar	03/04/2023	03/04/2023	Terminada	
T.01-S04	Diseño de la estructura del trabajo	04/04/2023	04/04/2023	Terminada	
T.02	**Planteamiento del trabajo**	05/04/2023	11/04/2023	Terminada	
T.02-S01	Investigación sobre la situación actual de la IA	05/04/2023	06/04/2023	Terminada	
T.02-S02	Análisis del objetivo del trabajo	07/04/2023	08/04/2023	Terminada	
T.02-S03	Redacción inicial de las secciones del capítulo 1	09/04/2023	11/04/2023	Terminada	
R.01	**Reunión de seguimiento con el tutor**	10/04/2023	10/04/2023	Desplazada	Desplazada de la fecha original: 10/04/2023 a 13/4/2023
T.03	**Planificación del trabajo**	12/04/2023	17/04/2023	Terminada	
T.03-S01	Análisis temporal y presupuestario del trabajo	12/04/2023	14/04/2023	Terminada	
T.03-S03	Redacción inicial de las secciones del capítulo 2	15/04/2023	17/04/2023	Terminada	
T.04	**Conceptos previos**	17/04/2023	23/04/2023	Terminada	1ª redacción finalizada el 25-04-2023. Necesidad de aumento de cohesión.
T.04-S01	Repaso de conceptos básicos	17/04/2023	17/04/2023	Terminada	
T.04-S02	Investigación SOTA y CNN	18/04/2023	19/04/2023	Terminada	Se ha realizado una revisión y adición de conceptos el 25/04/2023 por la necesidad de explicar las técnicas de regularización.
T.04-S03	Redacción inicial de las secciones del capítulo 3	20/04/2023	23/04/2023	Terminada	
T.05	**Redes generativas**	24/04/2023	30/04/2023	Terminada	
T.05-S01	Investigación sobre las redes a implementar	24/04/2023	27/04/2023	Terminada	
T.05-S02	Redacción inicial de las secciones del capítulo 4	28/04/2023	30/04/2023	Terminada	
R.02	**Reunión de seguimiento con el tutor**	28/04/2023	28/04/2023	Terminada	Reunión adelantada al 27-04-2023. Se ha hablado de cambios en el formato de la memoria y del progreso hasta el momento.
T.06	**Arquitectura e implementación**	25/04/2023	25/04/2023	Terminada	
T.06-S01	Implementación de CGAN, DCGAN	25/04/2023	25/04/2023	Terminada	
T.07	**Dataset utilizado**	25/04/2023	25/04/2023	Terminada	
T.07-S01	Recolección de datos sobre MNIST	25/04/2023	25/04/2023	Terminada	
T.08	**Entrenamiento y resultados**	26/04/2023	27/04/2023	Terminada	
T.08-S02	Entrenamiento y recolección de datos de CGAN	26/04/2023	26/04/2023	Terminada	Realizada la implementación de la DCGAN el 18-04-2023. Falta la implementación de la LAPGAN. Se ha adelantado el entrenamiento de la DCGAN.
T.08-S02	Entrenamiento y recolección de datos de CGAN	27/04/2023	27/04/2023	Terminada	Retraso en la tarea hasta el 28-04-2023

Figura 2.2: Diagrama de Gantt para el mes de Abril

1. Revisión del documento:
   - Revisión de la redacción y aumento de la cohesión al inicio del mes.
2. Arquitectura e implementación:
   - Implementación de una red *LAPGAN* en *Python*.
3. *Dataset* utilizados:
   - Recolección de información sobre *CIFAR10*.
   - Redacción de las secciones de los capítulos 5 y 6.
4. Entrenamiento y resultados:

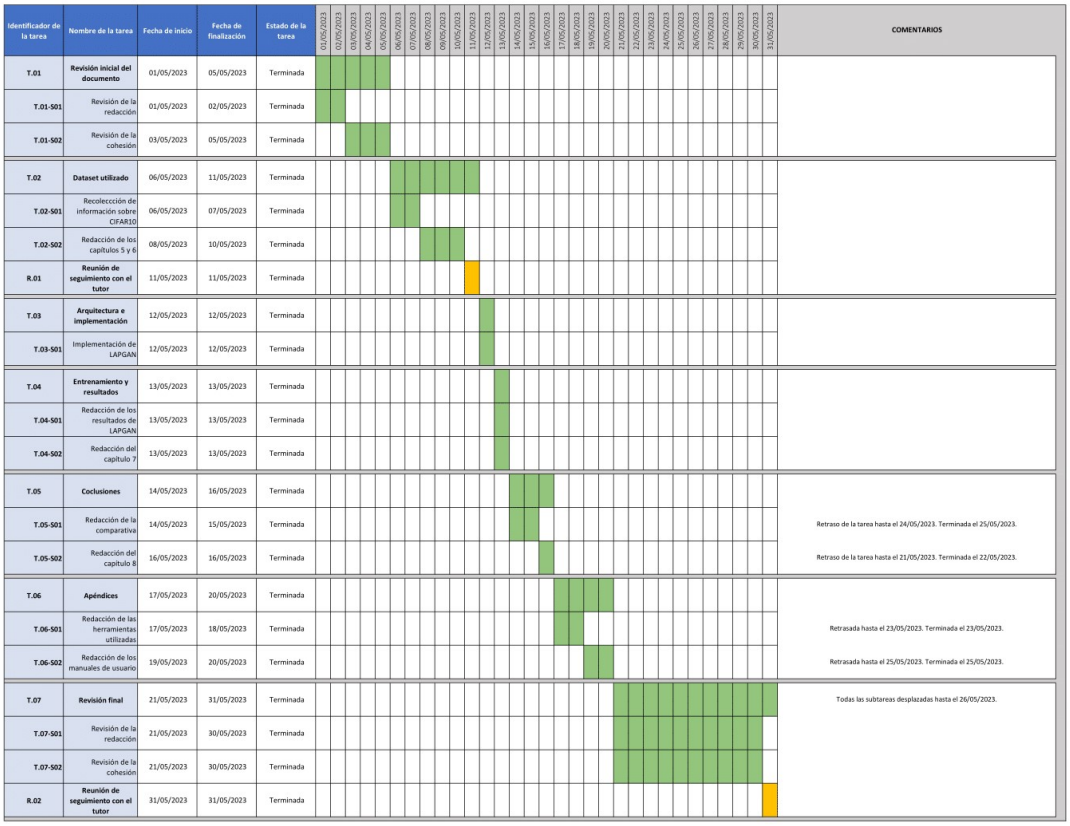


Figura 2.3: Diagrama de Gantt para el mes de Mayo

- Recolección de datos del entrenamiento de la *LAPGAN* implementada.

5. Conclusiones:

   - Redacción de la comparativa entre lo planificado y lo real.
   - Redacción de las secciones del capítulo 8.

6. Apéndices:

   - Redacción de las herramientas utilizadas.
   - Redacción de los manuales de usuario.

De nuevo, podemos ver la distribución temporal de las tareas asignadas a este mes en la figura 2.3.

Presentadas las tareas que se realizarán cada mes y los cronogramas a seguir, terminamos con esta sección de planificación temporal para dar paso al coste teórico del proyecto.

2.3. Presupuesto económico

La realización del presupuesto económico no ha sido una tarea sencilla, pues los trabajos de investigación raramente se ajustan a unos tiempos predeterminados debido a distintos factores como pueden ser las ideas que surgen a lo largo del proceso de recolección de información, según se van obteniendo más datos sobre el área en que se realiza el trabajo.

Dado que este proyecto es recopilatorio, con el objetivo de sentar las bases sobre el conocimiento actual, salvamos la dificultad de encontrar artículos e investigación en el área, pues ya se realizó en su mayoría y no parece estar sujeta a demasiados cambios.

Para realizar el coste económico a priori del trabajo calcularemos por separado el coste material y el coste humano, es decir, primero veremos que coste se asocia, sobre las 300 horas a priori de duración del proyecto, a materiales como puede ser un ordenador para el desarrollo, la infraestructura necesaria (*Internet*) o los costes del *software* asociados al desarrollo de las herramientas; y luego se calculará, según la asignación del rol de investigador, el coste que proviene de la utilización de recursos humanos, o lo que es lo mismo, la mano de obra.

Veamos primero cual es el coste estimado para los recursos materiales. Los recursos materiales que se utilizarán son:

- Ordenador Lenovo ideapad700 con coste (en el momento de su compra) de 1100€ aproximadamente y vida útil estimada de 6 años. Las especificaciones de este ordenador, importantes para el desarrollo del trabajo son:
  - Un procesador Intel Core i7-6700HQ CPU de 2.60GHz.
  - Una tarjeta gráfica NVIDIA GeForce GTX 950M, para el entrenamiento de las redes.
  - Dos discos duros; un HDD de 1TB y un SSD de 100GB.
  - 32 GB de memoria RAM.
  - Windows 10.

  Utilizando los meses útiles de vida de la computadora (72 meses) y los meses que se dedicarán al proyecto 2, obtenemos un uso del 2,78 %, lo que nos permite formular un coste a priori de 30,58€ por el uso de la computadora (a lo largo de Abril y Mayo).

- Conexión a la red (proveedor del servicio: Vodafone) de 600Mbps con coste 33,40€ por mes (pues no queremos pagar un año de permanencia). Como se utilizará la conexión durante 2 meses, el coste se eleva a 66,80€ por el uso en el tiempo de desarrollo.

- Herramientas *software open-source*, que son gratuitas y no incurrirán en gasto alguno, por lo que son añadidas al cálculo del presupuesto con coste de 0€.

Material	% uso	Coste (en €)
Ordenador portátil	2,78 %	30,58
Conexión a la Red	200 %	66,80
Anaconda Navigator		0
Spyder		0
Microsoft Office		0
Windows 10		0
TOTAL		97,38

Cuadro 2.1: Presupuesto dedicado al material utilizado

Reflejemos el presupuesto material en el cuadro 2.1 (sobre los dos meses de trabajo).

En el caso del presupuesto para recursos humanos, se ha decidido asignar el rol de investigador y desarrollador al alumno, pues son las tareas que se realizarán para la resolución de este proyecto. El rol de investigador abarcará la parte de búsqueda de información, así como el análisis y diseño de la memoria y las implementaciones. El rol de desarrollador se centra en las tareas de implementación y redacción, recayendo en el la mayor parte del trabajo.

Analicemos el coste asociado a contratar un profesional en cada uno de los roles que se van a desempeñar. El sueldo medio de un desarrollador o programador en España ronda los 25.900€ brutos anuales (referencia [14]). A su vez, un investigador cobra alrededor de 28.000€ anuales (referencia [13]).

Como las horas de un trabajo a jornada completa son 160 por mes, y se estiman necesarias 100 horas de implementación y 200 de investigación, calcularemos el presupuesto a pagar mediante el sueldo por hora de cada rol, y luego le añadiremos los costes de la Seguridad Social.

El sueldo por hora de un desarrollador será:

$$\frac{25900}{160 \times 12}€/h = \frac{25900}{1920}€/h = 13,4895€/h.$$

Y el sueldo por hora de un investigador será, realizando cálculos similares:

$$\frac{28000}{160 \times 12}€/h = \frac{28000}{1920}€/h = 14,5833€/h.$$

Además del gasto en pagar el sueldo bruto, es necesario calcular el gasto que se derivará en pagar la Seguridad Social de los empleados (ver [6]). Estos son:

- 23,60 % de contingencias comunes.
- 5,5 % tipo general de desempleo para un contrato indefinido.
- 0,20 % al Fondo de Garantía Social.

Rol	Coste (en €)	Coste de Seg. Social (en €)	Coste total (en €)
Investigador	2916,66	874,8	3791,46
Desarrollador	1348,95	404,685	1753,64
TOTAL			5545,1

Cuadro 2.2: Presupuesto dedicado a RRHH

Material	Coste (en €)
Coste material estimado	97,38
Coste humano estimado	5545,1
TOTAL	5642,48

Cuadro 2.3: Presupuesto total

- 0,70 % para formación profesional.

Que en total alcanzan un 30 % del sueldo bruto de la persona contratada. Sabiendo esto, habrá que añadir a los costes un 30 % del sueldo del desarrollador (404,685€), y el 30 % de las horas de trabajo del investigador (874,8€). Esto lo obtenemos de calcular:

$$13,4895€/h \times 100h \times 0,3\,\% = 404,685€.$$

Y:

$$14,5833€/h \times 200h \times 0,3\,\% = 874,8€.$$

Utilizamos una tabla para mostrar el presupuesto en recursos humanos para el proyecto (cuadro 2.2).

Conociendo ya la cantidad que gastaremos en material y recursos humanos, podemos calcular entonces un presupuesto conjunto que englobe los gastos estimados del proyecto. Para ello, únicamente necesitamos sumar el presupuesto material con el humano y obtendremos la suma total del gasto previsto. Visualicemos el coste estimado total en otra tabla (cuadro 2.3).

Concluimos entonces que el análisis del proyecto nos lleva a estimar un presupuesto de 5642,48€, juntando los posibles gastos, donde no se están contando los posibles días extra para solucionar problemas emergentes durante el desarrollo. Veremos el impacto de los problemas que han surgido durante el desarrollo del proyecto en la siguiente sección.

2.4. Ajuste a la realidad

Comenzaremos comparando el desempeño temporal real con la planificación realizada para los dos meses de trabajo.

Lo primero será mencionar que en los diagramas de Gantt se tiene un apartado de comentarios, que han sido los siguientes para el mes de Abril:

- Para la primera reunión con el tutor, y debido a problemas para poderse conectar en la fecha especificada (10/04/2023), se desplazó la fecha de reunión al 13/04/2023.
- Una vez terminada la redacción del apartado 3.2, el alumno se dió cuenta de la necesidad de introducir las técnicas de regularización más comunes en el *Deep Learning*. Esto produjo que fuera necesaria una revisión del apartado para añadir dichas técnicas, y se concluyó su redacción el 25/04/2023.
- En el caso de la segunda reunión con el tutor, fue necesario adelantar un día, lo que hizo que se produjera el 27/04/2023, en vez del 28/04/2023.
- Como preparación, los modelos de las redes *DCGAN* y *CGAN* se cambiaron de fechas, la primera se implementó (pero no se entrenó) el 18/04/2023, y la segunda tuvo un retraso en su implementación hasta el 28/04/2023. La red *DCGAN* se entrenó de acuerdo al cronograma, y la *CGAN* el mismo día de su implementación, el 28/04/2023. No fue hasta el mes de Mayo que se implementó la red *LAPGAN*.

Debido a eventos familiares del alumno el 13/05/2023 y el 20/05/2023 se retrasaron varias tareas de redacción del cronograma de Mayo:

- Se ha retrasado varios días la redacción de la comparativa, a realizar según el cronograma el 14 y 15 de Mayo, pero realizadas finalmente el 24 y 25.
- Por otra parte, también hubo que retrasar los tiempos de redacción de las conclusiones, que se realizaron antes que la comparativa, en los días 21 y 22 de Mayo, seis días después de lo planificado.
- También se retrasaron las tareas de redacción de referentes a las herramientas utilizadas y los manuales de usuario hasta el 23/05/2023 y 25/05/2023, respectivamente. Originalmente estas tareas se habrían terminado el 18 y el 20 de Mayo.

El retraso de estas tareas produjo un desplazamiento en la revisión final del documento, que no se produjo hasta el 26/05/2023.

Esto demuestra que el cronograma, pese a que no se ha podido cumplir en su totalidad, acertó con el tiempo necesario para la mayoría de las tareas, así como sirvió de guía durante el desarrollo del proyecto.

Comparemos ahora el gasto real con el presupuesto de la sección 2.3. Como se perdieron dos sábados el mes de Mayo, se intentaron compensar las 16 horas que se habrían empleado esos días en la siguiente semana respecto de cada uno de ellos. De este modo, se asignaron 1 hora y media extra de trabajo a cada uno de los días de esas semanas, siendo un total de $14 \times 1{,}5 = 21$ horas realizadas entre las dos semanas. Con esto, se conseguía paliar el efecto de pérdida de esos días festivos, pero que se reflejaron en 5 horas de más (pues sólo eran necesarias 16 horas extra). Esto ha de reflejarse entonces en el coste humano, pues contaría como un día más de jornada (a efectos del trabajo).

Dado que las 5 horas extra se dedicaron a la redacción y desarrollo, el coste añadido al presupuesto dado serían 5 horas más de trabajo de desarrollador que hay que pagar. De este modo, el coste real humano del proyecto sigue la tabla 2.4

Rol	Coste (en €)	Coste de Seg. Social (en €)	Coste total (en €)
Investigador	2916,66	874,8	3791,46
Desarrollador	1348,95	404,685	1753,64
Horas extra	67,45	20,23	87,68
TOTAL			5632,78

Cuadro 2.4: Coste en RRHH real

Lo que es, como señala la tabla, un coste humano de 87,68€ más caro que lo planteado inicialmente.

Dado que el coste material sobre el ordenador es de aproximadamente 5 horas extra de lo planteado, y no ha sido necesario contratar un mes más de conexión a la Red, no se considerarán gastos materiales extra, pues su influencia en el gasto real, aunque existente, es mínima.

La conclusión que se puede obtener de esta comparativa es que, pese a las interrupciones breves en el desarrollo, el trabajo se ha podido realizar en tiempo y costes bastante cercanos al presupuesto y planificación temporal realizados.

Capítulo 3

Conceptos técnicos previos

Ya hemos visto una pequeña introducción al problema, y también como se ha planificado y desempeñado el desarrollo del trabajo. El proyecto continúa con este capítulo, centrado en dar los conceptos básicos con los que se va a trabajar, ya sirvan de repaso para quien ya los conozca o de introducción a las redes generativas y la IA para quien los desconozca.

3.1. Bases del estado del arte

Pese a que nuestro objetivo principal es introducir las redes generativas sin llegar a los modelos más complejos, hemos considerado que es importante realizar una sección para repasar el estado actual de la tecnología en este área.

En estos últimos 9 años, las redes generativas de contenido han tenido un desarrollo floreciente y mejoras sustanciales, lo que las ha puesto de moda recientemente. Sin embargo, los productos que se presentan al público no están conformados por sólo la red generadora, si no que además suelen disponer de una interfaz web para aumentar la comodidad a la hora de utilizarlas, además de que no siempre son una única red o su arquitectura es más compleja de lo que se presentará en este proyecto. Dentro de las herramientas más modernas podemos encontrar *Bard*, *Bing Chat*, *Chat-GPT*, *Flamingo* y *DALL-E 2*, por ejemplo. Todas estas herramientas mantienen el denominador común del uso de variaciones de *GAN*, *LLM* o *Transformers* para implementar sus modelos, además de utilizar el llamado paradigma multimodal.

Si tuviéramos que explicar la multimodalidad brevemente, haríamos el siguiente símil: nosotros como humanos poseemos cinco sentidos; la vista, el oído, el tacto, el olfato y el gusto. En comparación con los ordenadores, que la mayoría de veces centran la atención de los agentes inteligentes en una sola tarea, llevamos mucha ventaja. La multimodalidad pretende saltar esa barrera, armonizando la comprensión del texto escrito y el habla con el reconocimiento de imágenes, pues actualmente las máquinas no pueden captar olores ni sabores, y menos utilizar el tacto. De este modo, abarcamos dos sentidos de los cinco, la vista y el oído, y en ocasiones podemos conseguir respuestas orgánicas del sistema inteligente.

Esta multimodalidad es la que dota a herramientas como *Flamingo* o *DALL-E 2* de un funcionamiento aparentemente completo: entienden lo que pedimos mediante un comando de texto y lo traducen a la producción de una imagen.

Dado que ya hablaremos de las *GAN* más adelante en el trabajo, introduzcamos el concepto de *Transformer*. Los *Transformer* nacen como solución a los problemas de *NLP* (*Natural Language Processing*), y como corrección al uso abusivo de redes recurrentes y convolucionales para la traducción automática de forma secuencial.

Generalmente, los problemas de traducción se resolvían, antes de esta arquitectura, con una arquitectura de codificador-decodificador basada en el uso de redes recurrentes y convolucionales. El uso de estas redes aumenta mucho el coste computacional según aumente la dimensión de los datos de entrada. En 2017, cuando se originan los *Transformer* [42], se utilizaban ya mecanismos de atención junto con las redes recurrentes para poder completar las tareas. Estos mecanismos aumentaban la calidad y la capacidad de computación paralela, pero en esencia las redes mantenían el mismo problema con la dimensionalidad. A la hora de atacar estos problemas de tiempo de entrenamiento, se recurría a trucos de factorización [21] y a computación condicional [37]. Aunque se redujese el impacto del problema, esta reducción no era lo suficientemente fuerte, por lo que se planteó una nueva arquitectura.

Transformer fue una novedosa arquitectura que, estudiando modelos contemporáneos como los anteriores [38, 4, 9], lograba abandonar la arquitectura recurrente y convolucional para centrarse únicamente en los métodos de atención. De este modo, se conseguían tiempos de entrenamiento menores y se atacaba de lleno el problema de dimensionalidad, reduciendo su impacto. Como bien se puede leer en [42], estas redes resultaron ser una solución pionera en traducción, consiguiendo buenos resultados con menor tiempo de entrenamiento.

Esta forma de estructurar las redes se ha llevado a la práctica en modelos como *GPT-3* y, más recientemente, *GPT-4* [28], del cual no se dispone de mucha información.

Las nuevas tecnologías de las que hemos estado hablando, que se modifican y estudian más a cada día que pasa, no sólo tienen su base en las *GAN*. El auge de los problemas de *NLP* llevó también a la implementación de los llamados *Large Language Models* o *LLM*, que también han jugado un papel crucial en los últimos avances científicos respecto a la comprensión textual por parte de las redes profundas.

El primer *LLM* fue presentado en [29]. Aunque fundamentalmente su desarrollo se basa en el uso de la arquitectura *Transformer*, se clasifica como gran modelo del lenguaje por el conjunto de datos de entrenamiento y su aplicación a distintas tareas dentro del procesamiento del lenguaje natural. Veamos en más detalle qué lo diferencia de los *Transformer*.

La comprensión del lenguaje natural comprende un gran rango de tareas diversas, que varían desde la capacidad de responder a preguntas hasta la clasificación automática de documentos completos. En [29] se demostró que se podían obtener mejoras significativas respecto a los desarrollos hasta el momento de la misma manera que antes, utilizando los *Transformer*, aunque en este caso el proceso se diferencia en el tipo de aprendizaje. El primer modelo *GPT* (*Generative Pre-Trained Transformer*) sigue una estrategia de

aprendizaje mixto de la siguiente manera; primero se trabajaría con textos no etiquetados, de modo que la red aprenda de forma no supervisada, y a continuación se realizaría un ajuste de pesos discriminativo que ajustaría el modelo a la tarea específica que se quisiera resolver. Es durante esta segunda etapa donde vemos los principales cambios, pues *GPT* utiliza transformaciones dependientes de la tarea para la que se va a dedicar. Estas transformaciones se utilizan sobre las entradas y permiten que, como el modelo de base está entrenado, se puedan conseguir resultados efectivos en distintas tareas con cambios mínimos en el modelo.

El objetivo de la red *GPT* era resolver dos problemas:

1. No se tenía claro el tipo de optimización a seguir a la hora de aprender representaciones textuales útiles para la transferencia de conocimientos.

2. No existía un consenso en la forma más efectiva de transmitir las representaciones aprendidas a la resolución de la tarea objetivo.

Estudios cercanos a este [32, 25, 18] buscaban varios objetivos para dar respuesta a estas preguntas, pero se superaban entre sí en cada tarea para la que eran diseñados específicamente, además de no tener uniformidad, ni en el método, ni en la optimización.

GPT pretendía dar una respuesta universal a estas preguntas, utilizando un objetivo de modelización sobre datos no etiquetados para obtener los parámetros iniciales de la red, y adaptando estos a las tareas objetivo. El modelo proveía de lo necesario para tener una memoria estructurada y manejar las dependencias entre palabras, evitando, gracias a los *Transformer*, una estructura recurrente o convolucional. Esta red revolucionó la manera de afrontar problemas de *NLP*, pues cuando se realizaron sus evaluaciones, se obtuvieron mejoras en 9 de 12 campos de aplicación, como por ejemplo:

- Inferencia sobre el lenguaje natural.

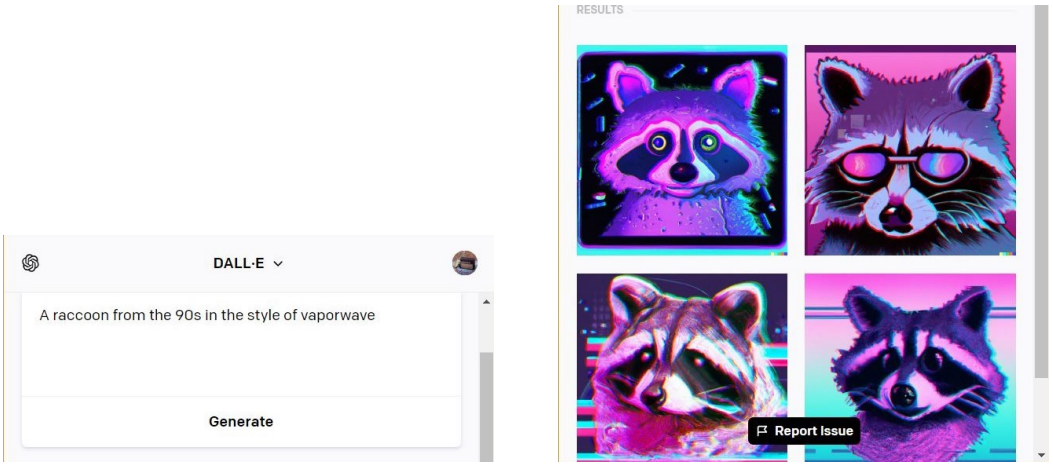
- Resolución de preguntas.

- Clasificación de textos.

El desarrollo de esta red, que ha dado pie a cuatro mejoras de la misma, introdujo un marco de trabajo para obtener una fuerte comprensión del lenguaje natural, además de presentar un modelo aparentemente multitarea, basado en el preentrenamiento generativo y el ajuste de pesos discriminativo.

Recordemos entonces que este tipo de modelos, y las *GAN*, son los modelos en los que se basan la mayoría de aplicaciones comerciales de generación de contenido, ya sea porque implementan solo una de estas redes, o varias a la vez, de modo que las salidas y entradas se complementen entre sí. Podemos ver un ejemplo de esto en las figuras 3.1 y 3.2, que muestra como podemos generar imágenes a partir de una petición basada en lenguaje natural en inglés.

Esta combinación de varios modelos para obtener una salida orquestada es a lo que nos referiremos como multimodalidad o paradigma multimodal. Aunque ya se ha introducido mediante una metáfora, expliquemos mejor en que consiste este paradigma. Para poder



(a) Texto para generar los ejemplos (b) Ejemplos generados

Figura 3.1: Ejemplos producidos por *DALL-E 2*

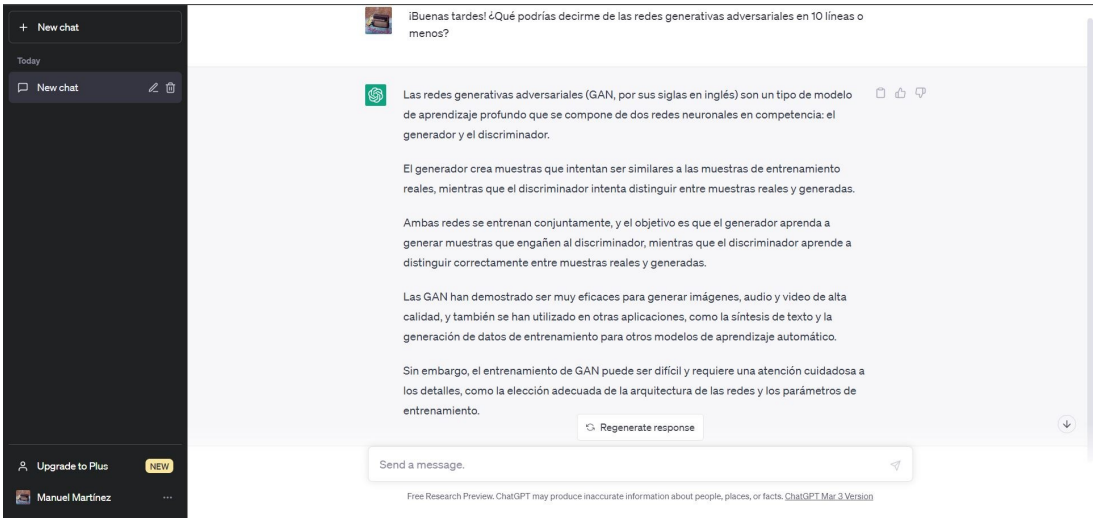


Figura 3.2: Ejemplo generado con *ChatGPT*

ejemplificar mejor su funcionamiento nos centraremos en los modelos de visión-lenguaje (*Vision-Language Models* o *VLM*) que son aquellos que combinan los modelos anteriores: modelos que generan imágenes y se dedican al *NLP* a la vez.

En el artículo [35], que presentaba *CLIP*, comenzamos a ver las primeras aplicaciones de los *VLM*. *CLIP* es una red que intenta armonizar el preentrenamiento de las redes con la multimodalidad para imágenes y texto. Esta red nació en respuesta a problemas de la visión computacional, generalmente teniendo que ver con la falta de generalización de estos modelos, ya que se trabajaba con datos etiquetados y para aprender nuevos ejemplos había que entrenar con una nueva etiqueta. La solución parecía sencilla: si se consiguieran aprender las cualidades de las imágenes desde texto plano, estas serían mucho más sencillas de transferir entre redes, facilitando así el proceso de aprendizaje de nuevos

objetos, y obteniendo mayor información sobre el conjunto de entrenamiento.

La idea principal de esta red se basa en entrenar el modelo previamente, para que empareje imágenes con descripciones de las mismas, de modo que se obtuviera un algoritmo eficiente y escalable que permitiera aprender representaciones más flexibles.

Aunque no fracasó estrepitosamente, las conclusiones que se obtuvieron fueron escasas. El modelo parecía funcionar y ser competitivo con los algoritmos existentes de visión computacional, pero requería de mucha capacidad computacional. También parecía que a suficiente escala era un acercamiento mucho más veloz y flexible que sus competidores, pero no se obtuvieron los resultados rompedores que se esperaban.

Sin embargo, era el primer paso en una buena dirección. Podemos ver el funcionamiento de tecnologías que se acercan más al objetivo de lo que pretendía esta red; como lo son *Stable Diffusion*, *Midjourney* o *DALL-E 2*. Estos ejemplos muestran una forma nueva de interactuar con la tecnología, pues combinan la generación de imágenes con la comprensión lectora, un ejemplo de multimodalidad.

En resumen, la multimodalidad no se refiere únicamente a este área de imágenes y procesamiento de lenguaje natural, si no más bien a la combinación de modelos y redes para conseguir un propósito, utilizando técnicas para armonizar la salida y así abarcar varias áreas de distintos modelos al mismo tiempo. Con este paradigma, se pretende asemejar más a nuestros autómatas al funcionamiento del aprendizaje humano, que no se centra únicamente en la utilización de un sentido, si no que se manejan varios para obtener información del entorno.

Con estos avances queda planteada la cuestión: ¿cuándo llegarán los modelos de inteligencia artificial de propósito general? Estos modelos, comúnmente llamados *AGI* tienen como objetivo dar la capacidad general de razonamiento a las IAs, de modo que sean capaces de resolver problemas muy diferentes, asimilando así el comportamiento de un humano, que puede por ejemplo hacer una tortilla y saltar a la comba, dos actividades poco relacionadas entre sí. La multimodalidad es una buena forma de avanzar en la investigación para obtener *AGI*, aunque aún estamos lejos de avances tan notorios, pues no sólo depende del campo de la computación, si no que también son necesarios avances en robótica.

Podemos concluir que pese a que el estado actual de las tecnologías da resultados más potentes de los que observaremos en el trabajo, no son más que mejoras sofisticadas sobre los conceptos que se presentan en él. Abordemos, entonces, una serie de conceptos previos para la buena comprensión del trabajo, para así poder introducir y explicar la teoría detrás de las redes generativas adversariales y los primeros modelos que surgieron.

3.2. Conceptos básicos

Ahora que ya conocemos cual es el estado de la tecnología que vamos a estudiar, introduzcamos una serie de conceptos básicos sobre inteligencia artificial para poder tratar modelos más complejos. Para ello, realicemos un repaso de conceptos como *Machine* y *Deep Learning* y de las redes neuronales. También veremos los dos tipos de aprendizaje

automático y dentro de cual de ellos se encuentran los modelos con los que vamos a trabajar, además de las técnicas de regularización más utilizadas en el campo del *DL*.

3.2.1. *Machine Learning*

El aprendizaje automático, *Machine Learning*, o simplemente *ML*, es una de las ramas de la IA. Esta rama se centra en desarrollar algoritmos que resuelvan problemas y tareas basados en la experiencia que se puede obtener de los datos iniciales, es decir, algoritmos que obtienen conocimiento a partir de una serie de entradas y mejoran su toma de decisiones de forma iterativa en base a la información de dichas entradas. El objetivo final de esta disciplina es simular de alguna manera el funcionamiento del cerebro humano cuando aprende conceptos y reglas nuevas, y reproducirlo en el terreno de las máquinas.

A la mejora iterativa en la toma de decisiones lo llamamos entrenamiento del algoritmo, y, cuando termina este proceso, nos permite dar respuestas a conjuntos de datos nuevos, gracias a extraer información generalizada de los datos iniciales. Matemáticamente esto se representa por el ajuste de pesos en matrices y funciones no lineales, que transforman un objeto (generalmente un vector) en otro distinto.

Una vez entrenado el algoritmo se obtiene una salida a partir de las entradas dadas, lo que nos lleva a medir la distancia entre lo obtenido por la ejecución del algoritmo y lo que esperábamos obtener. Esto lo hacemos mediante una función llamada función de pérdida que mide dicha distancia, y nos devuelve un valor que se interpreta para entender como de lejos está la salida de los objetivos deseados. La reducción del valor de esta función suele considerarse buena, y se procura que durante el entrenamiento se reduzca iterativamente el valor que generan las salidas con la función de pérdida.

Cuando consideramos que el valor de la función de pérdida es adecuado, obtenemos un modelo que generaliza los datos y puede realizar tareas relacionadas con ellos. Estos modelos son muy útiles cuando no podemos codificar las reglas o existe una alta complejidad para la resolución del problema. Sin embargo, no siempre es óptimo utilizar un algoritmo y modelo de *ML* pues el entrenamiento requiere, en ocasiones, altos tiempos de computación. Hay ocasiones en las que es mejor implementar algoritmos tradicionales, basados en programación estructurada, que resuelvan un problema concreto más rápido que los algoritmos de *ML*.

A continuación abordemos los dos tipos principales de aprendizaje dentro de los algoritmos de aprendizaje automático, que son el aprendizaje supervisado y no supervisado.

3.2.2. Tipos de aprendizaje

El aprendizaje de los algoritmos, tanto de *ML*, como *DL* puede seguir uno de los dos siguientes enfoques: si conocemos los resultados y las etiquetas esperadas para un conjunto de datos, decimos que el aprendizaje es supervisado, por el contrario, si no trabajamos con datos etiquetados hablamos de aprendizaje no supervisado.

Comenzaremos tratando el enfoque supervisado del aprendizaje automático. Como ya hemos dicho, partiremos de un conjunto de datos que se quieren etiquetar y para los

que conocemos las respuestas a los ejemplos iniciales. Como conocemos las entradas y las salidas correctas para estas entradas, nuestro objetivo es que, al realizar el entrenamiento, podamos realizar predicciones sobre nuevos datos, para su clasificación o regresión. Entre algunos ejemplos de aplicación de este tipo de aprendizaje tenemos la regresión lineal, las redes neuronales o las *SVM*.

También tenemos el enfoque no supervisado, con el que no manejamos datos etiquetados, por lo que no conocemos los resultados para los datos de ejemplo. En general este enfoque es utilizado para la búsqueda de patrones y asociaciones entre ellos, y, en particular, para la generación de elementos y contenido mediante algoritmos de *Deep Learning*. Ejemplos de este tipo de aprendizaje son las técnicas de clustering y reducción de dimensiones. En lo que refiere a este trabajo, dentro de este marco de aprendizaje tenemos las *GAN* y los *VAE*, dos de las primeras redes generativas implementadas y estudiadas, ahora precursoras de los algoritmos generativos punteros.

3.2.3. Redes neuronales

Antes de repasar el concepto de aprendizaje profundo o *Deep Learning* hemos de introducir las redes neuronales, pues son la premisa de las redes profundas de *ML*.

El objetivo de toda red neuronal es emular la sinapsis que ocurre en el cerebro humano cuando aprende o recuerda conceptos, lo que, traducido a nuestro lenguaje sobre los datos, significa que las redes neuronales se centran en extraer y reconocer patrones de los datos para poder generalizar información y predecir resultados.

Si tuviéramos que hacer una lista de los objetivos de la computación neuronal, un buen ejemplo sería el de las lecciones de [40]. Estos objetivos serían los siguientes:

- Entender el funcionamiento del cerebro, pues es muy frágil para un estudio en directo, mediante simulaciones por ordenador.
- Entender como trabajan con la información las neuronas, su trabajo en paralelo y la adaptabilidad de sus conexiones.
- Resolver problemas prácticos utilizando algoritmos inspirados en el cerebro.

También conviene notar que no todos los algoritmos se inspiran en el cerebro, y algunos de estos, por muy diferentes que sean, son muy útiles a la hora de trabajar. Un ejemplo de algoritmo que no se basa en el funcionamiento biológico de las neuronas son las *GAN*, como veremos más adelante con el ejemplo del falsificador y el tasador.

Conocidos los objetivos de las RRNN, veamos como funcionan las neuronas de nuestro cerebro, para luego detallar como se ha llevado esto a las redes neuronales.

Primero veamos la estructura de una neurona. Una neurona cuenta con las siguientes partes principales:

- Cuerpo: Es la parte principal de la neurona, con sus componentes para el buen funcionamiento y la vida de la misma.

- Axón: Es una de las conexiones de la neurona. Se ramifica para conectarse al resto de las neuronas y así poder enviar información al resto.
- Cono axónico: Es el punto de conexión entre el cuerpo neuronal y el axón. Controla los picos de energía.
- Dendritas: Son ramificaciones que salen del cuerpo, y su trabajo es recibir información de otras neuronas.

Las neuronas se comunican entre sí mediante un proceso que llamamos sinapsis neuronal, que principalmente se basa en la transmisión de energía entre neuronas mediante neurotransmisores. Estos neurotransmisores interactúan con las neuronas cambiando su forma, pues se unen con la membrana, lo que genera el aprendizaje. Cuando la energía transmitida por una neurona supera un umbral en la siguiente, la segunda recibirá el mensaje e intentará transmitirlo al resto de neuronas sucesivas.

La sinapsis funciona mediante picos de energía en el axón, que pasan los neurotransmisores hacia el resto de las neuronas conectadas. Para generar dicho pico de energía, el cono almacena carga y luego la pasa como sinapsis, siempre que la energía recibida supere el umbral necesario para la transmisión.

Y, ¿por qué queremos replicar este proceso en nuestras máquinas? Es sencillo, aunque el proceso de la sinapsis se produce a pequeña escala, con una gran cantidad de neuronas se pueden transmitir y almacenar grandes cantidades de datos. Además, durante el proceso de transmisión de información se consume muy poca energía y se genera una adaptación local por parte de las neuronas, es decir, no se modifica toda la información del cerebro cuando se reciben nuevos estímulos y datos. Entonces, dependiendo de las señales recibidas por las neuronas conectadas, podemos dividir el cerebro según distintas funcionalidades.

Entendiendo ya como son las neuronas, veamos el funcionamiento del cerebro en su conjunto. Cada neurona del cerebro está conectada a otras distintas, y recibe entradas de ellas, al igual que las transmite. Algunas de estas neuronas están conectadas a receptores nerviosos y son las que reciben señales del exterior, mientras que otras son neuronas corticales, que utilizan picos de energía para transmitir información.

El efecto de cada entrada que recibe una neurona se controla mediante pesos sinápticos, es decir, se asocia una importancia a cada entrada que llega. Estos pesos se adaptan según vamos aprendiendo conceptos, de modo que el cerebro aprenda conceptos que consideremos útiles. En nuestro cerebro hay aproximadamente 10^{11} neuronas con 10^4 pesos cada una y muchos pesos afectan a la vez a computaciones que realizamos en poco tiempo, porque gran cantidad de neuronas son utilizadas en cada proceso que consideramos natural (como puede ser andar por la calle).

En general, el cortex cerebral es muy similar y está formado por neuronas de propósito general que se transforman en neuronas de propósito específico como respuesta a la experiencia. Así pues, hay distintos módulos en el cerebro, es decir, distintas partes del cerebro realizan distintas acciones, y acciones específicas aumentan el flujo sanguíneo en zonas localizadas del cerebro. Esto nos brinda una buena computación paralela y mucha flexibilidad, que es lo que buscaremos también en las redes que programemos.

Saltemos al proceso artificial. Para fabricar nuestra red neuronal, definamos primero como es una única neurona artificial, y a partir de eso expliquemos la red completa. Cuando consideramos una neurona artificial, tenemos en cuenta los siguientes elementos:

- Un conjunto de entradas, generalmente en forma de vector: $\mathbf{x}$. Supondremos que este vector está formado de n componentes.
- Un conjunto de pesos, relacionados con cada componente de la entrada, que denotaremos por $\omega_1, \omega_2, ..., \omega_n$.
- Un sesgo, ω_0, que funcionará como un peso más, asociado a una entrada, generalmente negativa, que sirve como contrapeso. Este sesgo representa la predisposición al resultado de la neurona, y su objetivo es simplificar los cálculos que hay que realizar para saber si se transmite la información a las siguientes neuronas.
- Una regla de propagación, que principalmente opera pesos y entradas y obtiene un resultado. Distintas reglas de propagación dan lugar a distintos tipos de neurona.
- Una función de activación, que produce la salida y controla los resultados. También distintas funciones de activación dan lugar a distintas neuronas. Entre las funciones más utilizadas tenemos la *Rectified Linear Unit* ($ReLU$), la tangente hiperbólica o la sigmoide.
- Una salida, $\mathbf{y}$, que es lo que produce la neurona tras su activación. En ocasiones, la salida generada es un número entre 0 y 1, representando una probabilidad, aunque si se tienen distintas alternativas para la salida, esta se suele representar con un vector.

Es decir, la neurona es una función que se puede representar mediante una combinación lineal de las entradas y el sesgo, sobre los cuales aplicamos la función de activación, lo cual produce una salida. Cuando conectamos las salidas de distintas neuronas en forma de red, obtenemos lo que se llama una red neuronal. Esto se representa matemáticamente con la composición de funciones.

Esta conexión de neuronas, una tras otra, y la red, no son más que una función que transforma el vector $\mathbf{x}$ en $\mathbf{y}$. Para poder definir la red de manera correcta, las neuronas se distribuyen por capas conectadas entre sí; la primera capa es por la que entrarán los datos, luego se pasa por una serie de capas intermedias hasta llegar a la última, que genera la salida.

Se puede observar este proceso en la figura 3.3, con las relaciones entre neuronas (círculos) cuando se conectan todas entre sí (líneas). Las líneas a puntos representan la posibilidad de más capas intermedias. También se puede observar la agrupación por capas de una RN, separadas por líneas discontinuas. En este ejemplo se han conectado todas las neuronas con el resto de las de la siguiente capa.

Veamos de que modo afecta el entrenamiento a la red neuronal. Durante el proceso de entrenamiento, vamos modificando los pesos de las neuronas para obtener salidas cada vez

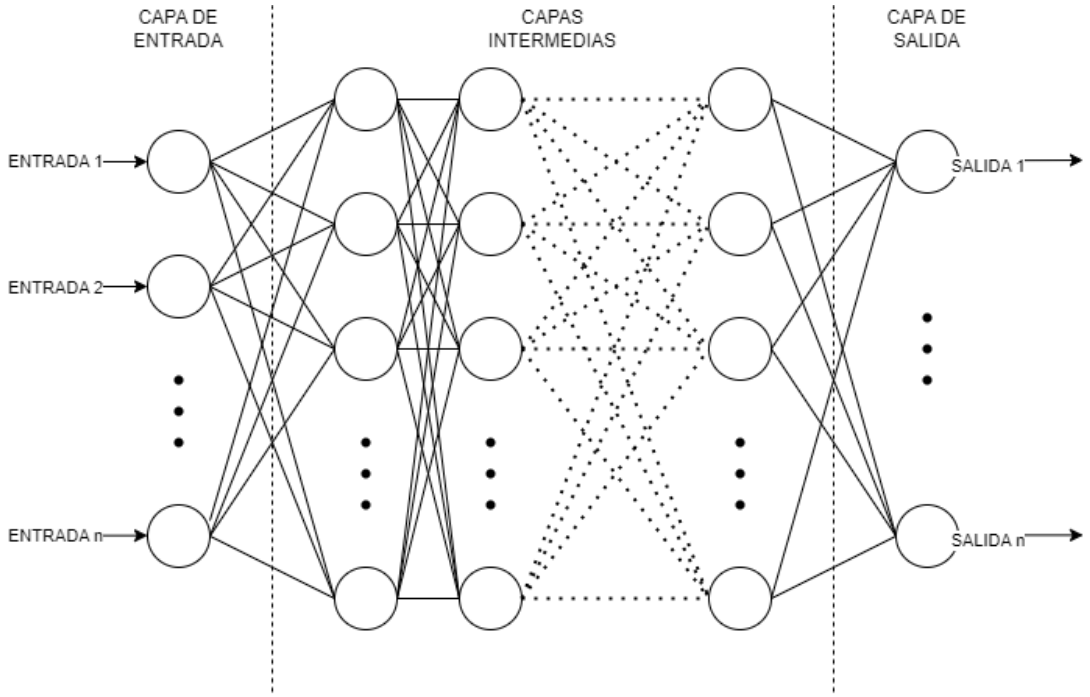


Figura 3.3: Diagrama de arquitectura de una RN

más cercanas a los objetivos que deseemos en base a la función de perdida que se aplica sobre el resultado de toda la red. Sigamos los pasos del entrenamiento y aprendizaje de las redes neuronales, ya que sabemos como afectan estos a la red una vez construida:

1. Inicializamos los pesos y el sesgo de las neuronas de la red, y preparamos los datos para el entrenamiento.

2. Pasamos los datos por la red y calculamos el error con la función de pérdida.

3. Utilizamos un optimizador para actualizar los parámetros de la red, de modo que se puedan obtener mejores resultados, es decir, optimizamos el valor de la función de pérdida.

4. Repetimos los pasos 2 y 3 hasta alcanzar una condición de parada que hayamos preestablecido. A cada iteración de los pasos anteriores lo llamaremos época.

Uno de los peligros del anterior entrenamiento es pensar que la condición de parada ha de ser el momento en que todos los ejemplos de los datos iniciales se clasifiquen correctamente. Esta clasificación correcta permite aprender sobre el conjunto de datos pero restringe la generalización de los datos, lo que se conoce como sobreajuste.

Los algoritmos de aprendizaje automático más sencillos tienen RRNN de una única capa intermedia. Cuando se mencionan las capas ocultas de una red, nos referimos a las capas intermedias que posee, pues son las que no producen resultados "visibles".

3.2.4. *Deep Learning*

Llegados a este momento, lo normal es preguntarnos: ¿qué ocurre con las RRNN con más de una capa intermedia? Cuando tenemos RRNN de estas características, hablamos de redes profundas. Son estas redes las que conforman el alcance del aprendizaje profundo o *Deep Learning* (*DL*).

El *DL* es un campo del *ML* centrado en los patrones complejos de datos mediante el uso de redes neuronales profundas. Su principal objetivo es extraer información de grandes volúmenes de datos, lo cual requiere generalmente altos tiempos de ejecución, donde la red aprende en cada capa. El diagrama 3.3 nos muestra, en realidad, una red profunda, pues tenemos más de una capa oculta, y muestra como es la arquitectura general de una RN profunda.

Actualmente, estas redes, tienen un gran protagonismo, pues las redes generativas, el reconocimiento del habla, o los grandes modelos del lenguaje son ejemplos de ellas muy utilizados hoy en día.

Dado que las redes neuronales profundas tienen muchas capas y neuronas, se genera el siguiente problema: la red se adapta mejor al conjunto de entrenamiento. Aunque a priori pueda parecer una buena cualidad para las redes que estamos intentando construir, a posteriori genera sobreajuste sobre el conjunto de entrenamiento. Para evitar este potencial sobreajuste, se pueden seguir algunas de las llamadas técnicas de regularización:

- Parada temprana: esta técnica sigue una serie de pasos para evitar que la red entrene más de lo necesario. Partiendo del conjunto de entrenamiento, lo dividiremos en dos, un conjunto de entrenamiento más pequeño y un conjunto de validación. Con este paso realizado, podemos monitorizar el comportamiento de la red con el conjunto de validación, de modo que, cuando se aplique el optimizador de la red, se detenga el entrenamiento en el momento que observemos aumento en el error para la validación.

- Normalización de lotes: esta estrategia tiene como objetivo normalizar la entrada de las funciones de activación, lo cual no es más que llevar las entradas a media 0 y varianza 1. Hay que tener en cuenta que esta estrategia no se debe aplicar siempre, pues depende de la función de activación escogida. Por ejemplo, si la activación es una *ReLU*, entonces las entradas muy grandes tendrán gradiente 1 y las muy pequeñas gradiente 0, luego es adecuado realizar la normalización para obtener mejores resultados. Otro ejemplo serían las funciones de tipo sigmoide, dado que, al normalizar, llevamos los valores muy altos o bajos (que tendrían gradiente 0) hacia el 0, obteniendo gradientes más altos. Para poder llevar a cabo esta técnica se aprenden dos parámetros para la normalización, la media y la varianza, y otros dos para el reescalado de los resultados tras la normalización.

- *Dropout*: la simulación de muchas *ANN* (redes neuronales artificiales) van de la mano de esta técnica. El método de aplicación consiste en lo siguiente; se escogen una serie de neuronas, normalmente un porcentaje del número total de neuronas, y se ocultan o desactivan. Estas neuronas cambian según la iteración del optimizador

en la que estemos, y su objetivo es evitar que la red se centre únicamente en aprender una característica, lo que lleva a un aprendizaje de representaciones más generales.

- *Data augmentation*: Idealmente, querríamos que el aprendizaje que realiza la red neuronal fuera *online*, pues la cantidad de datos en línea para el entrenamiento es virtualmente infinita. Sin embargo, este acercamiento al entrenamiento no es muy realista, ya sea por los tiempos de ejecución, el aprendizaje en áreas que no sean de interés u otros factores. La solución la presentan estas técnicas, que nos permiten aumentar el número de datos de que disponemos, manipulando levemente los elementos del conjunto de entrenamiento. Algunas manipulaciones pueden ser el añadido de ruido, el reflejado de parte de ellos, el desplazamiento... Generalmente esta estrategia se utiliza en conjuntos de imágenes, y como resultado tendremos datos más informativos y una mejora de la generalización de la red.

- Aprendizaje por transferencia o *transfer learning*: este tipo de aprendizaje se puede dar por falta de recursos de computación o por no disponer de un conjunto de datos suficientemente grande, entre otros problemas. El objetivo de esta técnica es el siguiente; se toma una red preentrenada, ya sea mediante la descarga de un modelo ya entrenado o por un entrenamiento anterior ya realizado. Tras obtener este modelo, utilizamos el optimizador con nuestro *dataset* y la nueva red, ajustándola eficazmente a nuestras necesidades. El nombre de la estrategia viene dado porque lo que estamos realizando es la transferencia de información de un problema similar al que tenemos a nuestro caso particular.

Además, también hay otro problema, referente a la dimensionalidad de los datos. Las redes profundas tienen más neuronas y más capas, lo que implica mayores tiempos de ejecución, sobre todo durante la fase de entrenamiento. Esto se debe a la gran cantidad de parámetros que han de ser actualizados en cada época, lo cual no se simplifica si los datos tienen una dimensión muy alta, como pueden ser las imágenes o la voz.

Para poder abarcar datos complejos (imágenes, audio o incluso vídeo, cuya dimensión es alta) es necesario utilizar técnicas de espacio latente para los modelos profundos, que permiten simplificar la información de los datos mediante la reducción a espacios de dimensión menor. El espacio latente es un espacio de dimensión más pequeña que el original (preferiblemente mucho menor) que busca la abstracción de las características más importantes de los datos, para así poder extraer información de manera más sencilla. Esta información extraída es la que permite a estos modelos generar, adecuadamente, contenido que sigue el estilo de los ejemplos, pero añadiendo modificaciones nuevas sobre los mismos, y asegurando cierta variabilidad sobre el resultado, pues al reducir dimensión perdemos parte de la información inicial. El uso de técnicas de espacio latente ha probado ser también una buena práctica a la hora de evitar el sobreajuste, además de simplificar el entrenamiento de las redes profundas, pues obviando ciertos matices de las entradas, generalizamos aquellos que se consideren más importantes.

3.2.5. Redes convolucionales

Vamos a centrarnos en un tipo especial de redes profundas, las redes de convolución o *CNN*; uno de los principales ladrillos para las redes profundas y muy importantes para trabajar con datos visuales.

A la hora de reconocer imágenes y objetos dentro de ellas, nuestra vista se centra en patrones volúmenes, siluetas, contornos, colores y líneas, entre muchas otras cosas. Si queremos que nuestra red neuronal sea capaz de generar o reconocer objetos de una forma adecuada, buscaremos que se comporte como la vista humana, o al menos de forma similar, yendo desde los conceptos más sencillos: líneas y colores; a los conceptos más complejos: caras y gestos, por ejemplo. Repasemos entonces como funcionan estas redes y la visión humana con un poco más de detenimiento.

Para entender la arquitectura de una red convolucional viajaremos hasta la década de 1960. Un estudio realizado sobre la retina de los gatos [17] descubría que la visión de estos animales agrupaba células por columnas y que cada columna servía a un propósito específico y sencillo. Una vez se tienen los objetos reconocidos por dichas columnas se comienza a producir la imagen general basada en los objetos captados, lo cual permite reconstruir la escena completa a partir de pedazos de la misma, idea que se utilizaría para la creación de las *CNN*.

Para hacer funcionar este complejo proceso, se concluyó que en lugar de que las neuronas aprendieran, con los pesos individuales por cada neurona que esto implica, había que agrupar neuronas entre sí, con el objetivo de que también compartan los pesos. De este modo se obtendría la agrupación por columnas que presentaban los gatos en su visión.

Conocida la idea tras las *CNN*, centrémonos en un ejemplo sobre el funcionamiento aplicado de estas redes. Imaginemos algo sencillo, como por ejemplo un mapache, y veamos como hemos de procesar la información para poder reconocerlo. Dentro de esta imagen el mapache puede encontrarse en cualquier parte de la misma; en el centro, en una de las esquinas, etcétera. Si dividimos la imagen por cuadrantes, puede ser que ocupe uno o varios a la vez, luego lo primero que hemos de asegurar es del reconocimiento del lugar que ocupa nuestro pequeño animal en la imagen. Para poder realizar esto, se implementan una serie de filtros (también llamados *kernel*). Nos bastará observar una pata del mapache para saber que si buscamos por zonas aledañas encontraremos más partes del animal que si saltamos a la zona opuesta en la imagen. Esta forma de razonar es un objetivo para el aprendizaje de la red convolucional. Las herramientas que nos ayuda a cumplir este objetivo son los filtros convolucionales, que, una vez son aprendidos, permiten a la *CNN* reconocer las partes deseadas de la imagen (el mapache).

Vamos a ver como se definen las imágenes para el ordenador, para ver como poder aplicar estos filtros.

Una imagen se puede representar como una matriz de píxeles, en la que cada uno de estos píxeles tiene un valor, sea la imagen en blanco y negro, escala de grises, o a color, con tres parámetros para los valores de mezcla de los colores primarios. La manera natural de recorrer esta matriz sería avanzar, píxel a píxel, por toda ella, hasta haberla recorrido por completo.

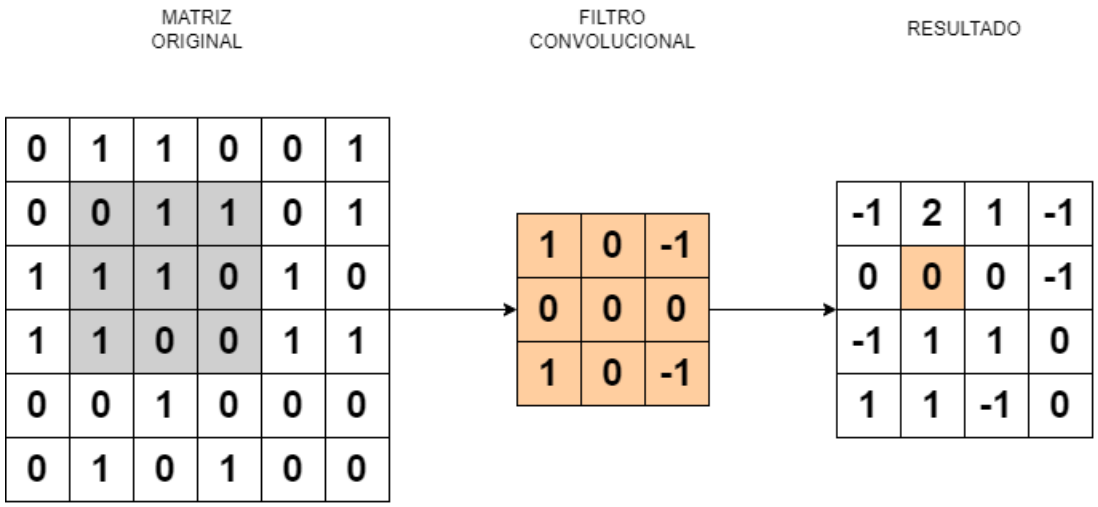


Figura 3.4: Funcionamiento de la convolución

Veamos, ahora, el funcionamiento de estos filtros.

Cuando nos referimos a aplicar un filtro convolucional sobre esta imagen, estamos tomando submatrices dentro de la misma y aplicando operaciones para producir una nueva matriz, llamada mapa de características de la imagen. Podemos ver el funcionamiento de un filtro para la convolución en la imagen 3.4.

En resumen, el funcionamiento de los filtros no es más que la multiplicación, elemento a elemento, de la matriz por el filtro, y su suma nos devuelve el elemento resaltado en el resultado. El filtro recorrerá, operando así, toda la imagen y con esto obtendremos el mapa de características. Es importante definir un buen número de píxeles de salto, pues estos definen como se recorrerá la matriz imagen, si se aplicará el filtro recorriendo cada píxel y las submatrices que estos generen, si se aplicará cada dos píxeles, etc.

Distintos filtros darán resultado a distintos mapas de características, lo cual es una técnica que se puede utilizar a la hora de manejar redes de convolución. Las distintas formas de tomar los filtros dependerán del tamaño de las submatrices tomadas, el número de píxeles de salto para el paso del filtro por la imagen y las operaciones utilizadas como convolución. Una vez hemos obtenido el mapa de características, utilizamos una función de activación para transformar los datos obtenidos.

Lo anterior es un proceso costoso de por sí, así que para llevar a cabo el entrenamiento de la red de forma más veloz, buscaremos condensar aún más la información de que disponemos.

Es en este punto donde entra en juego la capa de *pooling* tras realizar la convolución. El objetivo de esta capa es reducir el tamaño del mapa de características que ha pasado por la función de activación, perdiendo la menor información posible, pero reduciendo la dimensionalidad de los datos para un entrenamiento más rápido.

El funcionamiento es similar al de la convolución en cuanto a cómo se recorre el mapa de características tratado. Entre las técnicas de *pooling* más utilizadas están las siguientes:

- *Average pooling*, que toma el valor medio de los píxeles en el filtro.

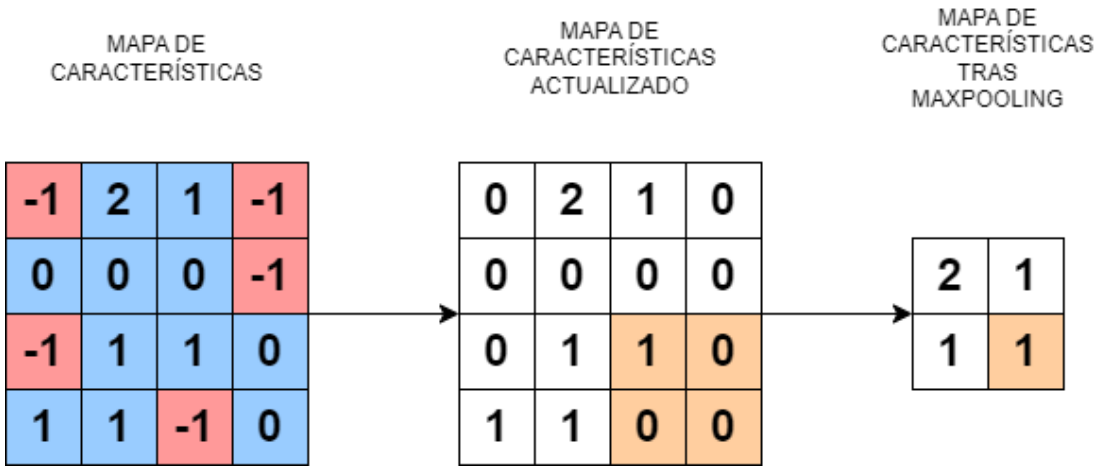


Figura 3.5: Ejemplo de *max pooling*

- *Max pooling*, que toma el valor máximo de los píxeles en el filtro.

Podemos ver en la imagen 3.5 un ejemplo del funcionamiento del *max pooling*. Primero, partimos del mapa de características, donde hemos resaltado en azul los elementos que no cambian al aplicar una función de activación *ReLU*, y en rojo los que sí que cambian. Tras ello, podemos ver como sobre el mapa de características actualizado con el paso de la función de activación, se utiliza el *max pooling* para obtener un mapa simplificado.

Cuando se ha terminado el proceso de *pooling*, pasamos a las capas de convolución *fully connected* y *softmax*, que nos permiten transformar las matrices obtenidas, primero en un vector, y luego en las probabilidades asociadas a la pertenencia en la imagen de un objeto.

Con esto termina nuestro viaje a través de la visión artificial, que utilizaremos para extraer características y generar imágenes relacionadas con ellas.

Capítulo 4

Redes Generativas

Ya conocemos todos los conceptos necesarios para una comprensión básica del tema de este trabajo, las redes generativas. Los objetivos de este capítulo son la introducción de las redes adversariales generativas, de forma teórica, para poder llegar a entender el funcionamiento de la *Deep Convolutional GAN*, o *DCGAN*, de las redes *Conditional GAN*, o *CGAN*, y del modelo *Laplacian GAN*, o *LAPGAN*. Una vez se haya explicado su funcionamiento teórico, podremos introducir su arquitectura e implementación y entrenar las redes y probar sus resultados. También veremos varios de los ejemplos de aplicación de las mismas, que resultan ser distintas implementaciones con la misma idea base.

Presentemos las redes generativas adversariales o *GAN*. Estas redes son los primeros modelos generativos y han resultado ser la base para redes más complejas, sobre todo en aquellas centradas en la generación de imágenes. Sus primeras implementaciones probaron que se podían obtener salidas a partir de codificaciones arbitrarias iniciales, y que con un entrenamiento adecuado los resultados eran más que satisfactorios.

Lo primero que hay que destacar de ellas es que son una herramienta útil y potente, si buscamos ejemplos *online*, podemos ver aplicaciones que van desde la generación de rostros fotorrealistas (*this person does not exist*) hasta los ejemplos que hemos visto en la figura 3.1b, que son imágenes generadas bajo demanda e interpretando el lenguaje natural.

A continuación, hagamos un resumen general de como son y funcionan estas redes.

4.1. Entrenamiento adversarial

Comenzaremos detallando como es el aprendizaje realizado por una *GAN*.

Estas redes aprenden a modelar la distribución de las entradas mediante el entrenamiento de dos redes en competición: un generador y un discriminador:

- El generador se centra en fabricar señales falsas a partir de ruido aleatorio, con el objetivo de engañar al discriminador.
- El discriminador, por otra parte, se entrena para distinguir entre señales reales y falsas, para poder desechar las señales que fabrica el generador, pero no las originales.

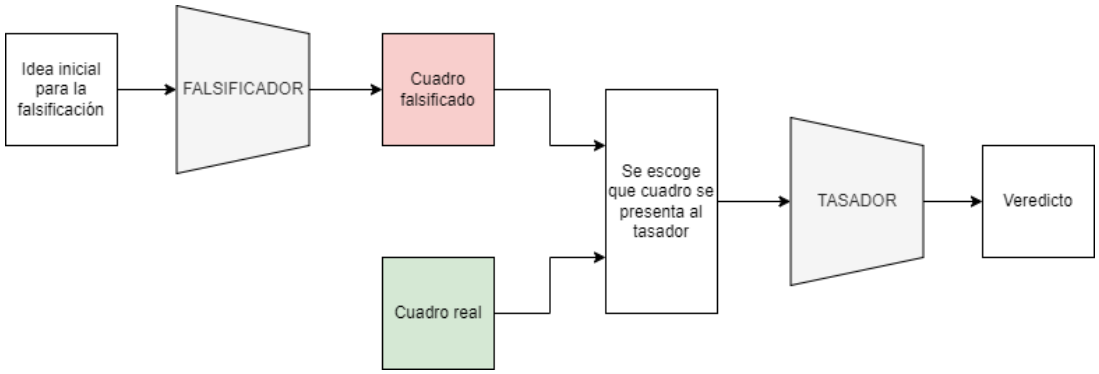


Figura 4.1: Ejemplo del escenario falsificador-tasador

Según progresa el entrenamiento, el generador mejora en su cometido, y por tanto el discriminador dejará de diferenciar entre las señales falsas y las verdaderas. Llegados a este punto, nos desharemos del discriminador y consideraremos que el generador es lo suficientemente bueno para realizar señales nuevas que se asemejen a las originales.

El principal problema de estas redes no es tanto el entrenamiento en sí, si no el como lograr que este sea estable, es decir, que el generador y discriminador se entrenen de forma coordinada para un aprendizaje simultáneo. Esto nace del hecho que para entrenar toda la red, la pérdida actuará sobre el discriminador, que es la segunda parte de esta red, lo que hace que sus parámetros se actualicen antes. Si se diera el caso que el discriminador convergiera antes que el generador, entonces la *GAN* no podrá funcionar, dado que no se podrán generar datos que superen la barrera del discriminador. Además, también hemos de tener en cuenta que las *GAN* producen salidas similares para codificaciones muy diferentes, lo cual es llamado colapso modal, y es otro de los problemas a tener en cuenta. Entendemos por colapso modal a la asignación del mismo resultado a muestras que deberían producir resultados distintos.

Con esta breve introducción del proceso de entrenamiento, expliquemos mejor el funcionamiento de estas redes.

Para comprender como es el entrenamiento, lo mejor es pensar en un escenario de competición como puede ser el de un tasador y un falsificador de cuadros. Inicialmente, el tasador conoce todos los cuadros falsos y reales, y es capaz de observar hasta la más mínima diferencia y desechar todo cuadro no real. A su vez, el falsificador va realizando intentos de copia de los cuadros reales e intenta pasar sus falsificaciones como verdaderas. Al principio, la tarea de identificar los cuadros falsos es fácil, el falsificador es torpe y subestima al tasador, y por tanto comete errores al realizar sus falsificaciones, pero, con el tiempo, el falsificador aprende, y el trabajo se le dificulta al tasador, que empieza a ser incapaz de determinar si un cuadro es real o no.

Podemos ver el un diagrama que representa el proceso en 4.1.

Para que todo este proceso funcione, hay de saber en todo momento cuales son cuadros reales y falsos, pero de forma que no lo sepan ninguno de los participantes del escenario que se maneja.

La solución es sencilla, basta con etiquetar los cuadros, si son reales con un 1 y si

son falsos con un 0, desde fuera del sistema. Si automatizamos este proceso, el paradigma dentro del aprendizaje es no supervisado, y controlamos el tipo de los cuadros sin necesidad de etiquetas que añadan complejidad al modelo.

Con el escenario claro, veamos el papel de las redes de la *GAN*, y el funcionamiento real del entrenamiento.

El objetivo del discriminador será aprender del conjunto de entrenamiento para distinguir las señales reales de las falsas. Durante esta parte del entrenamiento, sólo cambiaremos los parámetros del discriminador, y lo haremos como si fuera un clasificador binario y lineal, es decir, se entrenará para predecir en un rango entre 0 y 1, que representará la probabilidad de que una entrada recibida sea real, es decir, provenga del *dataset*.

Sin embargo esto es sólo la mitad de la red que buscamos. En intervalos regulares, el generador producirá una salida y buscará que el discriminador la etiquete con un número cercano a 1. Al principio, cuando esta salida se presente al discriminador, este clasificará la salida (que será su entrada) como real con una confianza cercana a 0. En base a esta puntuación, el optimizador tendrá que modificar los parámetros, donde se tendrá en cuenta el ejemplo presentado a la hora de entrenar con datos nuevos.

Es decir, el discriminador puede tener dudas de si una entrada es real o no, y, posteriormente, esto lo tendrá en cuenta la *GAN* completa. Una vez se ha clasificado, aplicamos la retropropagación desde la última capa del discriminador hasta la primera del generador, congelando los parámetros del discriminador durante este momento. De este modo conseguimos que se entrenen los parámetros del generador y que la calidad de las señales falsas sea mejor, de modo que puedan acercarse más al objetivo de ser clasificadas como correctas.

En su conjunto, la red generativa no es más que un equilibrio entre redes que compiten y cooperan a la vez. Cuando el entrenamiento de la *GAN* converge, el resultado será un generador capaz de sintetizar bien datos nuevos, y el discriminador nos devolverá valores cercanos a 1 para estos datos sintéticos. Una vez alcanzado este objetivo ya no servirá de utilidad el discriminador y será desechado.

La manera de utilizar el generador será proporcionando entradas ruidosas, ruido estadístico con la forma de los datos (matriz, vector...), y a partir de este ruido se generará una salida que procure simular las características principales de los datos de entrenamiento.

Este funcionamiento se puede ver utilizando el diagrama 4.2, que es igual que el del escenario de falsificador-tasador.

Como todo entrenamiento, esta red necesita de una función de pérdida para poder actualizar los pesos y refinar el resultado que producen las redes.

Matemáticamente, se utiliza la siguiente expresión:

$$\mathcal{L}(\theta^{(G)}, \theta^{(D)}) = -\mathbb{E}_{\mathbf{x}\sim P_{data}}(\log(\mathcal{D}(\mathbf{x}))) - \mathbb{E}_{\mathbf{z}}(1 - \log(\mathcal{D}(\mathcal{G}(\mathbf{z})))),$$

donde se ha representado por $\theta^{(G)}$ y $\theta^{(D)}$ a los parámetros del generador y el discriminador (parámetros de RRNN).

Esta fórmula no es más que la entropía cruzada binaria, es decir, la suma (en negativo) de la esperanza de identificar correctamente datos reales y la esperanza de clasificar los datos falsos como erróneos. Esta última esperanza no es más que 1 menos la posibilidad

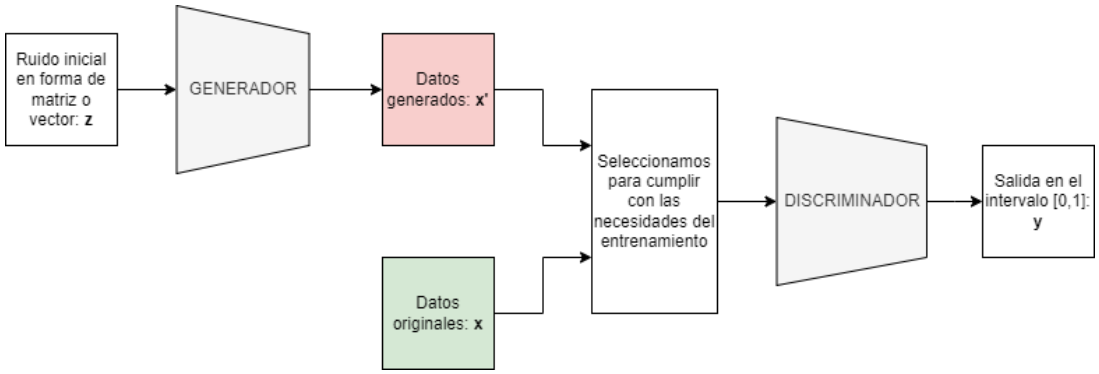


Figura 4.2: Diagrama del funcionamiento de una *GAN*

de identificar como correcto los datos sintéticos. Como el logaritmo no cambia la posición de los mínimos (debido a su convexidad), entonces se puede utilizar para simplificar las salidas y obtener resultados uniformes y más fáciles de tratar.

En resumen, utilizamos la entropía cruzada para calcular la pérdida y la retropropagamos para poder cambiar los parámetros del discriminador y el generador, de modo que podamos realizar copias sintéticas que pasen por reales.

Para terminar con el entrenamiento, describamos como funciona la función de pérdida.

Se comenzará con dos *minibatches* o minilotes que se dan de entrada al discriminador:

- $\mathbf{x}$, el minilote de imágenes reales, con etiqueta 1.
- $\mathbf{x}' = \mathcal{G}(\mathbf{z})$, el minilote de imágenes sintéticas, con etiqueta 0. En esta fórmula, $\mathcal{G}$ representa al generador y $\mathbf{z}$ a una entrada de ruido estadístico.

Para minimizar la función de pérdida, los parámetros del discriminador $\theta^{(D)}$ se actualizan dependiendo de la identificación de las entradas (que representamos por $\mathcal{D}$):

- Si tenemos $\mathcal{D}(\mathbf{x})$, entonces queremos obtener un 1 como respuesta.
- Si tenemos $\mathcal{D}(\mathcal{G}(\mathbf{z}))$, entonces queremos obtener un 0 como respuesta. Esto es que si se lo restamos a 1, obtengamos un 1 como respuesta.

Las siguientes dos fases corresponden al proceso minimización de la función de pérdida, luego convendrá tenerlas en cuenta para realizar un entrenamiento adecuado.

Durante el entrenamiento del generador, se considera la pérdida del generador como un juego de suma 0:

$$\mathcal{L}^{(D)}(\theta^{(G)}, \theta^{(D)}) = -\mathcal{L}^{(G)}(\theta^{(G)}, \theta^{(D)}).$$

Donde se asigna un superíndice para identificar a cual de las dos redes que conforman la GAN pertenece cada una de las funciones de pérdida.

Reescribiendo esta ecuación de la siguiente manera:

$$\mathcal{V}(\theta^{(G)}, \theta^{(D)}) = -\mathcal{L}^{(D)}(\theta^{(G)}, \theta^{(D)}).$$

De donde se obtiene $\mathcal{V}$, la función objetivo de un problema de optimización doble.

Ahora, minimizando $\mathcal{V}$ se cumplirán los objetivos del generador, y maximizándola, se busca que el discriminador cumpla los suyos.

Entonces, se buscarán los parámetros del generador mediante un problema de maximización y minimización (*minimax*), que es lo necesario para poder generar nuevo contenido, el objetivo final de estas redes.

El problema de optimización sería el siguiente:

$$\theta^{(G)*} = \arg \min_{\theta^{(G)}} \max_{\theta^{(D)}} \mathcal{V}(\theta^{(G)}, \theta^{(D)}),$$

donde $\theta^{(G)*}$ son los parámetros que se quiere tener y $\theta^{(G)}$ son los parámetros de la red durante el entrenamiento.

Para obtener la solución a este problema se procura engañar al discriminador, pretendiendo que datos generados sean reales, es decir, asignándoles la etiqueta 1.

Además, respecto al problema de obtener los parámetros:

- Maximizando respecto $\theta^{(D)}$, el optimizador envía cambios en el gradiente al discriminador para que los datos sintéticos se tomen como reales.
- Minimizando respecto $\theta^{(G)}$, el optimizador entrena al generador sobre como engañar al discriminador.

Sin embargo, con este modelo así planteado, en la práctica, el discriminador confía en la clasificación realizada y no cambia sus parámetros (los congelamos) y los cambios de gradiente son tan pequeños que que no permitirán que el generador converja.

Este problema se soluciona de la siguiente manera: hay que reformular la forma en la que se ha planteado la pérdida del generador.

Considerando:

$$\mathcal{L}^{(G)}(\theta^{(G)}, \theta^{(D)}) = -\mathbb{E}_{\mathbf{z}}(\log(\mathcal{D}(\mathcal{G}(\mathbf{z})))),$$

se ha obtenido una función de pérdida que maximiza la probabilidad de que el discriminador cometa un error, en vez de minimizar la probabilidad de acierto.

Con esto se abandona el juego de suma cero para pasar a un algoritmo puramente heurístico, lo que soluciona el problema que había surgido.

Con esto, con sólo cambiar los parámetros del generador cuando se entrene toda la red, dado que los gradientes del discriminador pasarán hacia el generador, gracias al hecho de que congelamos sus parámetros durante el entrenamiento adversarial, se consiguen los resultados esperados.

Al orientar este tipo de redes hacia el *DL*, encontramos dos formas comunes de implementar estas redes, dependiendo del tipo de dato que se quiera generar:

- Si queremos imágenes o vídeo, se utilizan *CNN* principalmente.
- Si se quieren elementos de una sola dimensión (como secuencias temporales p.ej) o *Natural Language Processing* (*NLP*), se manejarán redes recurrentes, como lo son las *RNN*, *LSTM* o *GRU*.

Esto termina el apartado teórico del entrenamiento y funcionamiento de una red generativa adversarial. Aunque ahora le añadiremos apellidos, según las mejoras e implementaciones que sigan, el proceso subyacente es el mismo, y lo que cambia es la arquitectura de la red.

4.2. *Conditional GAN*

Esta sección va dedicada a explicar el concepto de *CGAN* o *Conditional GAN*, pues es una red que utilizaremos en la implementación de la siguiente red generativa que explicaremos. Veamos porqué estas redes suponen una mejora sustancial respecto de las *GAN* normales y corrientes.

Las redes generativas tienen un defecto principal, este es que su generación proviene de un ruido aleatorio inicial, el cual nos quita el control sobre los dígitos que se generan. La solución a este defecto es lo que se pretende lograr con la red condicional.

Utilizando el mismo principio de las *GAN*, vamos a imponer una condición sobre el generador y discriminador para poder obtener los dígitos que les requiramos. Dicha condición se traducirá en un vector cuyos valores representarán un dígito. Para realizar esta representación lo haremos de la siguiente manera: utilizaremos un vector de 10 ceros, de modo que en la posición del dígito que queremos representar (de 0 a 9) figurará un valor 1 y en el resto de posiciones un 0. A este vector lo llamaremos etiqueta del dígito, que nada tendrá que ver con cómo etiqueta el discriminador (que discierne entre imágenes reales y falsas con etiquetas 1 y 0).

Con el vector ya formado, hay que encontrar una forma de que la red lo asocie a las imágenes que genera y que están en el *dataset*. Para ello, juntaremos el vector a la imagen que queremos producir, es decir, antes de que se genere una imagen, y previamente a la entrada del discriminador de las imágenes a clasificar. En conclusión, el vector actuará antes de que se inicie el trabajo de cualquiera de las dos redes que conforman la *GAN*.

A excepción de la etiqueta del dígito, la red es exactamente igual que una *GAN*:

- Para el generador, la etiqueta del dígito se concatenará al vector de ruido antes de la capa densa.
- Para el discriminador, añadiremos una nueva capa densa que se encargará de procesar la etiqueta del dígito para sí poder concatenarla con las entradas para las *CNN* sucesivas de la red.

También conviene tener en cuenta que pese a que, formalmente, las funciones de pérdida de generador y discriminador cambian, estas siguen siendo la entropía cruzada binaria, solo que ahora estarán condicionadas por la etiqueta del dígito.

Veamos cuales son los cambios en ellas($\mathcal{L}^{(G)}$ es la pérdida del generador y $\mathcal{L}^{(D)}$ la del discriminador):

$$\mathcal{L}^{(G)}(\theta^{(G)}, \theta^{(D)}) = -\mathbb{E}_{\mathbf{z}}(\log(\mathcal{D}(\mathcal{G}(\mathbf{z}|\mathbf{y}')))).$$

$$\mathcal{L}^{(D)}(\theta^{(G)}, \theta^{(D)}) = -\mathbb{E}_{\mathbf{x}\sim P_{data}}(\log(\mathcal{D}(\mathbf{x}|\mathbf{y}))) - \mathbb{E}_{\mathbf{z}}(1 - \log(\mathcal{D}(\mathcal{G}(\mathbf{z}|\mathbf{y}')))).$$

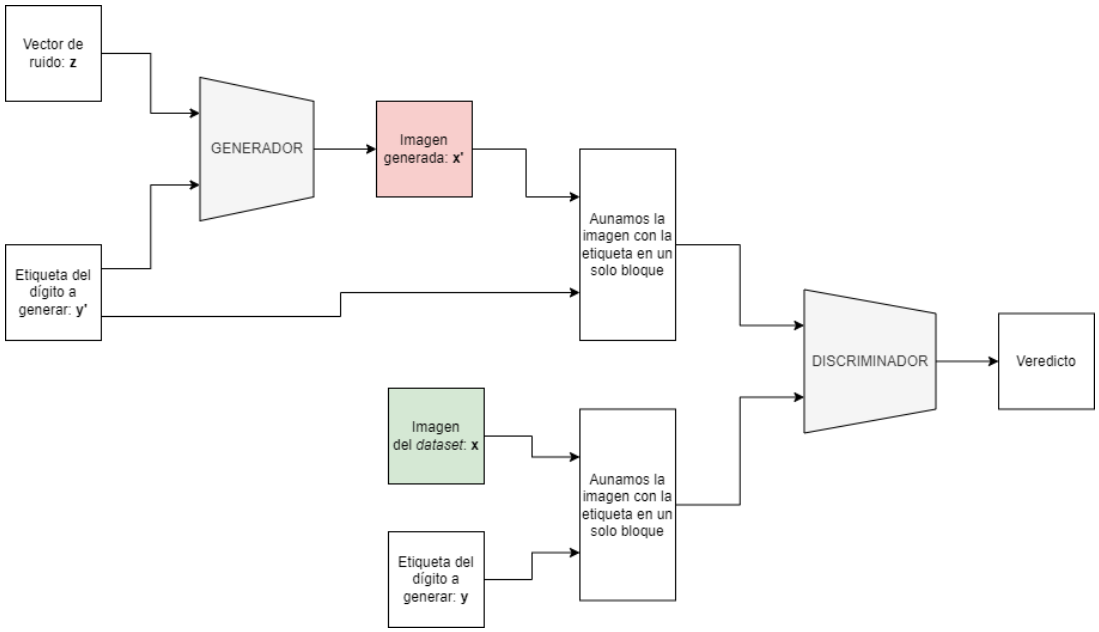


Figura 4.3: Diagrama del entrenamiento completo de una *CGAN*

Los objetivos de las funciones de pérdida no han cambiado significativamente; el objetivo de la pérdida del generador busca minimizar que se reconozcan sus imágenes como falsas, aunque condicionadas a la etiqueta del dígito seleccionada, y la pérdida del discriminador buscará minimizar el error de predecir incorrectamente las imágenes reales y falsas, dadas las etiquetas de dígito. El entrenamiento de la red tampoco tiene cambios fundamentales, aunque veremos ahora dos diagramas para explicar de nuevo cada parte del entrenamiento de la red en su completitud.

En el diagrama 4.3 podemos observar como el discriminador se entrena frente a datos falsos y reales simultáneamente, donde ambos irán acompañados, aparte de por su etiqueta de falsedad, por la etiqueta de dígito correspondiente. En este caso, las actualizaciones de parámetros que realicemos durante esta parte del entrenamiento no afectarán a los parámetros del generador, si no que sólo se verá modificado el discriminador.

Debido a los cambios que tenemos, aunque ligeros, en la pérdida del generador, conseguimos que este se centre en aprender dígitos específicos de *MNIST* dado un vector identificador de los mismos, para finalmente poder engañar al discriminador.

En el diagrama 4.4 se ve como procede el entrenamiento de los parámetros del generador. En este caso, el generador produce datos falsos condicionados a una etiqueta y el discriminador los clasifica, y congelando los parámetros del discriminador, modificamos únicamente los del generador para destilar el resultado que obtenemos del mismo.

El entrenamiento completo sigue el siguiente proceso; primero entrenamos el discriminador de la *CGAN* con un lote de datos verdaderos y falsos condicionados por la etiqueta del dígito. Tras este paso, modificamos los parámetros del generador mediante el entrenamiento adversarial y la congelación del discriminador, también condicionando los datos que pasamos como entrada a ambas redes.

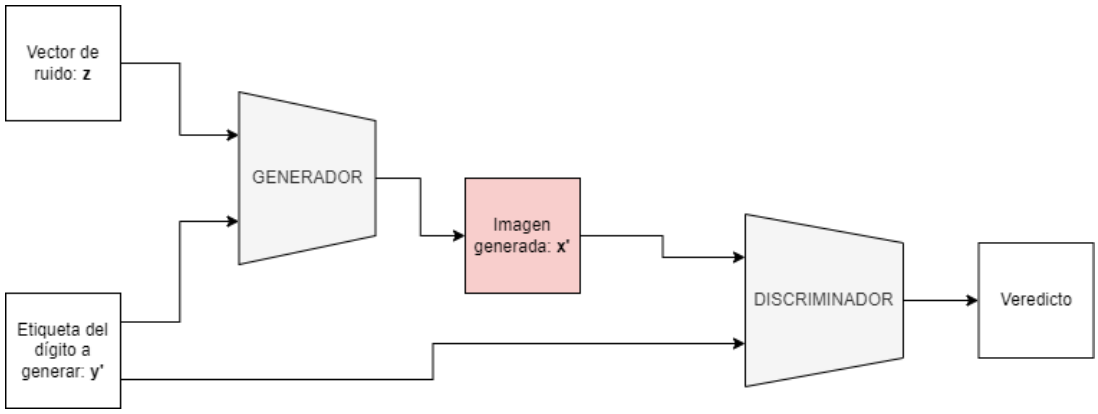


Figura 4.4: Diagrama del entrenamiento del generador en una *CGAN*

La conclusión que se obtiene de este proceso es haber obtenido una mejora sobre la *GAN*, pues ahora disponemos de un agente al que realizar peticiones específicas, en este caso dígitos, para obtener resultados personalizados.

4.3. *Laplacian GAN*

Llegados a este punto, introducimos una red generativa que es capaz de producir resultados con mejor definición, aunque de baja dimensión en cuanto a píxeles se refiere, basada en la pirámide Laplaciana.

Nuestro objetivo será construir un modelo generativo de imágenes naturales, aunque de nuevo el primer problema que se nos presenta es la alta dimensionalidad de las imágenes, además de su complejidad. Para solucionar este problema, se plantea una red que genera imágenes plausibles de tamaños 32×32 y 64×64 píxeles.

La manera de implementar este modelo será explotando la estructura multiescala de las imágenes naturales, utilizando una serie de modelos generativos, cada cual capturará parte de la estructura en una escala particular de una pirámide Laplaciana.

Una pirámide Laplaciana es, a grandes rasgos, una representación invertible y lineal de imágenes, consistente en un conjunto de imágenes en banda que están espaciadas por una octava, y un residuo de baja frecuencia. Formalmente consta de dos partes:

- La primera es una operación que reduce la escala de una imagen, I, la difumina y la recorta. Partiendo de que I de tamaño $j \times j$, y denotando por d a la función de reducción de escala, obtenemos una imagen $d(I)$ de tamaño $\frac{j}{2} \times \frac{j}{2}$.

- La segunda es una función de reescalado para devolver la imagen a su tamaño original. Esta función será u, y duplicará el tamaño de una imagen cualquiera I de tamaño $j \times j$, de modo que $u(I)$ tiene tamaño $2j \times 2j$.

Tomando las dos funciones definidas para la pirámide Laplaciana, podemos construir una pirámide Gaussiana a partir de una imagen, $\mathcal{G}(I) = \{I_0, ..., I_K\}$, donde I_0 es la imagen

original e I_k es la composición d-ésima de d actuando sobre I, por ejemplo: $I_2 = d(d(I))$. K representa el número de niveles que hay en la pirámide, seleccionados de modo que el número de píxeles del último nivel sea bajo (menor o igual que 8×8).

Ahora, a partir de esta pirámide, construimos la pirámide Laplaciana. Para calcular los coeficientes h_k de cada nivel de esta nueva pirámide, $\mathcal{L}(I)$, se construyen tomando la diferencia entre niveles adyacentes de la pirámide Gaussiana, utilizando la función de reescalado para que los tamaños con los que se opera sean compatibles:

$$h_k = \mathcal{L}_k(I) = \mathcal{G}_k(I) - u(\mathcal{G}_{k+1}(I)) = I_k - u(I_{k+1}).$$

Intuitivamente, cada nivel captura la estructura de la imagen en una escala particular, y cuando se llega al último nivel de la pirámide Laplaciana, dado que h_K no puede ser la diferencia, tomamos el residual de baja frecuencia, que no es más que tomar $h_K = I_K$.

Ahora, si queremos realizar la reconstrucción de una imagen, basta con realizar la siguiente recurrencia:

$$I_k = u(I_{k+1}) + h_k,$$

la cual comienza con $I_K = h_K$, de modo que la imagen reconstruida será $I = I_0$. En otras palabras, empezando desde una imagen que ha sido reducida en escala, y mediante un proceso de reescalado obtenemos la imagen original, tras añadir la diferencia h correspondiente.

Con esta aproximación, se rompe el problema principal en otros más pequeños y manejables. En cada escala de la pirámide, entrenaremos una *GAN* convolucional (como aproximación a las propuestas en [16]).

Los modelos generativos han sido bien estudiados desde 2014, y generalmente se tomaron dos aproximaciones distintas para dar los primeros pasos en este campo: los modelos paramétricos y no paramétricos. El funcionamiento de estos modelos se basa en la copia de parches de las imágenes para poder sintetizar texturas o generar super resolución de imágenes.

Ya introducido el funcionamiento de la pirámide Laplaciana, definamos el funcionamiento de la red *LAPGAN*.

La propuesta principal de esta red es la mezcla entre una *CGAN* y la pirámide que acabamos de introducir. Para explicar el modelo, comencemos con el proceso de muestreo.

Si tomamos un conjunto de redes convolucionales generativas: $\{G_0, ..., G_K\}$, las cuales capturan cada una una distribución h_k de las imágenes naturales a distintos niveles de la pirámide, el muestreo se realizará de forma que se reconstruya una imagen como en la pirámide, con la excepción de que los h_k provienen de redes generativas y no de una pirámide Gaussiana. De forma matemática, esto se expresa de la siguiente manera:

$$I_k = u(I_{k+1}) + h_k = u(I_{k+1}) + G_k(z_k, u(I_{k+1})).$$

La recurrencia en este caso comenzará con $I_{K+1} = 0$ y utilizando el modelo G_K obtendremos una imagen residual I_K, para lo que habremos necesitado un vector de ruido z_K. Una vez tenemos esta imagen residual, continuamos como si fuera la pirámide

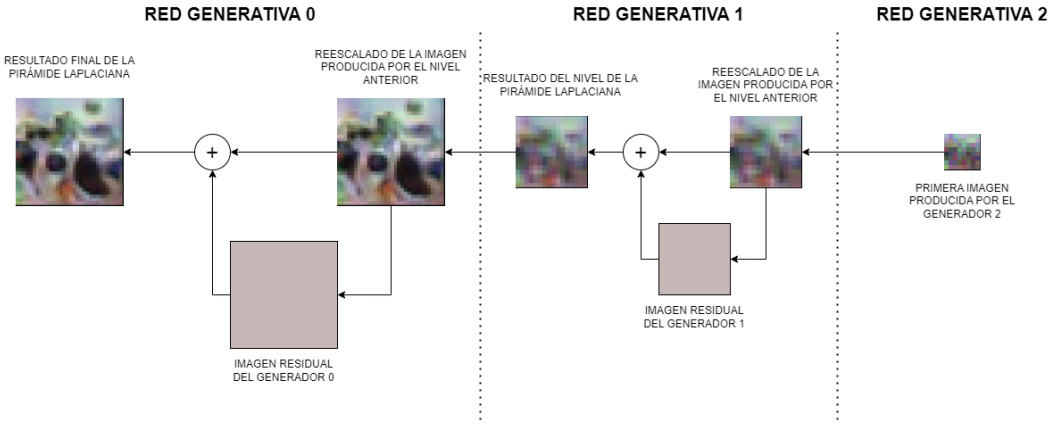


Figura 4.5: Funcionamiento de una *LAPGAN* como en [10]

Laplaciana, con el objetivo de reconstruir una imagen que nunca existió, lo que da lugar al proceso generativo.

Cabe destacar que los modelos de cada nivel, a excepción del último, son modelos generativos condicionales que toman una imagen reescalada, I_{k+1}, como variable condicional, además del vector de ruido z_k, para la generación de una nueva imagen que se parezca a la variable condicional. Esto se traduce en que, en cada nivel de la pirámide, hay una serie de modelos $CGAN$ enlazados que forman todo el proceso de generación de imágenes. Aparte, también estaremos creando una pirámide Laplaciana por cada elemento del conjunto de entrenamiento, y en cada nivel escogeremos si utilizar una construcción de los coeficientes de forma estándar o mediante la red generativa condicional adecuada. Esta elección se realiza de forma estocástica, es decir, aleatoria.

También hay que mencionar que cada G_k es una red convolucional que utiliza una imagen reescalada como entrada, así como un vector de ruido z_k, y como toda red generativa, irá acompañada de un discriminador D_k. Este discriminador recibirá como entrada los coeficientes junto con la imagen reescalada y predirá si la imagen era real o falsa. En el último paso de la pirámide, el modelo solo recibirá de entrada el coeficiente h_K, pues la imagen residual será demasiado pequeña para su discriminación.

Al final, todo este proceso no ha sido más que una ruptura de la solución al problema de generación de imágenes en una serie de refinamientos sucesivos, olvidándonos del concepto de fidelidad global para poder discriminar en cascada las sucesivas salidas, con lo que basta que cada paso sea posible para que la red funcione en su totalidad. Además, obtenemos a mayores la siguiente ventaja: el entrenamiento independiente de cada nivel de la pirámide nos ayuda a evitar la memorización de imágenes del conjunto de entrenamiento.

El entrenamiento de esta red se realiza de forma no supervisada, aunque se puede explorar la idea de añadir etiquetas de clase. Esto último se puede realizar añadiendo un vector c que indique la identidad de clase como una variable condicional más para los generadores y discriminadores.

Terminemos las explicaciones teóricas con otra red generativa basada en redes convolucionales, pero con mejoras sustanciales frente a los problemas de las GAN.

4.4. *Deep Convolutional GAN*

Como ya mencionábamos en la sección 4.1, al final de la sección, el uso de *CNN* y *RNN* está más que justificado para la generación de contenido. En esta sección vamos a estudiar a uno de los sucesores más importantes de las redes *LAPGAN* en la tarea de generación de imágenes.

Estas nuevas redes se centraron en el uso de las redes convolucionales para conseguir objetivos, no sólo de competitividad con el resto, si no de estandarización del método de diseño para las arquitecturas de redes generativas, de modo que se pudiera asegurar una mejor convergencia.

Alrededor de la fecha de publicación de [34] (2016), el enfoque de uso para las *CNN* se centraba en el aprendizaje supervisado. En este artículo, que introducía las redes generativas convolucionales profundas (o *DCGAN*), se buscaba armonizar esta separación existente entre el aprendizaje no supervisado y las redes convolucionales, entre muchos otros objetivos. Estas nuevas redes se presentaban como una serie de restricciones sobre el modelo *GAN* y, utilizando redes convolucionales, conseguían probar que este novedoso modelo comprendía bien las características y representaciones de los datos de entrenamiento, produciendo réplicas más que satisfactorias.

El objetivo en aquel entonces era encontrar representaciones de las características de los datos que pudieran ser reutilizadas, lo que permitía la generación automática de nuevos datos. Además de este objetivo principal, también se planteaba en [34] como atacar el problema de convergencia que presentaban las *GAN*, pues incluso cambios pequeños en la arquitectura llevaban a situaciones en las que la red no convergía como se deseaba. La manera de acercarse a cumplir estos objetivos seguía cuatro pasos esenciales claros:

1. Era necesario dar restricciones para las arquitecturas de las redes, de modo que se pudiera conseguir una mejora en la convergencia.

2. También produciría mejores resultados utilizar discriminadores entrenados para la clasificación de imágenes.

3. Se añadirían visualizaciones para mostrar las representaciones y filtros aprendidos por la red para poder generar nuevas muestras.

4. Demostrarían que estas redes poseían propiedades aritméticas que permitían una mejor manipulación de las cualidades semánticas de los resultados generados.

Las *DCGAN* resultaron ser una buena propuesta tanto para la generación de imágenes como para la clasificación de las mismas, resultando ventajosas frente a los métodos de máxima verosimilitud del momento.

Centrémonos en las restricciones que se planteaban sobre la arquitectura, de modo que sea claro el porqué estas redes produjeron tan buen resultado. Hasta que no llegaron estas redes, no se había conseguido un buen resultado en la mezcla entre redes generativas y convolucionales. Aunque otros artículos, como [10], desarrollaban un acercamiento

alternativo a las *GAN*, las *DCGAN* fueron una solución más confiable que las *LAPGAN* en la generación de nuevas muestras de datos.

Explorando distintos modelos se obtuvo la arquitectura final para las redes generativas convolucionales, que producían un entrenamiento más estable y menos dependiente del *dataset*, además de ser más adecuado para mayor resolución de imágenes. Los fundamentos de esta nueva arquitectura también eran cuatro, como los pasos para obtener sus objetivos:

1. La primera consideración sobre la arquitectura se centraba en eliminar el *pooling*, para mantener la atención únicamente en convoluciones con salto. Esto permitía a la red aprender como se modifica la escala de las imágenes con las que se entrena.

2. Como segundo cambio a las arquitecturas anteriores, se eliminaba el uso de capas completamente conectadas (*fully connected*). La manera de conseguir esto era añadir una única capa de este tipo al comienzo de la red, de modo que se pudiera transformar el vector de ruido en una matriz de tamaño adecuado, que realmente era una multiplicación matricial en vez de una capa *fully connected*. En el caso del discriminador, se añadía una última capa, que transformaba la salida del mismo en un vector para poder utilizar la función de activación.

3. La tercera modificación suponía aplicar la normalización de lotes, lo que estabilizaba el entrenamiento, llevando cada entrada a media 0 y varianza 1. Este cambio también resolvía el problema de escoger una mala inicialización de los datos y ayudaba a un buen flujo del gradiente en el modelo. Estas tareas resultan ser críticas en los pasos iniciales del aprendizaje del modelo y evitan el colapso modal de las *GAN*. Sin embargo, no se puede abusar de la normalización, pues usada en cada capa genera inestabilidad y oscilaciones en el entrenamiento. Para evitarlo, no se añade una capa de normalización ni a la salida del generador, ni a la entrada del discriminador.

4. El cuarto y último cambio en la arquitectura fue la sustitución de la activación *ReLU* por la *Leaky ReLU* [24, 44]. Utilizar funciones acotadas mejoraban la capacidad de aprendizaje del modelo sobre la distribución de los datos de entrenamiento, especialmente en casos de alta resolución de imágenes.

Estas mejoras en la arquitectura produjeron muy buenos resultados, y se mantuvieron firmes, pues se utilizaron redes convolucionales y recurrentes, hasta que aparecieron los *Transformer* y los métodos basados en la atención. Aun así, estudiar estos métodos es útil para comprender el camino que se ha recorrido y el funcionamiento de las redes actuales, que se basan en las presentadas para obtener resultados y generar nuevo contenido.

Con esta explicación, termina la parte teórica del trabajo, y daremos parte a la solución propuesta, que presentará las arquitecturas específicas de una *CGAN*, una *LAPGAN* y una *DCGAN*, además de implementaciones y pruebas de estas redes.

Parte II

Solución propuesta

Capítulo 5

Arquitectura e implementación

En esta segunda parte del trabajo nos centraremos en la implementación de las redes profundas que se han introducido en el capítulo 4 y en las conclusiones del trabajo.

Todo el código implementado se puede encontrar en el repositorio de *GitHub*: https://github.com/mapacheAgridulce/TFG-Generacion-elementos-DL.git.

Comenzaremos describiendo las arquitecturas de las redes implementadas.

5.1. *CGAN*

Seguiremos el mismo orden por el que se han introducido las redes en el capítulo anterior. La primera red que se había presentado era la *CGAN*, que nos permitía escoger números a generar a partir de un vector con una etiqueta de dígito. No seguiremos una implementación estrictamente cercana a las *GAN* originales, si no que la realizaremos centrada en la generación de imágenes mediante *CNN*, de modo que se puede entender esta implementación como una mejora sobre las *CGAN* originales, previas a la armonización de redes convolucionales y generativas.

Veamos como realiza el proceso de generar y discriminar imágenes hasta obtener un generador adecuado que produce las salidas que esperamos.

El proceso del generador es como sigue.

Partiremos de un vector de ruido aleatorio y una etiqueta, escogida aleatoriamente, de dígito. Estas dos cosas conformarán un vector 100 dimensional y un vector 10 dimensional respectivamente.

Con estos dos vectores definidos, y tras concatenarlos, pasamos el vector 110 dimensional por una capa densa, con el objetivo de obtener un vector de dimensión $7 \times 7 \times 128$. Esto es para poder dar forma a este nuevo vector, transformándolo en una imagen de tamaño 7×7, sobre la que vamos a aplicar 128 filtros.

Una vez dada forma a la imagen de tamaño cuadrado de 7 píxeles, comenzamos el proceso de convolución hasta tener una imagen de tamaño 28×28, como las de MNIST. Para llevar a cabo el proceso de convolución, seguimos los siguientes pasos:

1. Realizamos un proceso de normalización de lotes sobre la imagen inicial, y luego

aplicamos una activación *ReLU*.

2. Utilizamos una capa convolucional traspuesta de 128 filtros, de tamaño 5×5 y con un salto de 2 píxeles.

3. A la salida de esta convolución traspuesta se le aplica de nuevo normalización de lotes y una activación *ReLU*.

4. Se vuelve a utilizar una capa convolucional traspuesta, pero esta vez de 64 filtros, con el mismo tamaño de antes y con salto de 2 píxeles.

5. Normalizamos la salida y activamos mediante una *ReLU*.

6. Realizamos una convolución traspuesta de 32 filtros de tamaño 5, con salto de un píxel.

7. Aplicamos la última normalización y activación *ReLU*.

8. Utilizamos una última convolución traspuesta de un filtro de tamaño 5 y salto 1, y activamos su salida con una sigmoide.

Tras este proceso, obtenemos una imagen falsa de *MNIST*, producida por el generador de la red *CGAN*.

Ahora, veamos el proceso que sigue el discriminador.

Esta vez comenzaremos con sólo el vector de etiqueta del dígito, que transformaremos, primero en un vector de tamaño $7 \times 7 \times 16$ mediante una capa densa, y luego en un mapa 28×28 mediante una capa de *Reshape*.

Una vez tenemos el mapa para la etiqueta del dígito, podemos empezar a trabajar con la discriminación de las imágenes. Para ello, concatenamos una imagen (real o falsa) con su etiqueta correspondiente, ahora transformada en un mapa del mismo tamaño. Una vez concatenados, utilizamos una activación *Leaky ReLU* antes de realizar la primera convolución, y seguimos el siguiente proceso:

1. Activamos la entrada con una *Leaky ReLU*.

2. Comenzamos la primera convolución, de 32 filtros de tamaño 5 y salto 2.

3. Su salida pasa por una función de activación *Leaky ReLU*.

4. Realizamos otra convolución, de 64 filtros de tamaño 5, con salto 2.

5. Volvemos a activar la salida de esta convolución con una *Leaky ReLU*.

6. Utilizamos una convolución de 128 filtros, con salto 2, de tamaño 5×5.

7. La salida vuelve a pasar por una *Leaky ReLU*.

8. Realizamos la última convolución, pero esta vez con saltos de 1 píxel y 256 filtros de tamaño 5.

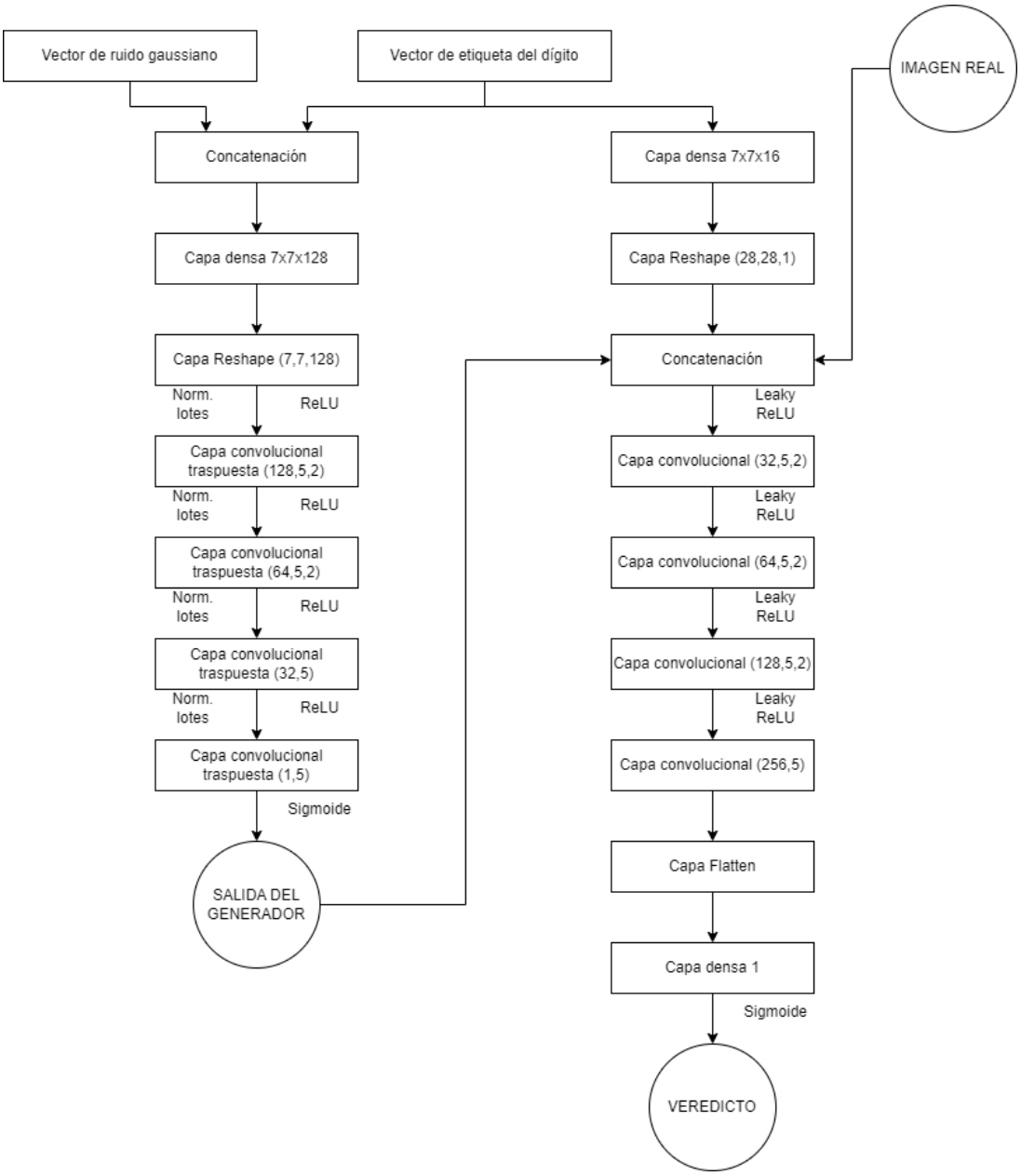


Figura 5.1: Arquitectura completa de una red *CGAN*

9. Utilizamos una capa *Flatten* para transformar esta última salida en un vector que podamos interpretar.

A esta última salida, la transformamos en un elemento mediante una capa densa, de modo que se active mediante una sigmoide y se obtenga el veredicto sobre la veracidad o falsedad de la entrada dada al discriminador.

Para todas las funciones de activación *Leaky ReLU* anteriores manejamos el mismo coeficiente de pendiente $\alpha = 0,2$.

Cuando juntamos toda la arquitectura, obtenemos el diagrama 5.1.

5.2. *LAPGAN*

Continuemos el recorrido con la arquitectura de la LAPGAN implementada. Esta arquitectura también tiene diferencias con la que se originó en [10]. En nuestra implementación habrá tres niveles de la pirámide laplaciana, y cada nivel estará conformado por una red generativa, por lo que se tendrán tres generadores y tres discriminadores en la misma red.

Veamos la manera de proceder de los generadores y de los discriminadores por separado, primero veremos los generadores, que se denotarán por $\mathcal{G}_0$, $\mathcal{G}_1$ y $\mathcal{G}_2$, y luego los discriminadores (emparejados por subíndice), que llamaremos $\mathcal{D}_0$, $\mathcal{D}_1$ y $\mathcal{D}_2$.

Lo primero que hay que tener en cuenta es que la entrada de los discriminadores $\mathcal{D}_0$ y $\mathcal{D}_1$ está condicionada, porque la manera de recorrer la red es comenzando por la red generativa 2, luego la 1, y finalmente la 0. Al condicionar la salida con una imagen de la anterior red, podemos generar una imagen similar a la condición, lo que aúna la salida; es decir, si no se impusiera una condición, no sería posible que las distintas redes de la pirámide generaran la misma imagen a distintas resoluciones.

Ahora que ya sabemos por qué se condiciona la entrada de parte de la pirámide, expliquemos cada uno de los niveles, yendo del de menor índice hacia el de mayor índice.

Comencemos viendo la arquitectura de $\mathcal{D}_0$ y $\mathcal{G}_0$. Estas dos redes neuronales conforman el último nivel de la pirámide, y producen las imágenes de mayor tamaño, que consideramos el producto final de toda la pirámide Laplaciana. Para esta red, utilizaremos capas lineales y de convolución, normalización de lotes y un ajuste de tamaño para los filtros y las dimensiones de los tensores y vectores. Describamos las capas de $\mathcal{D}_0$.

1. Lo primero que tendremos que hacer será concatenar la entrada del discriminador con la condición dada por la imagen de la anterior red.

2. Con esta entrada concatenada, realizaremos una primera convolución, con una entrada de 3 canales y 128 filtros, y un tamaño de convolución 5×5. Esto se activará con una *Leaky ReLU* y se normalizará.

3. La salida de la anterior capa pasará por una segunda convolución, que tomará el tensor con los 128 filtros y utilizará 128 filtros, un tamaño de 5×5 y un salto de 2 píxeles. Como activación de esta capa, también utilizaremos una *Leaky ReLU* y normalización.

4. Tras la segunda convolución, daremos forma a su salida para que la entrada a la última capa tenga la forma adecuada.

5. Finalmente, utilizaremos una sigmoide y una capa lineal para terminar con el proceso de $\mathcal{D}_0$.

Para $\mathcal{G}_0$ manejaremos convoluciones traspuestas y capas lineales, además de tener que realizar una reasignación de tamaño sobre las salidas de las capas anteriores. También necesitaremos utilizar la normalización de lotes entre capas.

1. Lo primero que haremos es darle forma a la imagen que vamos a generar, condicionada por la imagen que produce el nivel anterior de la pirámide. Para poder añadir la concatenación, esta se añadirá en forma de matriz una vez dada la forma a un tensor, que será la salida que produzca el generador.

2. Definidas las dimensiones y concatenada la condición, procedemos con la primera convolución traspuesta, que se activará con una *ReLU* y terminará con una normalización de la salida de la capa. La convolución traspuesta actuará sobre una entrada de 3 canales y la condición con 128 filtros de tamaño 3×3 píxeles y un margen de 1.

3. Al igual que en el discriminador, la segunda convolución traspuesta es idéntica a la del segundo paso, salvo por que la entrada es de 128 filtros, y no de 3 canales y una condición.

4. Finalmente, la salida de esta convolución pasa por una tercera convolución, que devuelve la imagen con 128 filtros a 3 canales (los canales de color), con un tamaño de filtros de 3×3 y un margen de 1 píxel.

Podemos ver las arquitecturas del discriminador $\mathcal{D}_0$ y su generador $\mathcal{G}_0$ en 5.2.

Ahora, abordaremos la arquitectura del segundo nivel de la pirámide Laplaciana: $\mathcal{D}_1$ y $\mathcal{G}_1$. De nuevo, para la red generativa que conforman, solo serán necesarias capas de convolución y lineales, normalización de lotes y funciones de activación de tipo *ReLU* y sigmoide. Comencemos con el proceso de $\mathcal{D}_1$.

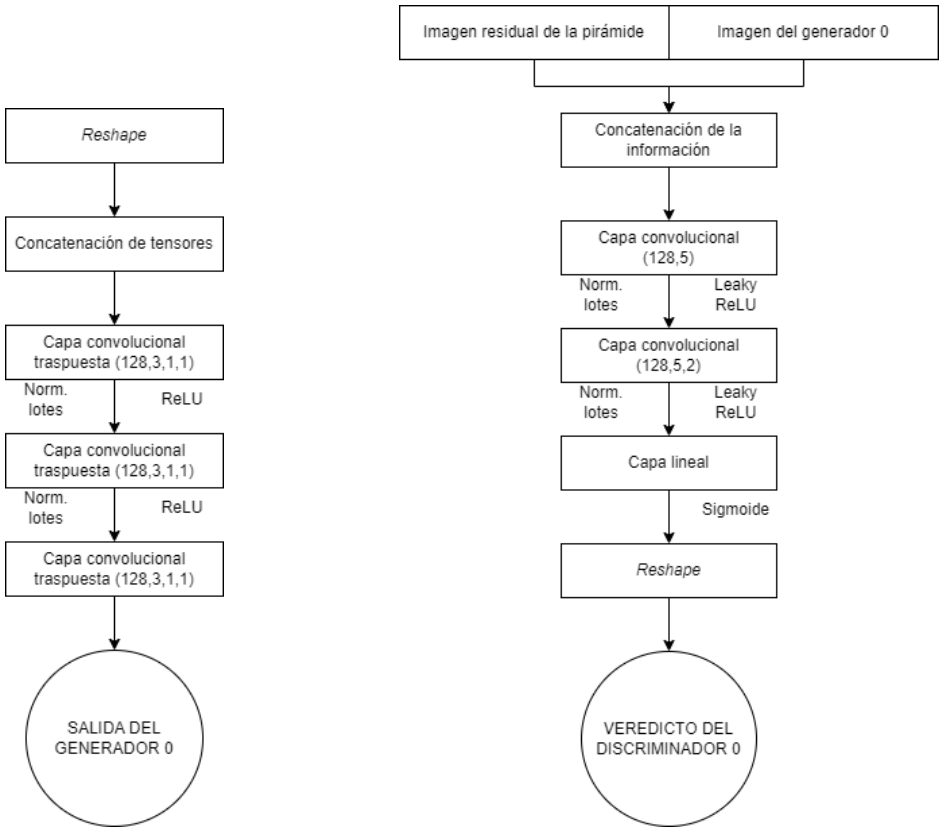
1. Inicialmente, tendremos que concatenar la entrada del discriminador con la condición dada por la imagen de la anterior red.

2. Con esta entrada concatenada, realizaremos una primera convolución, con una entrada de 3 canales y 64 filtros, y un tamaño de convolución 5×5. La salida de esta primera capa se activará con una *Leaky ReLU* y utilizará normalización de lotes.

3. La salida de la anterior capa pasará por una segunda convolución, que tomará el tensor con los 64 filtros y utilizará 64 filtros, un tamaño de 5×5 y un salto de 2 píxeles. De nuevo, se activará y normalizará como en el anterior paso.

4. Tras la segunda convolución, daremos forma a su salida para que la entrada a la última capa tenga la forma adecuada.

5. Finalmente, utilizaremos una sigmoide y una capa lineal para terminar devolviendo una imagen generada, que servirá para condicionar la entrada del nivel conformado por $\mathcal{D}_0$ y $\mathcal{G}_0$.



(a) Arquitectura del generador $\mathcal{G}_0$ (b) Arquitectura del generador $\mathcal{D}_0$

Figura 5.2: Arquitectura de la red generativa 0

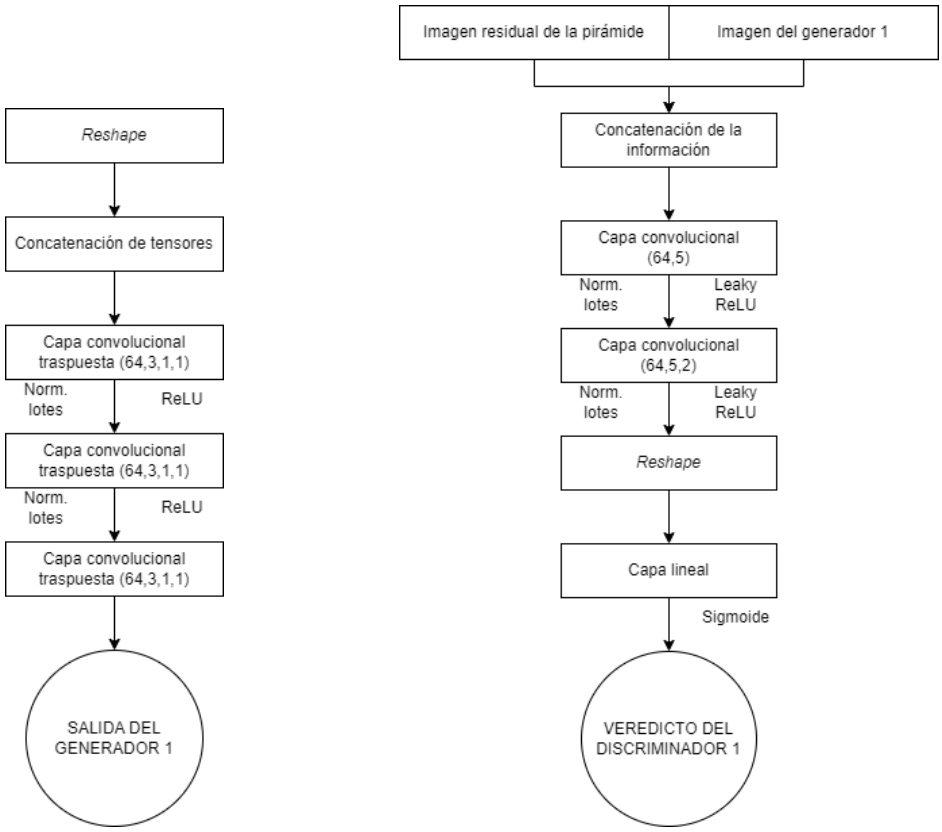
Ahora, para poder realizar el proceso de $\mathcal{G}_1$ hay que utilizar convoluciones traspuestas y capas lineales, además de una redimensión inicial de la salida del nivel anterior de la pirámide.

1. Los pasos son idénticos a lso del nivel 0. Se comienza con una redimensión de la entrada para que tome la forma de la imagen de salida, más pequeña que la imagen final que esperamos producir tras toda la pirámide. Tras el ajuste a la forma deseada, concatenamos la entrada con la condición del nivel anterior.

2. Concatenada la información, esta ha de pasar por la primera convolución traspuesta, activada con una $ReLU$ y cuya salida tendrá normalización de lotes. La convolución traspuesta actuará sobre la entrada de 3 canales y la condición con 64 filtros 3×3 y un margen de 1 píxel.

3. La segunda convolución traspuesta es exactamente igual a la primera, salvo que toma por entrada un tensor con 64 filtros, en vez de una imagen a color.

4. Para terminar, aplicamos una última convolución de 64 filtros a 3 canales (los canales de color), con un tamaño de filtros de 3×3 y un margen de 1 píxel.



(a) Arquitectura del generador $\mathcal{G}_1$ (b) Arquitectura del generador $\mathcal{D}_1$

Figura 5.3: Arquitectura de la red generativa 1

Mostramos en 5.3 la arquitectura utilizada en $\mathcal{D}_1$ y $\mathcal{G}_1$.

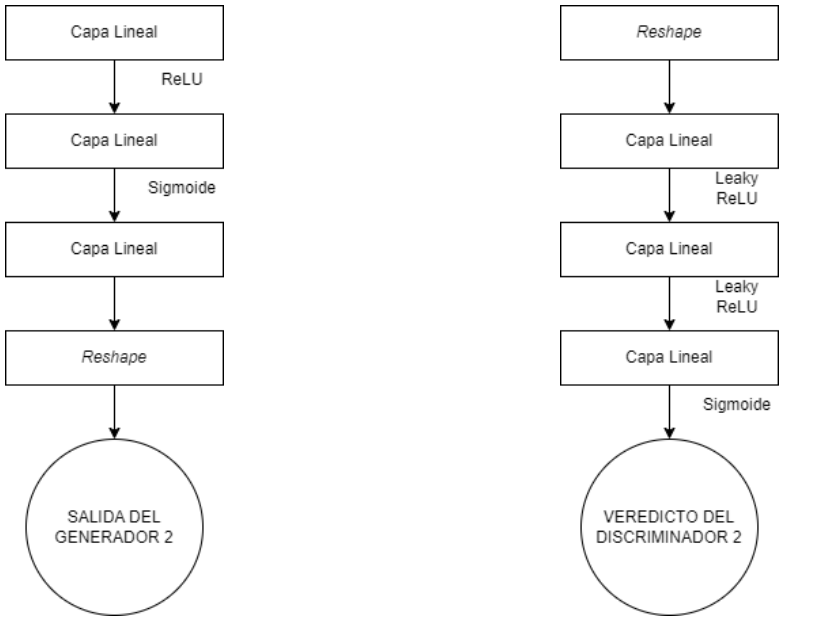
El primer nivel utiliza una arquitectura distinta a la de los niveles descritos en las anteriores páginas, pues realmente lo que realiza es un proceso lineal para producir la primera imagen, que servirá de condición para los niveles siguientes. Estudiemos los pasos de $\mathcal{D}_2$

1. El primer proceso será dar forma a la salida del discriminador, tomando las salida del generador y dándole la forma adecuada, que será una imagen de 8×8 píxeles a color.

2. Esta salida formateada pasará por la primera capa lineal y se activará con una *Leaky ReLU*.

3. La salida de la anterior capa pasará por otra capa lineal activada por otra *Leaky ReLU*.

4. Como paso final, utilizaremos una capa lineal sobre la salida de la anterior capa, salvo que la activación en este caso es una sigmoide, en vez de una función tipo *ReLU*.



(a) Arquitectura del generador $\mathcal{G}_2$

(b) Arquitectura del generador $\mathcal{D}_2$

Figura 5.4: Arquitectura de la red generativa 2

Y definamos la última de las redes de esta pirámide: $\mathcal{G}_2$. Al igual que en su correspondiente discriminador, sólo necesitaremos utilizar capas lineales para poder generar las imágenes.

1. La entrada, inicialmente como vector de ruido gaussiano, será transformada por una capa lineal, activada por una función $ReLU$.

2. La salida de la anterior capa pasará por una segunda capa lineal, activada esta vez por una sigmoide.

3. Esta segunda capa produce una salida que pasará por una última capa lineal, que finalmente se dará forma de matriz para producir una imagen a color.

Se puede ver la arquitectura de las redes de este primer nivel ($\mathcal{D}_2$ y $\mathcal{G}_2$) en 5.4.

5.3. *DCGAN*

La arquitectura de la *DCGAN* es muy similar a la de la *CGAN*, pues lo único que pierde es realmente la capacidad de elección.

De nuevo, como en la sección 5.1, repasaremos las capas que conforman el generador y discriminador, viendo el proceso de generación a partir de un vector de ruido gaussiano de dimensión 100.

Comenzaremos con el proceso del generador

A diferencia de en la *CGAN*, partiremos únicamente de un vector aleatorio de tamaño 100, que pasaremos por una capa densa, que producirá como salida un vector de dimensión $7 \times 7 \times 128$. Ese vector se convertirá en una imagen de tamaño 7×7, gracias a la capa *Reshape*, sobre el que se aplicarán 128 filtros.

El proceso del generador nos lleva por los mismos pasos que antes, hay que generar una imagen cuadrada de 28 píxeles de lado, de modo que simule las imágenes de *MNIST*. Seguiremos un proceso de convolución:

1. Comenzamos el proceso con la imagen de la capa *Reshape*, sobre la que aplicamos normalización de lotes y una activación *ReLU*.

2. Este proceso da la primera entrada de una capa convolucional traspuesta de 128 filtros 5×5 con salto 2.

3. Aplicaremos de nuevo normalización de lotes y una activación *ReLU* sobre el resultado de la convolución anterior.

4. Utilizaremos una capa de convolución traspuesta de 64 filtros, con tamaño 5 y con salto de 2 píxeles.

5. Necesitamos normalizar la salida y activar mediante una *ReLU* de nuevo.

6. Reducimos el salto de la convolución traspuesta a un píxel, y el número de filtros a 32, pero mantenemos el tamaño de los filtros.

7. Aplicamos la última normalización y activación *ReLU*.

8. Utilizamos una última convolución traspuesta de un filtro de tamaño 5 y salto 1.

Con la salida de esta última capa de convolución traspuesta, utilizamos una activación sigmoide, como se realizaba en la arquitectura de la *CGAN*.

Tras este proceso, se ha generado una imagen falsa de *MNIST*, aunque no controlamos la imagen que se ha generado.

Como en el caso de la *CGAN*, ahora hay que explicar la arquitectura del discriminador.

Tomando una imagen o la salida generada por el proceso anterior, seleccionadas de forma aleatoria, realizaremos el proceso de deshacer la convolución para poder obtener el resultado del discriminador. Para conseguirlo, seguimos el siguiente proceso:

1. Activamos la entrada con una activación *Leaky ReLU*.

2. Aplicaremos la primera convolución, con 32 filtros cuadrados de lado 5 y salto 2 píxeles.

3. La salida de la convolución anterior se activa con una *Leaky ReLU*.

4. Utilizaremos otra convolución, de 64 filtros 5×5 y con salto 2.

5. Volveremos a activar la salida con una *Leaky ReLU*.

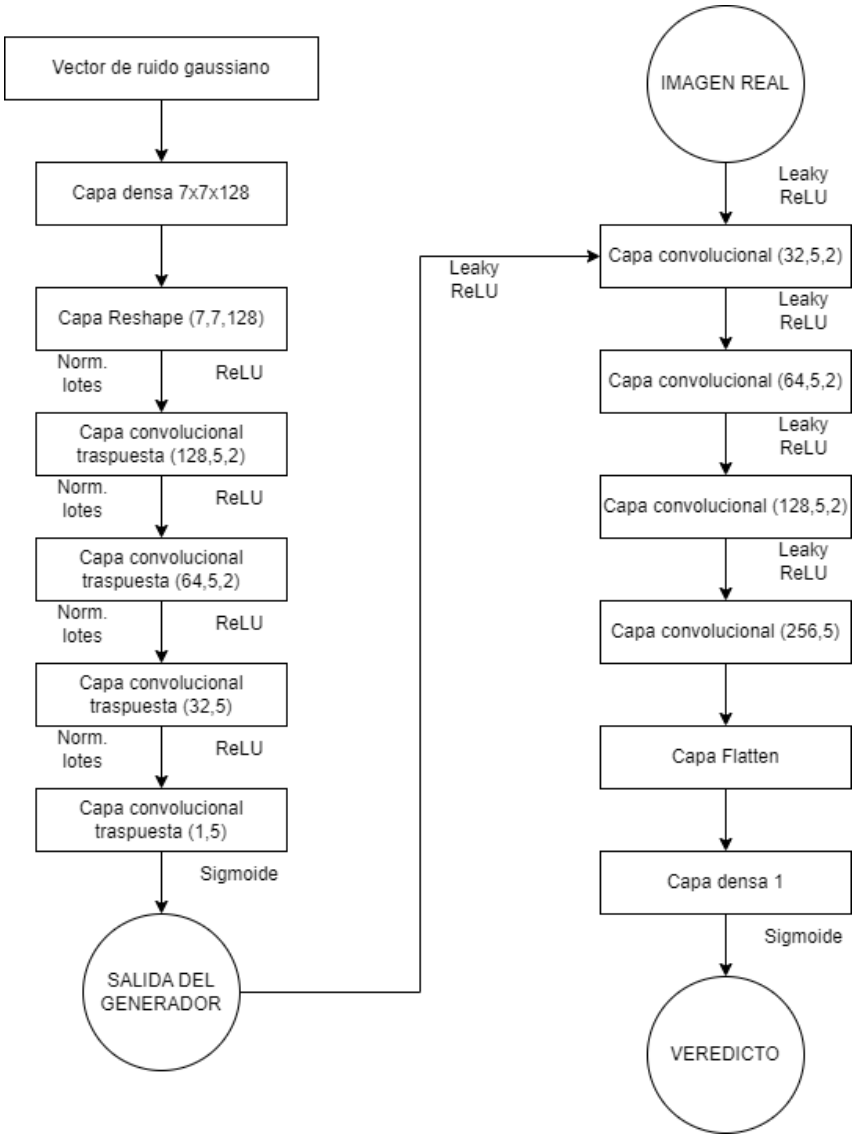


Figura 5.5: Arquitectura completa de una red *DCGAN*

6. Duplicaremos el número de filtros de nuevo, obteniendo una convolución de 128 filtros, con salto 2, de tamaño 5.

7. Se aplicará una *Leaky ReLU* como función de activación.

8. Calculamos la última convolución, pero esta vez con saltos de 1 píxel y 256 filtros de lado 5.

9. Finalizamos con una capa *Flatten* para transformar esta última salida en un vector para interpretar la salida.

El vector lo transformaremos en un elemento mediante una capa densa, que se activará

con una sigmoide, y nos dará el veredicto del discriminador, es decir, si considera que la entrada es falsa o real.

De nuevo, el coeficiente de pendiente es $\alpha = 0,2$ para todas las funciones *Leaky ReLU*.

Se puede observar la arquitectura completa de la red en el diagrama 5.5.

Capítulo 6

Dataset, entrenamiento y resultados

Para el entrenamiento de las redes hemos escogido dos conjuntos de datos, uno sencillo y tradicionalmente utilizado para la gran mayoría de ejemplos (*MNIST*), y otro más complejo, aunque también utilizado en ejemplos sobre la clasificación de imágenes en varias etiquetas (*CIFAR10*).

Por otra parte, el entrenamiento de redes generativas exige, para obtener buenos resultados, de una buena potencia de computación. Para obtener la capacidad necesaria, dado que las operaciones realizan las redes neuronales se basan en operaciones matriciales que se pueden paralelizar bien, se utilizan, actualmente, tarjetas gráficas. La utilización de tarjetas gráficas permite de un entrenamiento más veloz de las redes, para lo que generalmente se utilizan paquetes de *Python* como *CUDA* o *CUML* para lograr buenos resultados.

Para los entrenamientos de nuestras redes hemos utilizado *CUDA* para acelerar el rendimiento del entrenamiento, lo que nos ha permitido realizar 20000 épocas de entrenamiento para las redes *DCGAN* y *CGAN* implementadas, que se han entrenado sobre *MNIST*.

Para la red *LAPGAN*, el resultado y el entrenamiento son distintos, pues se ha utilizado como conjunto de datos *CIFAR10*. Además, el entrenamiento en este caso no depende únicamente del número de épocas por la que se entrena cada nivel de la pirámide, si no en el refinamiento de la pirámide y actualización de los parámetros globales de los niveles de la misma. Los resultados que observaremos para esta red serán como máximo, sobre 100 épocas y 100 actualizaciones de la pirámide (que en el ordenador detallado en la sección 2.3 tomaba alrededor de 20 horas).

6.1. *MNIST*

Se puede obtener más información del *dataset MNIST* en la web [22].

El *dataset MNIST* se originó en [7], como respuesta a un problema generado en una competición de clasificación automática de imágenes.

El *US NIST* o, simplemente, *NIST*, es el Instituto Nacional de Estándares y Tecnología estadounidense. Este instituto produjo un conjunto de datos de entrenamiento y otro de

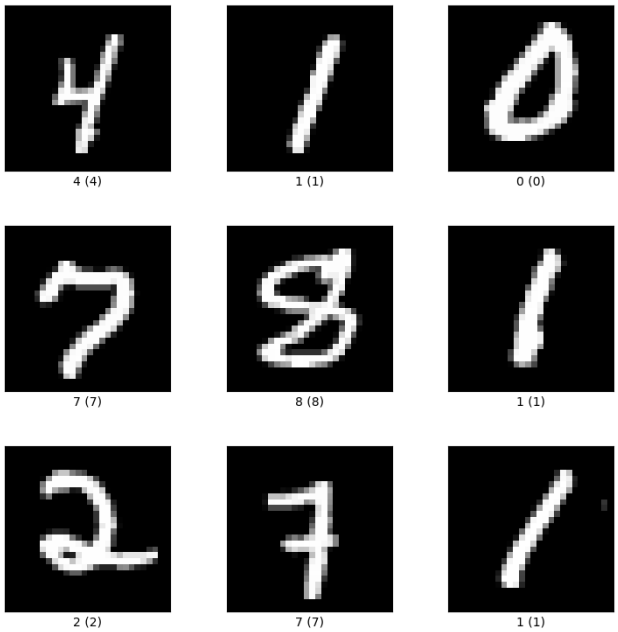


Figura 6.1: Imágenes de ejemplo de *MNIST*, tomadas de https://www.tensorflow.org/datasets/catalog/mnist?hl=es-419

validación llamados *NIST SD-3* y *NIST Test Data 1*, consistentes de caracteres escritos a mano, y organizó una competición para encontrar el mejor algoritmo de clasificación sobre escritura realizada a mano.

Los algoritmos de clasificación de la época demostraron ser muy buenos en la clasificación de los elementos del conjunto de entrenamiento, cuando se tomaba la validación de los mismos en este conjunto también. Sin embargo, al probar con el conjunto de validación dado por el *NIST*, se comprobaba que los resultados del entrenamiento de los algoritmos no eran del todo correctos, pues fallaban.

El problema de este conjunto de datos era el siguiente: el muestreo realizado para obtener los caracteres a mano alzada de ambas partes provenía de distintas distribuciones de probabilidad. Mientras que el conjunto de entrenamiento había sido realizado por trabajadores del censo estadounidense, el conjunto de validación se había generado con muestras de escritura realizada por estudiantes de universidad.

Los autores de [7] se dieron cuenta de que esto generaba, no sólo el problema de el origen de los datos, si no que, como los algoritmos de clasificación se basaban en la minimización del riesgo empírico, donde el número de parámetros se ajusta a la cantidad y complejidad de los datos, y ser las distribuciones distintas, los algoritmos no servían sobre la partición dada. Con este problema a la vista, la solución que se propuso fue la siguiente: había que mezclar de nuevo todos los datos y reparticionar el conjunto completo

lo que solventaba el problema de las dos distribuciones, pues la mezcla genera una nueva y única distribución de los datos.

De este modo se obtenía *Modified NIST training and test sets*, o, comúnmente, *MNIST*, un conjunto de datos que permitía aplicar los algoritmos de la época en la clasificación de caracteres escritos.

Abandonando el carácter histórico del *dataset*, centrémonos en sus cualidades.

Este conjunto de datos dispone de 70000 dígitos escritos a mano alzada, de los cuales 60000 se dedican al conjunto de entrenamiento y 10000 al de validación, con el objetivo de tener un conjunto de datos sencillo para el reconocimiento de patrones en imágenes.

Los dígitos de este *dataset* están normalizados en tamaño y centrados en la imagen, lo que facilita el reconocimiento de patrones. Esta es una cualidad que nos permite acelerar el entrenamiento de nuestras redes, pues eliminamos la necesidad de realizar preprocesado sobre las imágenes, y también permite esquivar el formateado de las mismas para que se adecúen a nuestra red. Todas las imágenes del conjunto son de tamaño 28×28 píxeles, lo cual también nos ayuda en el entrenamiento de nuestras redes, dado que no será necesario reescalar las imágenes que no se ajusten al tamaño de entrada, si no que, como todas tienen el mismo tamaño, basta únicamente con definir bien el tamaño de la entrada.

Aunque las redes que implementamos no se han diseñado para el reconocimiento de patrones, estrictamente hablando, es en esta capacidad de reconocer cualidades de las imágenes en la que se basa el aprendizaje de las redes generativas, pues lo que buscan es aprender las cualidades significativas de los elementos del conjunto, de modo que se puedan reproducir en nuevos elementos.

6.2. *CIFAR10*

Podemos encontrar parte de la información de esta sección en [20], que relata el uso del *dataset CIFAR10* para el aprendizaje automático.

El nombre del *dataset* proviene de *Canadian Institute For Advanced Research* y el número de clases distintas que posee el conjunto de imágenes.

Principalmente, *CIFAR10* es un conjunto de 60000 imágenes de tamaño 32×32 separadas en 10 clases de objetos distintos: aviones, automóviles, pájaros, gatos, ciervos, perros, ranas, caballos, barcos y camiones. Estas clases son mutuamente excluyentes entre sí, pues para diferenciar las clases de automóvil y camión se tomaron las precauciones necesarias para evitar el solapamiento entre ambas.

De las 60000 imágenes que conforman el *dataset*, 50000 son imágenes de entrenamiento repartidas en 5000 por cada una de las clases anteriores, y 10000 imágenes de prueba, 1000 por cada clase. La división de estos conjuntos permite utilizar 5 lotes de entrenamiento de 10000 imágenes con un lote de *test* del mismo tamaño. Aunque el conjunto de prueba siempre contendrá 1000 imágenes de cada una de las clases, los lotes de entrenamiento contienen un número distinto de imágenes de cada clase, de modo que pueda darse que haya más imágenes de un tipo que del resto en cada lote de *train*.

Relatemos la forma en que se recolectó y etiquetó este conjunto de datos. Para la

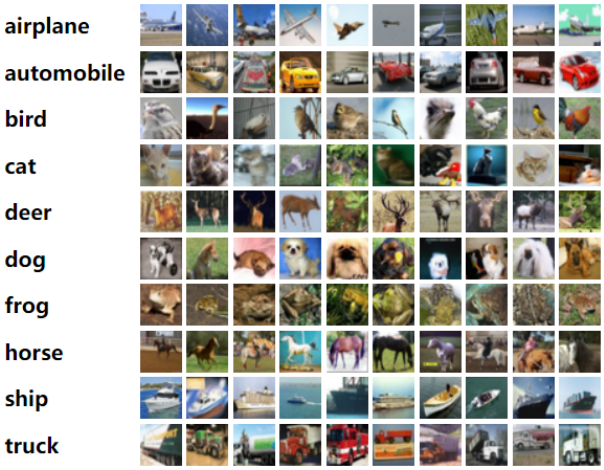


Figura 6.2: Imágenes de ejemplo de *CIFAR10*, tomadas de https://www.cs.toronto.edu/~kriz/cifar.html

recolección de las imágenes se realizaron búsquedas de los elementos y se etiquetaron con respecto a la búsqueda realizada. Con el objetivo de obtener una mejor división de las imágenes en clases, se pagó a estudiantes para que etiquetaran 6000 imágenes dentro de cada una de las 10 clases que se han mencionado. Además, no era necesario clasificar las imágenes desde cero, si no que únicamente se necesitaba filtrar aquellas que habían recibido la imagen incorrecta.

Cada uno de los estudiantes fue provisto de una clase y se les pidió que examinaran las imágenes bajo su etiqueta siguiendo unas directrices:

1. El nombre de la etiqueta de la imagen ha de responder claramente a la pregunta: ¿Qué se observa en esta imagen?

2. Las imágenes clasificadas han de ser fotorrealistas. No se aceptarían dibujos como imágenes válidas para los lotes de entrenamiento.

3. La imagen debe contener únicamente una instancia preeminente del objeto descrito por la etiqueta.

4. El objeto puede estar parcialmente oculto, o visto desde un ángulo inusual, pero ha de poderse reconocer mediante la etiqueta por la persona que revisa las imágenes.

Con estos criterios se conseguía un conjunto de datos tamizado, pero también era necesario reducir los duplicados de imágenes. Para ello se realizó una revisión del conjunto de imágenes aplicando la norma de L^2.

Tras todos estos pasos se obtenía un *dataset* que tenía 5000 imágenes de entrenamiento en cada una de las 10 clases, y 1000 imágenes de prueba para cada una de las clases, que es el conjunto de imágenes que hemos utilizado.

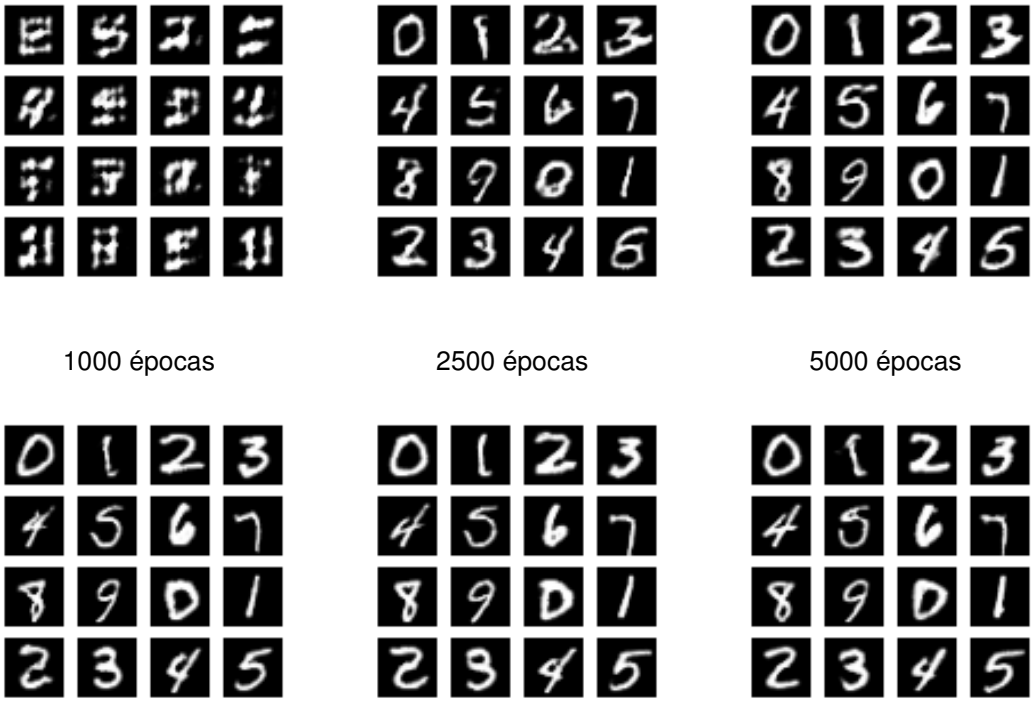


Figura 6.3: Resultados obtenidos del entrenamiento de la *CGAN*

6.3. Resultados de la *CGAN*

Para entrenar esta red, se han determinado unas 20000 épocas de entrenamiento para la red completa, lo cual produce los resultados de 6.3.

En la figura podemos ver distintos resultados producidos por la red, en distintas épocas del entrenamiento. Con el objetivo de producir distintos dígitos, generamos 16 imágenes que se muestran en forma de matriz 4×4, que están ordenados de 0 a 9 y repiten aquellos del 0 al 5. Pese a que sólo se muestran 6 imágenes como resultados de la red, se puede extraer información de valor de estos resultados producidos. Inicialmente, tras 1000 épocas, podemos ver como se empieza a concentrar la masa de píxeles blancos (que posteriormente serán los dígitos) en el centro de las imágenes. Tras 2500 épocas, ya se obtienen dígitos manuscritos reconocibles, aunque no todas las imágenes generadas parecen dígitos. Por ejemplo, puede haber dudas en si 6.4 son dígitos o no.

Tras las 2000 épocas, los resultados son más uniformes, y se puede observar como obtenemos, al finalizar el entrenamiento, las imágenes siguiendo la sucesión 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5 (fig. 6.3).

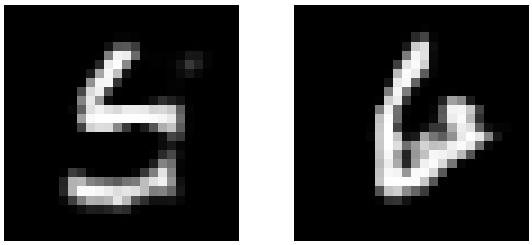


Figura 6.4: Resultados ambiguos de la *CGAN*

Figura 6.5: Resultados obtenidos del entrenamiento de la *DCGAN*

6.4. Resultados de la *DCGAN*

Al igual que en el caso de la *CGAN*, cada 500 épocas se han producido 16 imágenes, con el inconveniente de no tener control sobre los dígitos que produce esta red.

De este modo, se pueden observar ciertos aspectos problemáticos del planteamiento de las *GAN* originales.

Si observamos los resultados que se presentan en 6.5, podemos darnos cuenta de que al principio se tienen resultados parecidos para las 1000 y 2500 épocas. Sin embargo, según

avanza el entrenamiento de la red, se puede observar cierta evolución en la calidad de los dígitos generados, además de la transformación de algunos de ellos en otros distintos. Si nos fijamos en las primera y tercera imágenes, podemos ver más claramente la transformación de algunos dígitos en otros distintos. La primera imagen, que representa inicialmente lo que recuerda a un cuatro, tras 20000 épocas se ha convertido en un uno. La tercera no se puede prever que número es al principio, si un dos con un lazo o un ocho, pero finalmente se convierte en un ocho.

Esto demuestra que la imposición de etiquetas, en las implementaciones realizadas, aumenta la precisión temprana del generador a la hora de producir salidas de una forma ordenada. También es un ejemplo de colapso modal; como el 2 con un lazo y el 8 se parecen, en este caso se ha colapsado sobre el 8. Igualmente pasa en el caso del 4 que se transforma en 1.

6.5. Resultados de la *LAPGAN*

Para el entrenamiento y la implementación, nos hemos basado en la implementación que se lleva a cabo en [10], adaptada a nuestras necesidades de *CIFAR10*.

En este caso el entrenamiento de la red es más complicado, porque no sólo hay que entrenar las tres redes generativas que conforman la pirámide, si no que hay que entrenar su comportamiento conjunto como pirámide laplaciana.

Para el entrenamiento de la red, como podemos ver en los resultados de 6.6, hemos entrenado la red 10, 20, 50 y 100 épocas, y realizado 100 actualizaciones de los parámetros de la pirámide. Lo anterior quiere decir que se presentan resultados del entrenamiento de la red completa en 10, 20, 50 y 100 épocas, y que dentro de cada época de entrenamiento se han realizado 100 épocas de entrenamiento para cada una de las redes generativas que conforman los niveles de la pirámide.

También podemos observar como se escala, desde el tamaño menor hasta los 32×32 píxeles finales, las imágenes que se han ido produciendo en cada uno de los niveles de la pirámide. Hemos elegido producir 32 muestras de la misma generación, entre las cuales hay pocas diferencias según aumenta el número de épocas y tamaño de la imagen, aunque son perceptibles si se examinan con cuidado.

Como la imagen final se produce cuando termina el entrenamiento de la red, los resultados de generación de las distintas épocas no son la misma imagen, si no que cambia; algo fácilmente observable si se tienen en cuenta los patrones de color de los distintos entrenamientos.

Debido a que no se disponía de la suficiente potencia de computación, no se han podido producir resultados de mayor calidad, o más parecidos a los del *dataset* de imágenes *CIFAR 10*. Aun así, se considera que los resultados producidos son suficientes para la demostración del aprendizaje de la red.

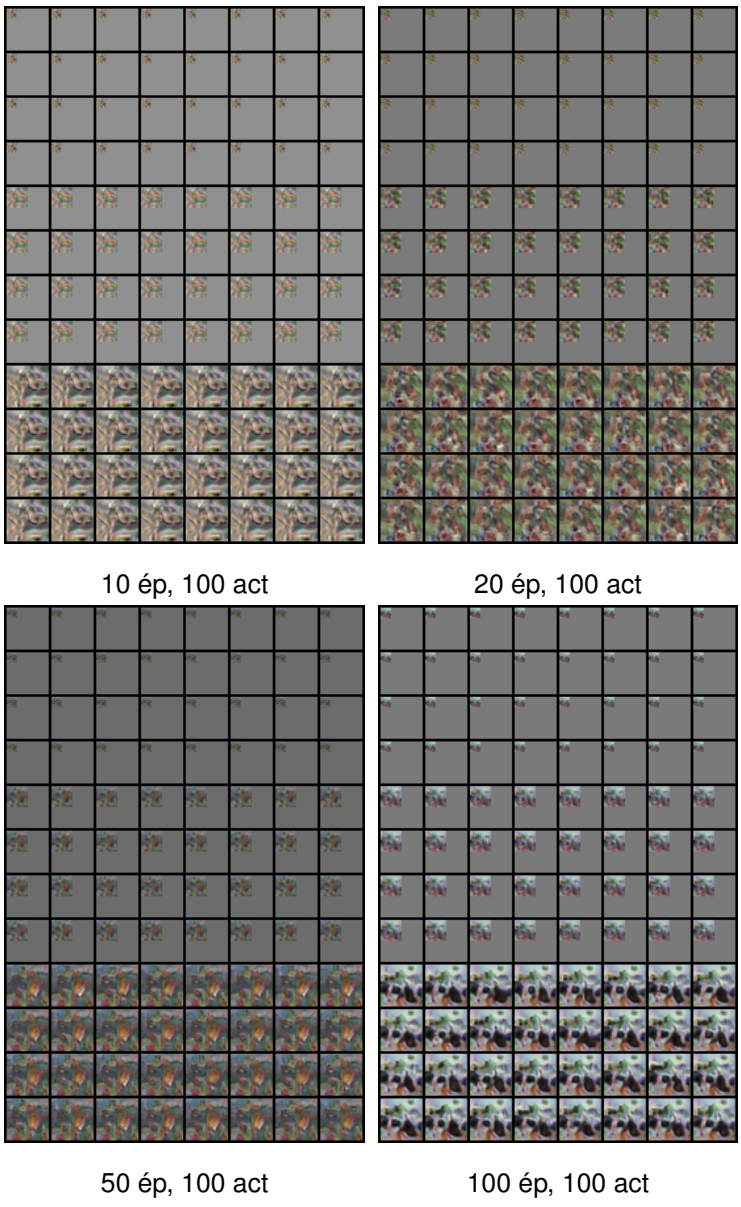


Figura 6.6: Resultados obtenidos del entrenamiento de la *LAPGAN*

Capítulo 7

Conclusiones y trabajo futuro

Damos pie al último capítulo del trabajo, las conclusiones, donde veremos la consecución de los objetivos planteados y lo aprendido, además de plantear las líneas de estudio futuras para completar la comprensión de los modelos actuales.

7.1. Conclusiones

Tras haber finalizado con la redacción, estudio e investigación sobre el funcionamiento de las redes generativas, es necesario ver si se han alcanzado los objetivos planteados para este trabajo en 1.2.

- El primer objetivo era introducir el funcionamiento de las redes adversariales para la generación de elementos, objetivo que se ha alcanzado claramente mediante la sección 4.1.
- Respecto a las similitudes y diferencias de estas redes con sus precursoras, se ha mencionado que el trabajo de las redes convolucionales y profundas anteriores a las generativas se utilizaban para la clasificación de imágenes, y el salto que supusieron las redes generativas, pues abstraen la información de los datos para generar nuevos ejemplos similares a estos.
- Se ha detallado también la arquitectura teórica y la arquitectura que se ha seguido para implementar tres modelos distintos de redes generativas adversariales, como son la *DCGAN*, *CGAN* y *LAPGAN*.
- También se han entrenado los modelos y se han mostrado sus resultados, aunque en el caso de la *LAPGAN* no se han obtenido resultados suficientemente buenos, únicamente aceptables.
- Finalmente, se ha hablado del estado del arte y de los nuevos modelos de atención, que fundamentan los modelos actuales y nacieron como solución a la utilización abusiva de las capas convolucionales en redes generativas.

El listado anterior muestra que prácticamente todos los objetivos se han cumplido sin problemas significativos, aunque también muestran que en algunos casos los resultados producidos no son tan buenos como se podría esperar.

El principal inconveniente sobre los resultados producidos ha sido el desarrollo en local, pues un desarrollo en la nube, como podría haber sido con *Google Colab* o *Microsoft Azure* podría haber supuesto una mejora significativa en las velocidades de entrenamiento para las redes implementadas. Al desarrollar las propuestas en el propio ordenador se ha dispuesto de un hardware limitado y esto ha producido entrenamientos de larga duración debido a una baja capacidad de computación. Esto es parte del margen de mejora que veremos en la siguiente sección de este capítulo.

Este proyecto ha servido como canal para observar si las habilidades y técnicas aprendidas durante el transcurso de la carrera se habían aprendido, además de plantear nuevos retos para el alumno. Trabajar un tema que requiere tiempo para investigar y tiempo para implementar ha resultado de utilidad para aprender sobre la división del tiempo de trabajo y sobre como una buena planificación de las tareas puede ser decisiva a la hora de realizar un proyecto. El trabajo ha servido también para reforzar el conocimiento sobre aprendizaje automático que ya se poseía, además de ampliarlo con nuevos conceptos, como lo es el entrenamiento adversarial de redes neuronales. Entre algunas de las habilidades que ha permitido perfeccionar el desarrollo del proyecto están las habilidades de redacción a LaTeX, la lectura y comprensión de artículos científicos, el espíritu de investigación y la comprobación de la adecuación de resultados.

7.2. Trabajo futuro

Como ya se ha comentado al hablar del estado del arte, hay arquitecturas más novedosas en el campo de la generación de contenido como puede ser la arquitectura *Transformer*. Esta sería la continuación natural del estudio de este trabajo, pues una vez comprendido el funcionamiento de las *GAN*, el siguiente paso sería entender las *Cross-Domain GAN*, los modelos de aprendizaje por refuerzo, los modelos de atención (donde entran los modelos *Transformer*) y la multimodalidad.

También sería parte del trabajo futuro entender el funcionamiento de *Google Colab* y *Microsoft Azure*, pues son herramientas en la nube que permiten a estudiantes disponer de mayor capacidad de computación en caso de ser necesario. En el caso de *Microsoft Azure*, se tienen 100 € en créditos mediante el *GitHub Student Pack*, y en el caso de *Colab*, se dispone de un método gratuito de uso con cuadernos de *Jupyter* y aceleración de ejecución mediante GPUs y TPUs.

Por otra parte, también se puede estudiar el impacto que ha tenido el desarrollo de las redes generativas en tareas de clasificación generales, pues se ha comprobado que algunas arquitecturas de redes adversariales han ayudado en problemas de clasificación de *NLP*.

Dentro de las posibilidades de trabajo futuro también estarían el estudio de la inferencia variacional, el *Evidence Lower Bound* o el funcionamiento de los autocodificadores. Pese a que estos temas son de alto interés en la actualidad, no se podrían introducir en

un trabajo sin conocer previamente los conceptos que se han presentado y muchos otros relacionados con el cálculo variacional, la optimización de problemas y herramientas de probabilidad y estadística que son suficientemente extensas como para ocupar un trabajo por sí solas.

Lo anterior, que escapa al alcance de este proyecto, es especialmente útil para obtener conclusiones a priori sobre el funcionamiento de las redes actuales, pues en algunos casos son capaces de prever el colapso modal o fallas potenciales en el modelo, como puede ser la no convergencia del mismo. En algunos casos también plantean novedosas arquitecturas que generan resultados parecidos a los obtenidos mediante redes generativas, pero con sus propias particularidades. Si se presentaran estos conceptos, el trabajo se extendería demasiado, tanto en el tiempo como en el texto, y se ha preferido mantener una explicación con menor detalle matemático pero comprensible.

Concluimos entonces con este trabajo, que ha presentado los conceptos de red generativa adversarial y algunas de sus arquitecturas.

Parte III

Apéndices

Apéndice A

Manuales de ejecución e instalación

A.1. Instalación y ejecución de LaTeXen *VS Code*

Visual Studio Code es una herramienta cómoda para el desarrollo pues permite integrar distintos lenguajes de programación y paquetes de una forma sencilla y ágil, todo con el mismo entorno de desarrollo. Esto lo transforma en una herramienta versátil con la que trabajar, y en particular ha resultado bastante útil para redactar la memoria del trabajo que se ha presentado.

El proceso que se ha seguido para realizar la instalación de los elementos necesarios para manejar LaTeXen *VS Code* es el mismo que el de [26].

La instalación de los elementos siguientes se realizó en un entorno *Windows 10*, pues *Strawberry Perl* está diseñado para entornos de *Microsoft*.

Lo primero que hemos de hacer es descargar tres herramientas en nuestro ordenador: *Visual Studio Code*, *Miktex* y *Strawberry Perl*. Con ellas descargadas, instalaremos *Miktex* primero, siguiendo los pasos del instalador y escogiendo la carpeta en la que queramos realizar la instalación. *Miktex* permite el uso del sistema LaTeXen nuestra computadora, lo que servirá para poder redactar la memoria utilizando *Visual Studio Code* como editor. Tras ello, será necesario añadir el directorio en el que se ha instalado a las variables de entorno del sistema. Para ello, conociendo la ruta de instalación, buscaremos con el buscador del sistema: `Variables de entorno` y pulsaremos la tecla *Enter*.

Como podemos ver en A.1, una vez veamos el menú y hagamos *click* en las variables de entorno, solo será necesario añadir una nueva, pulsando el botón resaltado y añadir el directorio en que se encuentra la instalación de *Miktex*. En caso de instalar *Miktex* en un directorio que ya estuviera dentro de las variables del entorno, esto no sería necesario.

Una vez hemos finalizado con la instalación de *Miktex*, procederemos a instalar *Strawberry Perl*. Su instalación se basa en seguir las instrucciones del instalador descargado. Del proceso sólo es destacable el hecho de poder realizar la instalación en el directorio que se prefiera, cuestión de gusto personal del usuario que lo instale.

Tras finalizar esta instalación, ejecutaremos el instalador de *VS Code*, y una vez instalado (en la ubicación que prefiramos), se instalará la extensión, desde el menú lateral, de *LaTeX Workshop*. En la figura A.2 podemos ver resaltado el icono de las extensiones. Una

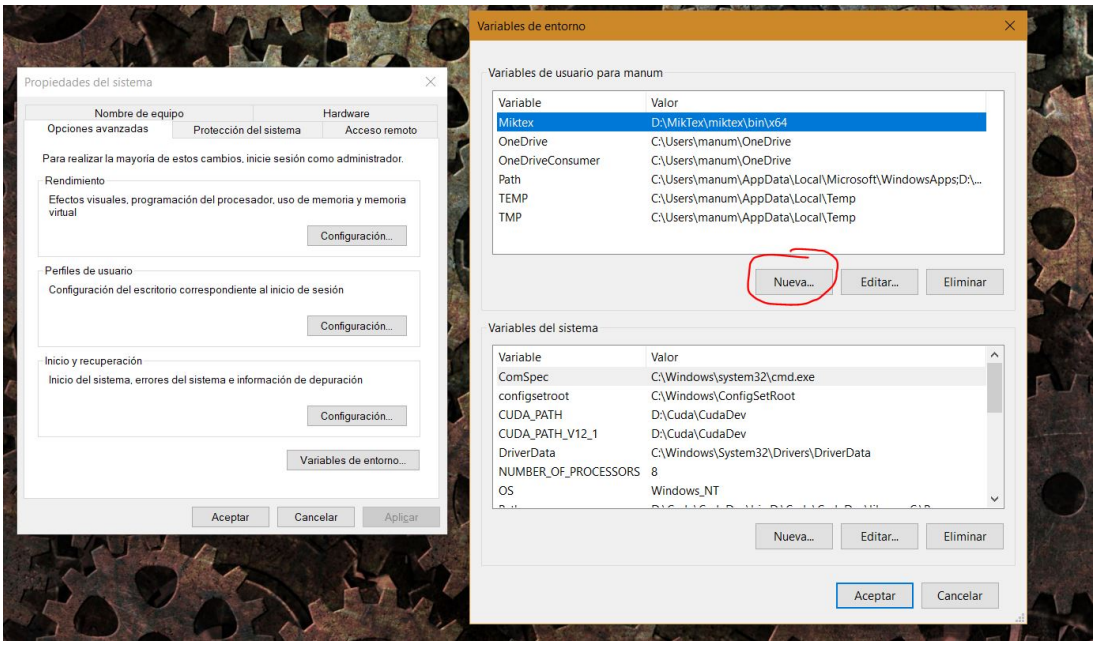


Figura A.1: Variables de entorno y creación de nuevas

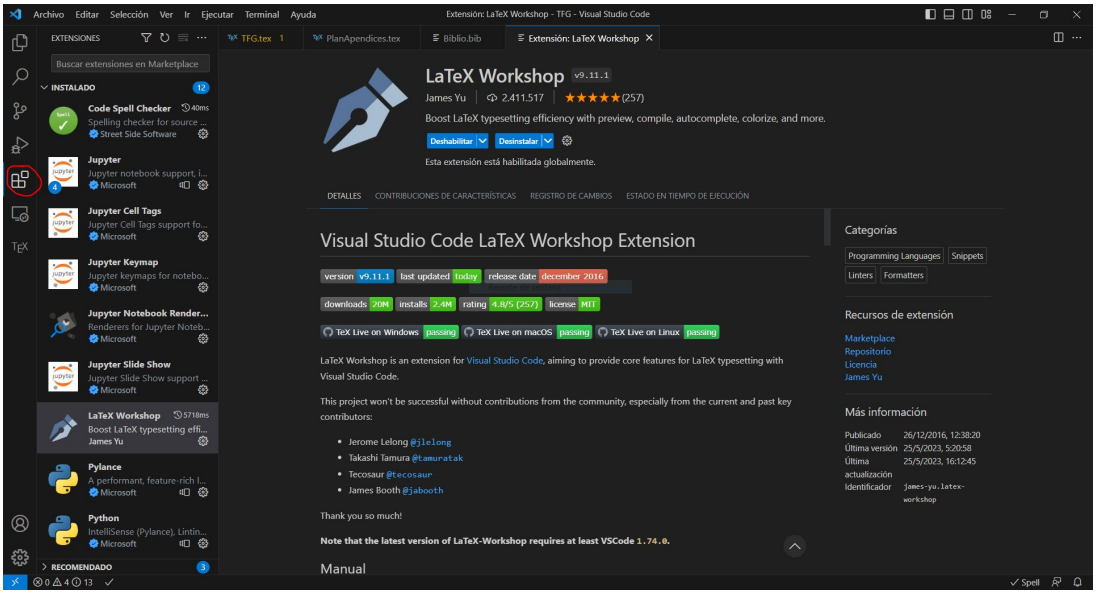


Figura A.2: Instalación de *LaTeX Workshop*

vez seleccionado, usaremos la barra de búsqueda para encontrar la extensión y haremos *click* sobre ella, y también en el botón de instalación, una vez se muestre la información sobre la extensión.

Con esto termina este manual para poder trabajar con LaTeXdesde *Visual Studio Code*.

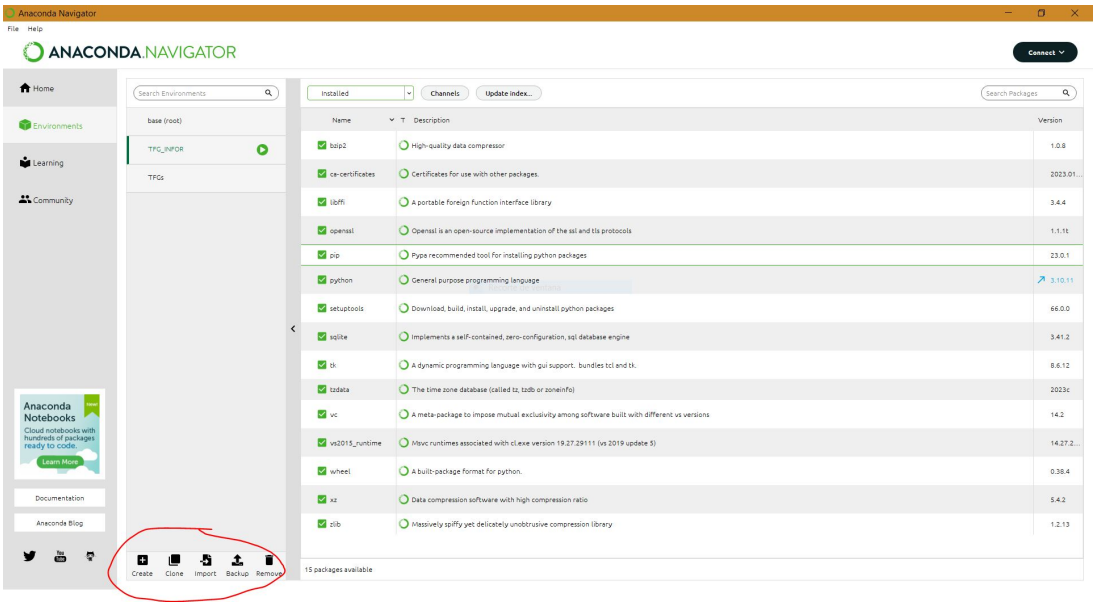


Figura A.3: Creación de un entorno de trabajo en *Anaconda*

A.2. Instalación y ejecución de *Anaconda Navigator* y *Spyder*

Se ha escogido utilizar *Anaconda Navigator* y *Spyder* por estar ya familiarizado con estas herramientas gracias a las asignaturas vistas en el grado. Estas herramientas ayudan a la instalación de ciertos paquetes de *Python* y al desarrollo de *scripts* que permiten la implementación de las redes generativas mostradas en el capítulo 4. Aún así, para tener una instalación personalizada, sobre todo de cara a las versiones de *PyTorch* y *CUDA*, los paquetes se han instalado mediante la consola de anaconda, con el comando `conda -install`.

Para instalar *Anaconda*, visitaremos su página web, y descargaremos el instalador. Ejecutando el instalador, y seleccionando las configuraciones recomendadas y el directorio en el que guardar los archivos principales, instalaremos *Anaconda*. Tras instalarlo, ejecutaremos el programa que lo inicia y crearemos un entorno de trabajo para el desarrollo, de modo que una mala instalación de paquetes, o que se den errores graves, no puedan causar daños a la instalación base.

Para crear el entorno, es suficiente con iniciar la aplicación de *Anaconda* y pulsando en la pestaña izquierda de *Environments*, hacer *click* en *Create* (esto se puede ver en la figura A.3).

Al trabajar en un entorno, podemos personalizar también los paquetes que se instalan, de modo que dispongamos de aquellos que son necesarios únicamente.

Cuando haya terminado la creación del entorno de *Anaconda Navigator*, seleccionaremos este para realizar la instalación de *Spyder*, que se hace desde la aplicación. Para ello, desde la pestaña principal (que aparece cuando se inicia la aplicación), buscaremos

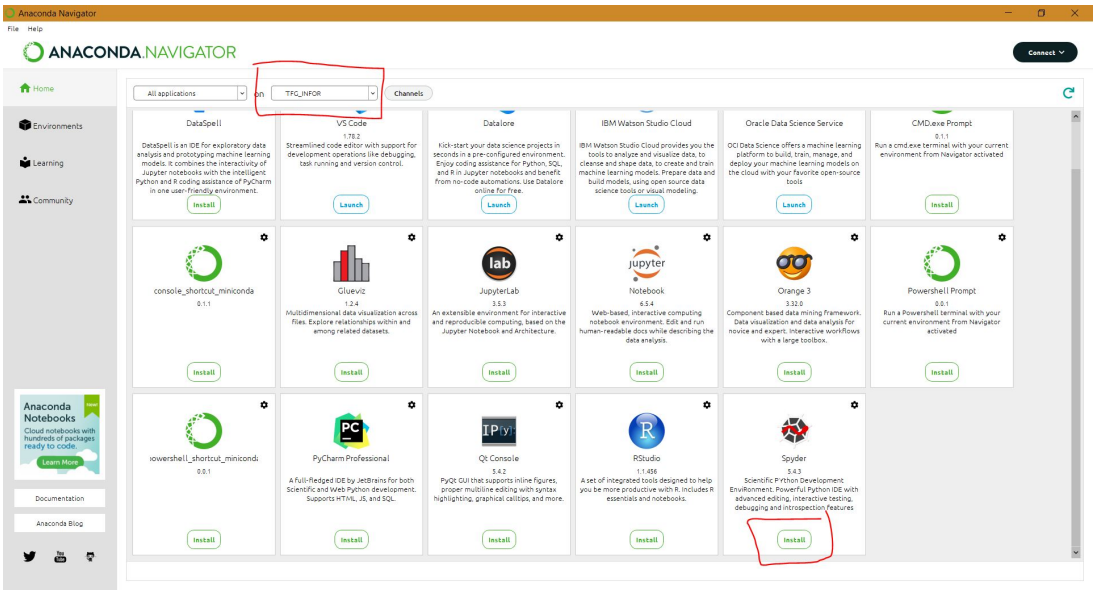


Figura A.4: Instalación de *Spyder* en un entorno

la casilla de *Spyder* y pulsaremos el botón de *Install*. Este botón es suficiente para que es instale el *IDE* y sus posibles dependencias y paquetes, aunque es importante asegurarnos de estar instalando en el entorno creado previamente. Para saber en que entorno trabajamos, basta fijarse en la segunda pestaña desplegable de la parte superior izquierda.

Una vez instalado *Spyder*, tendremos prácticamente preparadas las herramientas necesarias para el desarrollo de los experimentos, pues faltará instalar los paquetes y dependencias necesarios para implementar las redes generativas.

Se han resaltado en la figura A.4 la pestaña que muestra el entorno y el botón de instalación.

A.3. Instalación de *TensorFlow* y *Keras*

Hemos hablado ya de *TensorFlow* y *PyTorch* en la sección 1.3, así que damos por informado al lector sobre los paquetes que va a instalar. En resumen, ambos paquetes sirven para el desarrollo de métodos de *Deep Learning* de forma sencilla.

Para instalar los paquetes *TensorFlow* y *Keras* utilizaremos la interfaz gráfica de *Anaconda*. De nuevo, iremos al apartado de *Environments* y seleccionaremos el entorno que hemos creado, en el cual hemos instalado *Spyder*.

Una vez seleccionado el entorno de trabajo, seleccionaremos la opción *All* del desplegable que muestra los paquetes instalados en el entorno, y utilizaremos la barra de búsqueda en la parte derecha para encontrar el paquete *TensorFlow* (escrito así, en minúsculas).

Dentro de la lista que aparece, seleccionaremos el paquete con el mismo nombre y pincharemos sobre el botón *Apply* de la esquina inferior derecha. Esto buscará las dependencias del paquete y nos las mostrará en pantalla, previo a la instalación. Una vez se

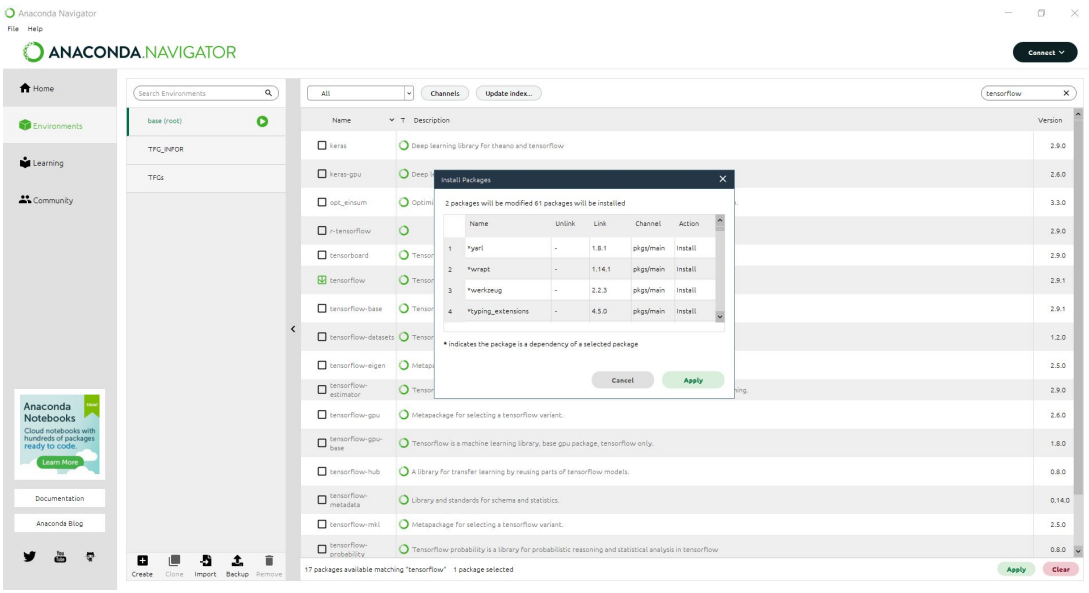


Figura A.5: Instalación de paquetes en *Anaconda*

muestren los paquetes (debería ser parecido a la figura A.5), volveremos a dar al botón *Apply* del cuadro modal de las dependencias y esperaremos a que finalice la instalación.

Tras instalar *TensorFlow*, nos aseguraremos de que *Keras* también se ha instalado, realizando una búsqueda como hemos hecho para encontrar el paquete de *TensorFlow*. En caso de que no se haya instalado, basta realizar el mismo proceso para instalar el paquete *Keras*.

Siguiendo el mismo proceso, es aconsejable instalar también los paquetes de *Pandas*, *graphviz*, *Pydot*, *numpy* y *matplotlib*, pues pueden ser necesarios para la correcta visualización de resultados y salidas de consola producidas por las implementaciones.

A.4. Instalación de *PyTorch* y *CUDA*

Para la instalación de *PyTorch* seguiremos un procedimiento distinto, pues se ha realizado la instalación de uns versión específica para trabajar con *CUDA*. Lo primero será instalar el *Anaconda cmd prompt*, cuyo procedimiento de instalación es el mismo que el de *Spyder*.

Una vez instalada la consola de comandos de *Anaconda*, instalemos los paquetes con los que vamos a trabajar. Se puede ver con más detalle en https://pytorch.org/get-started/locally/, en la sección de *start locally*. Seleccionaremos la versión que queramos instalar de *PyTorch*, nuestro sistema operativo, la versión de *CUDA* a instalar y copiaremos el comando (fig. A.6).

Con el comando copiado, iniciaremos la consola de comandos de *Anaconda*. Para ello pulsaremos en el botón *Start* del elemento *Cmd Prompt* de la pantalla principal de la aplicación de *Anaconda*. Pegaremos el comando (hay que seleccionar que queremos utilizar

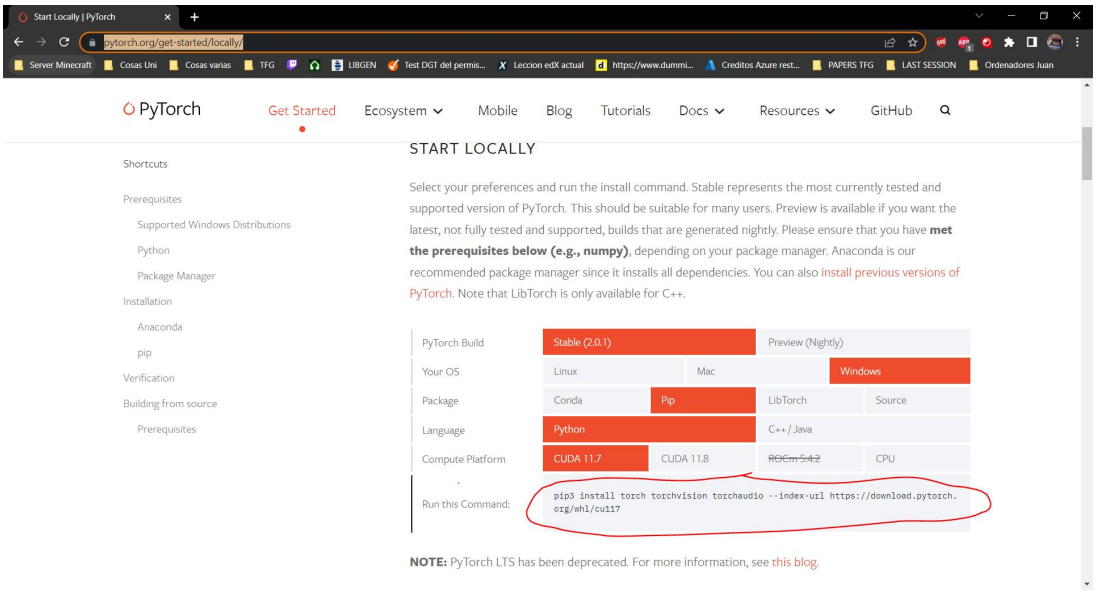


Figura A.6: Comando a copiar para la instalación de *PyTorch*

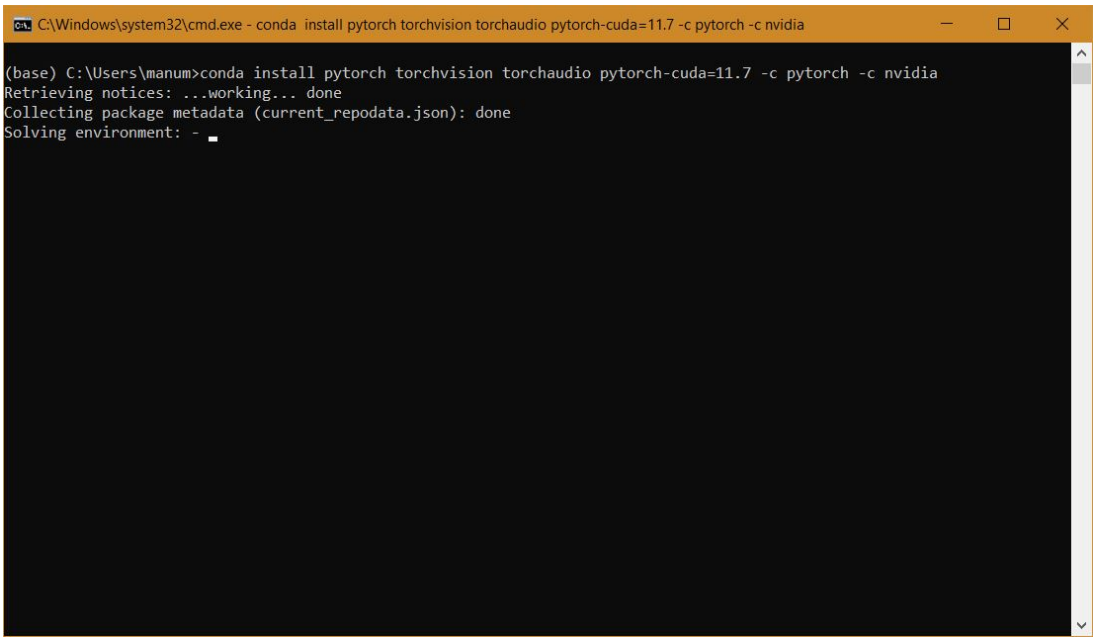


Figura A.7: Ejecución del comando para la instalación de la última versión de *PyTorch* con *CUDA*

conda a la hora de obtener el comando de instalación) en la consola y pulsaremos *Enter* si no se ha ejecutado al pegarlo. Podemos ver un ejemplo de ejecución en la figura A.7

Es importante tener en cuenta que se presupone que el lector dispone de un dispositivo compatible con *CUDA*, pues si no la instalación dará error o no permitirá la ejecución de los scripts mediante *CUDA*. Si no se dispone de un dispositivo compatible con *CUDA*, al seleccionar las opciones para la instalación de *PyTorch*, seleccionaremos CPU, en vez de

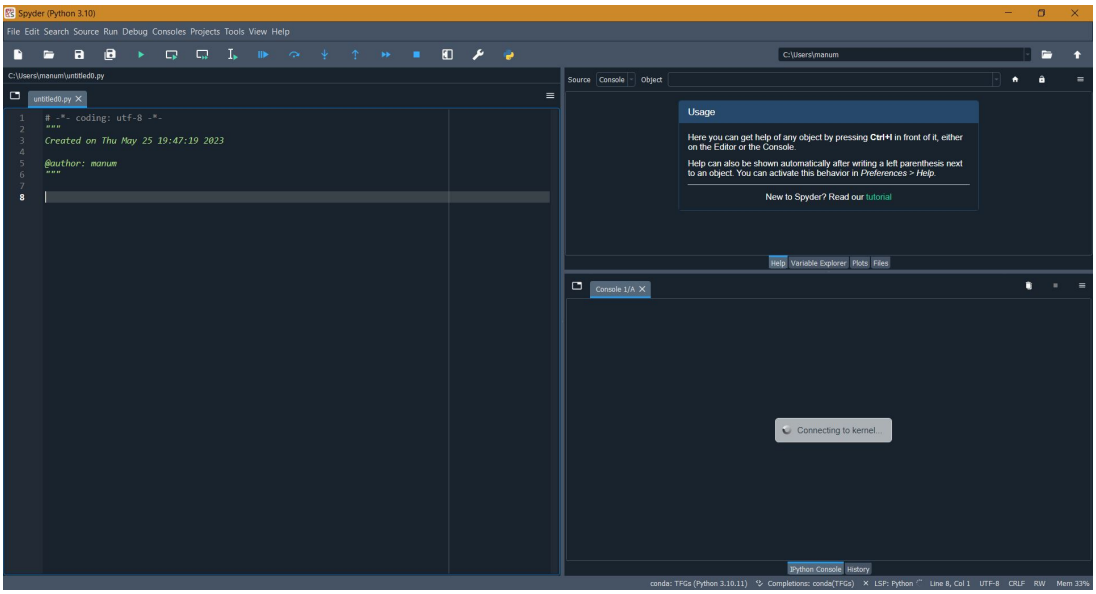


Figura A.8: Pantalla inicial de *Spyder*

una versión de *CUDA*.

Una vez ejecutado el comando, cuando hayan cargado las dependencias de los paquetes, la consola preguntará al usuario si desea instalar los paquetes. Para continuar con la instalación, es suficiente con teclear `y` y pulsar la tecla *Enter*.

Cuando haya terminado la instalación de los paquetes, el lector estará preparado para iniciar *Spyder*, importar los *scripts* implementados y ejecutarlos si desea repetir los experimentos realizados en este trabajo.

A.5. Manual de ejecución de los *scripts*

Finalmente, queda mostrar como importar y ejecutar el entrenamiento de las implementaciones realizadas.

Ejecutaremos la aplicación *Anaconda Navigator*, y cuando veamos la interfaz, lanzaremos *Spyder*. Cuando se haya iniciado el *IDE*, veremos una pantalla parecida a la de la figura A.8.

Ahora, si pulsamos en el icono de la carpeta, situado en la esquina superior izquierda, nos lanzará una ventana del explorador de archivos. Bastará seleccionar los scripts que se adjuntan con el trabajo (se pueden seleccionar varios manteniendo pulsada la tecla *Ctrl*), y ejecutar cada uno de ellos para comenzar el entrenamiento de las redes y poder ver los resultados.

Los *scripts* `cgan-mnist` y `dcgan-mnist` se pueden ejecutar por si solos, y entrenan la red generativa adversarial condicional y la red generativa convolucional durante 20000 épocas, produciendo imágenes como resultados cada 500 épocas. Conviene que sólo se ejecute un *script* a la vez salvo que se tenga buena capacidad de cómputo, y la ejecución

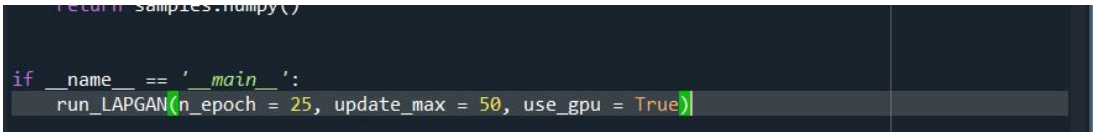


Figura A.9: Parámetros de entrenamiento de la *LAPGAN*

se comienza, o bien pulsando la tecla *F5*, o bien pulsando el icono de *Play* que aparece en la barra superior de tareas.

Por otra parte, los *scripts* `lapgan` y `train_lapgan` se han de ejecutar en ese orden, de modo que se cargue primero el modelo de la red *LAPGAN* y luego se ejecute el entrenamiento de la misma. Los parámetros de ejecución del entrenamiento de la *LAPGAN* se pueden modificar con la última línea de código. Si se quiere utilizar *CUDA*, aconsejamos dejar el parámetro `use_gpu` en `True`, y los parámetros `n_epochs` y `update_max` controlan el número de épocas de la red completa y el número de actualizaciones de parámetros de las redes que conforman los niveles de la pirámide.

Bibliografía

[2] Cem Akkus et al. «Multimodal Deep Learning». En: (2023). arXiv: 2301.04856 [cs.CL].

[4] Dzmitry Bahdanau, Kyunghyun Cho y Yoshua Bengio. «Neural Machine Translation by Jointly Learning to Align and Translate». En: (2016). arXiv: 1409.0473 [cs.CL].

[5] Tadas Baltrušaitis, Chaitanya Ahuja y Louis-Philippe Morency. «Multimodal Machine Learning: A Survey and Taxonomy». En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.2 (2019), págs. 423-443. DOI: 10.1109/TPAMI.2018.2798607.

[8] Xi Chen et al. «InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets». En: (2016). URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/7c9d0b1f96aebd7b5eca8c3edaa19ebb-Paper.pdf.

[9] Kyunghyun Cho et al. «Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation». En: (2014). arXiv: 1406.1078 [cs.CL].

[10] Emily Denton et al. «Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks». En: (2015). arXiv: 1506.05751 [cs.CV].

[11] Jacob Devlin et al. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». En: (2019). arXiv: 1810.04805 [cs.CL].

[16] Ian J. Goodfellow et al. «Generative Adversarial Networks». En: (2014). arXiv: 1406.2661 [stat.ML].

[17] D. H. Hubel y T. N. Wiesel. «Receptive fields, binocular interaction and functional architecture in the cat's visual cortex». En: *The Journal of Physiology* 160.1 (1962), págs. 106-154. DOI: https://doi.org/10.1113/jphysiol.1962.sp006837. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1962.sp006837. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1962.sp006837.

[18] Yacine Jernite, Samuel R. Bowman y David Sontag. «Discourse-Based Objectives for Fast Unsupervised Sentence Representation Learning». En: (2017). arXiv: 1705.00557 [cs.CL].

[19] Rafal Jozefowicz et al. «Exploring the Limits of Language Modeling». En: (2016). arXiv: 1602.02410 [cs.CL].

[20] Alex Krizhevsky. «Learning Multiple Layers of Features from Tiny Images». En: (2009). URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[21] Oleksii Kuchaiev y Boris Ginsburg. «Factorization tricks for LSTM networks». En: (2018). arXiv: 1703.10722 [cs.CL].

[23] Y. Lecun et al. «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324. DOI: 10.1109/5.726791.

[24] Andrew L. Maas, Awni Y. Hannun y Andrew Y. Ng. «Rectifier Nonlinearities Improve Neural Network Acoustic Models». En: 30 (2013). URL: https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.

[25] Bryan McCann et al. «Learned in Translation: Contextualized Word Vectors». En: (2018). arXiv: 1708.00107 [cs.CL].

[28] OpenAI. «GPT-4 Technical Report». En: (2023). arXiv: 2303.08774 [cs.CL].

[29] OpenAI. «Improving Language Understanding by Generative Pre-Training». En: (2018). URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[30] Keiron O'Shea y Ryan Nash. «An Introduction to Convolutional Neural Networks». En: (2015). arXiv: 1511.08458 [cs.NE].

[31] Daniel W. Otter, Julian R. Medina y Jugal K. Kalita. «A Survey of the Usages of Deep Learning in Natural Language Processing». En: (2019). arXiv: 1807.10854 [cs.CL].

[32] Matthew E. Peters et al. «Deep contextualized word representations». En: (2018). arXiv: 1802.05365 [cs.CL].

[34] Alec Radford, Luke Metz y Soumith Chintala. «Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks». En: (2016). arXiv: 1511.06434 [cs.LG].

[35] Alec Radford et al. «Learning Transferable Visual Models From Natural Language Supervision». En: (2021). arXiv: 2103.00020 [cs.CV].

[36] Tim Salimans et al. «Improved Techniques for Training GANs». En: (2016). arXiv: 1606.03498 [cs.LG].

[37] Noam Shazeer et al. «Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer». En: (2017). arXiv: 1701.06538 [cs.LG].

[38] Ilya Sutskever, Oriol Vinyals y Quoc V. Le. «Sequence to Sequence Learning with Neural Networks». En: (2014). arXiv: 1409.3215 [cs.CL].

[42] Ashish Vaswani et al. «Attention Is All You Need». En: (2017). arXiv: 1706.03762 [cs.CL].

[43] Oriol Vinyals et al. «Grammar as a Foreign Language». En: (2015). arXiv: 1412.7449 [cs.CL].

[44] Bing Xu et al. «Empirical Evaluation of Rectified Activations in Convolutional Network». En: (2015). arXiv: 1505.00853 [cs.LG].

Webgrafía

[1] Microsoft (Marzo 2023). *Microsoft 365 Copilot*. Última vez visitado: 2023, 20 de Abril. URL: https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot-your-copilot-for-work/.

[6] Kenjo blog. *¿Cuál es el coste de la empresa al contratar a un trabajador?* Última vez visitado: 2023, 27 de Mayo. URL: https://blog.kenjo.io/es/cual-es-el-coste-de-la-empresa-al-contratar-a-un-trabajador.

[13] Glassdoor. *Sueldos para el puesto de Investigador en España*. (2023, 25 de Mayo) Última vez visitado: 2023, 27 de Mayo. URL: https://www.glassdoor.es/Sueldos/investigador-en-ia-sueldo-SRCH_KO0,12_KE13,18.htm.

[14] Glassdoor. *Sueldos para el puesto de Programador en España*. (2023, 25 de Mayo) Última vez visitado: 2023, 27 de Mayo. URL: https://www.glassdoor.es/Sueldos/programador-sueldo-SRCH_KO0,11.htm?clickSource=searchBtn.

[22] Y. LeCun y C. Cortes. *The MNIST Database*. Última vez visitado: 2023, 27 de Mayo. URL: http://yann.lecun.com/exdb/mnist/.

[26] Medium. *Latex + VS Code*. Última vez visitado: 2023, 27 de Mayo. URL: https://medium.com/@idanielech/latex-vscode-aed802384a2b.

[27] Microsoft. *Microsoft Excel*. Última vez visitado: 2023, 27 de Mayo. URL: https://www.microsoft.com/es-es/microsoft-365/excel.

[33] The LaTeX Project. *An introduction to LaTeX*. Última vez visitado: 2023, 27 de Mayo. URL: https://www.latex-project.org/about/.

[39] Overleaf Team. *LaTeX, Evolucionado*. Última vez visitado: 2023, 27 de Mayo. URL: https://es.overleaf.com/.

[40] University of Toronto (2012). *Lecture slides from the 2012 Coursera course: Neural Networks for Machine Learning*. Última vez visitado: 2023, 24 de Abril. URL: https://www.cs.toronto.edu/~hinton/coursera_slides.html.