Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Generación de escenarios de conducción sintéticos usando Deep Learning y técnicas de Inpainting

---

Alumno/a: Javier Abad Hernández

Tutores: Manuel Barrio Solórzano
Álvaro García García (Fundación CIDAUT)

Agradecimientos

A mi familia y mi pareja, por ser ese apoyo personal y familiar que uno necesita para sacar adelante las cosas en momentos difíciles. A mi tutor de empresa Álvaro y mis compañeros Marcos, Juan, Dani, Enrique, Nico y Chuchi que me animaron a lanzarme en la aventura de sacar adelante una tarea tan poco explorada como es la del inpainting. No me quiero olvidar de Manuel, que no dudó en llevar el seguimiento de este TFG pese a las fechas tan tardías en las que se le propuso la tutoría.

Por último, como aficionado al Atlético de Madrid, quiero agradecer que a lo largo de mi vida me haya enseñado que con trabajo y con esfuerzo las metas que uno se propone, se pueden conseguir. Pese a que en muchos casos se puede sufrir y pasarlo mal, merecerá la pena, ya que finalmente, esos logros se disfrutan mucho más que si el camino hubiera sido sencillo.

Resumen

En este trabajo presentamos un problema en el campo de la conducción autónoma que se apoya en Inteligencia Artificial (IA) en el área de Visión por Computadora (CV) para recrear imágenes mediante la técnica de inpainting.

Actualmente, es un problema que no ha sido explorado en profundidad, por ello, buscamos hallar una solución válida mediante el uso de Redes Neuronales Convolucionales (CNN). Para lograrlo, empezamos a probar distintas soluciones y finalmente, elegimos una solución concreta.

Se presentan los resultados obtenidos para cada red y para varios conjuntos de imágenes (entre los que destacan los de conducción autónoma) mediante el uso de distintas métricas, imágenes y gráficas. De manera que se vea el progreso y los conocimientos obtenidos durante este Trabajo de Fin de Grado.

Palabras clave: inpainting, conducción autónoma, visión por computador, inteligencia artificial, aprendizaje automático, aprendizaje profundo, conjunto de datos, preprocesamiento, postprocesamiento, Python.

Abstract

The problem in autonomous driving relies on Artificial Intelligence in the area of Computer Vision to recreate images using the inpainting technique.

This problem is not well-known, so we want to find a good solution using convolutional neural networks (CNN). To achieve this, we will try different solutions until we choose a concrete one, taking it as the final solution.

Different metrics, images, and graphs are used to present the results achieved for each network and for various image sets. In order to demonstrate the advancements and understanding achieved in this Final Degree Project.

Keywords: inpainting, autonomous driving, computer vision, artificial intelligence, machine learning, deep learning, datasets, pre-process, post-process, Python.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

En la actualidad, la Inteligencia Artificial (IA) está tomando un gran valor e importancia gracias a todas sus aplicaciones en los ámbitos de asistencia a la toma de decisiones. La explosión de diferentes soluciones de IA, como ChatGPT [1] de la compañía OpenAI, ya ofrecen entornos con modelos de chatbot [2] que utilizan enfoques de IA generativa. No obstante, pese a que sea la IA más conocida en la actualidad, no quiere decir ni que sea la única, ni sé que solo sirvan para "mantener conversaciones con ellas".

La IA se utiliza en otros ámbitos, tanto profesionales como no profesionales. Podemos ver ejemplos en aplicaciones médicas, a la hora de comprar por Internet, en asistentes virtuales, traducciones en tiempo real, en videojuegos y también en problemas de robótica móvil.

Una vez se ha presentado de forma breve lo que la sociedad identifica como IA, hemos de profundizar un poco más. Dos "subconjuntos o subniveles"de la IA son el *Machine Learning* y el *Deep Learning*. Estos conforman las distintas arquitecturas internas de las IAs que vemos a diario, y podemos decir que en el campo de la informática, han surgido como herramientas para resolver una gran variedad de problemas, tanto sencillos como complejos. Por hacer un símil, es comparable al momento en el que surgió la calculadora para el ámbito matemático.

En este Trabajo de Fin de Grado, particularmente, nos centraremos en la aplicación del *Machine Learning* y *Deep Learning* dentro de la conducción autónoma, ya que estas técnicas, han impulsado el avance de la visión por computadora de manera significativa en las últimas décadas.

En el campo de la conducción autónoma y visión artificial, han ido apareciendo nuevos problemas como: *Denoising, Low-Light, Deblur*, etc. para los que se han aportado soluciones basadas en técnicas de Deep Learning. En nuestro caso, trataremos el problema del "inpainting"sobre imágenes que es el proceso de relleno de zonas, partes o áreas faltantes de una imagen, de forma que la reconstrucción sea lo más perfecta y coherente en el contexto posible. Al igual que pasa con la IA en general, el inpainting, tiene aplicaciones en un montón de áreas, como la reconstrucción y/o restauración de imágenes antiguas, edición de imágenes para reemplazamiento y/o eliminación de objetos, y en nuestro caso más específico mejora de sistemas de visión en la conducción autónoma.

Para entender cómo esta técnica puede ayudar a mejorar los sistemas de visión en la conducción autónoma, es crucial pensar como humanos, no como máquinas. Suponemos dos escenarios, uno con un humano conduciendo un vehículo, donde hay constante atención a la carretera y otro en el que el vehículo depende del uso de sensores, cámaras y algoritmos para conducir de forma autónoma. Si aparece un niño que se dispone a cruzar la carretera, detrás de un vehículo aparcado, una máquina debería de tener en cuenta todas las posibles situaciones de detección de riesgos. Con la técnica de inpainting, es posible anticiparse al problema gracias a la recreación del escenario antes de que aparezca. Así, mediante esta técnica, los vehículos autónomos puedan ser capaces

de interpretar su entorno visual de forma precisa.

Además, dada la gran diversidad de requisitos para obtener conjuntos de datos apropiados para el entrenamiento de modelos basados en visión por computador para inpainting, en este Trabajo de Fin de Grado se abordarán diferentes técnicas tanto tradicionales como mediante el uso de IA para así generar finalmente escenarios sintéticos, usando *Deep Learning* e Inpainting.

1.1. Motivación

La IA tiene un gran papel en la conducción autónoma. Podemos encontrar noticias y artículos especializados [3] [4] [5] que presentan *"La inteligencia artificial en vehículos autónomos"* [6], y por ende como *"La inteligencia artificial revoluciona el sector del automóvil"* [7]. Sin embargo, no queda claro como funcionan realmente estas IAs, que es necesario para crearlas, como se entrenan y otras muchas cuestiones que nos pueden surgir.

En realidad, la IA aplicada a problemas de los vehículos inteligentes se viene desarrollando desde hace más de una década [8]. Trabajos como *"Are we ready for autonomous driving? the KITTI vision benchmark suite."* [9] de Raquel Urtasun, actualmente una de las líderes en inteligencia artificial en este campo [10][11], nos permiten entender estos problemas desde un punto de vista informático, más concretamente desde el ámbito de la visión por computador.

La posibilidad de trabajar en IA como Ingeniero en Informática me motivó a unirme a Fundación CIDAUT [12]. En particular, tras ver todo lo que se realizaba en este centro de investigación asociado a los proyectos de la línea de visión artificial (*Computer Vision*) para el tratamiento de imágenes para la conducción autónoma. Durante un periodo de dos meses de prácticas, pude estar involucrado en los proyectos ya existentes de mejora de imagen aplicados a escenarios de conducción autónoma, como podría ser *Low-Light Enhancement*[13], *Deblur*[14] y *Denoising*[15]. Esto me ofreció un entorno multidisciplinar para completar mi formación con la realización del Trabajo de Fin de Grado (TFG) en el ámbito del vehículo inteligente. De esta manera, empecé a trabajar en un problema de investigación abierto, la restauración de imágenes con inteligencia artificial para la generación de datasets específicos y modelos de entrenamiento. Esta técnica, conocida como "**inpainting**"[16], se utilizará en este TFG para entrenar diferentes modelos y adaptarlos a situaciones de conducción autónoma. De manera que pueda aportar una posible solución base y sencilla al problema del inpainting en el tratamiento de imágenes.

1.2. Alcance y Objetivos

Este Trabajo de Fin de Grado tiene como objetivo principal la generación de escenarios sintéticos a partir de imágenes ya existentes utilizando técnicas de inpainting mediante *Deep Learning*. De manera que se pueda crear y evaluar un modelo de *Deep*

Learning capaz de rellenar las áreas faltantes de una imagen y aplicarlo específicamente en el contexto de la conducción autónoma. Se pretende demostrar también que es un trabajo replicable en un framework con unas características adaptables a un entorno de estación de trabajo, sin necesidad de usar una alta capacidad de computación distribuida.

Para alcanzar el objetivo propuesto se plantean los siguientes objetivos específicos:

1. Aportar una revisión de la literatura con la revisión de papers ya existentes que utilizan patrones y modelos con técnicas de inpainting mediante *Deep Learning*. Pudiendo así:
    - Investigar y analizar técnicas ya utilizadas de inpainting basadas en *Deep Learning*.
    - Comparar distintos enfoques utilizados para el inpainting, como el uso de redes convolucionales de tipo Encoder-Decoder, GAN o UNet.
2. Recopilación y estudio de varios conjuntos de datos con imágenes y adaptarlos para su uso en nuestro modelo de red neuronal posterior. Concretamente:
    - Toma de varios conjuntos de imágenes tanto generalistas como específicos para la conducción autónoma.
    - Adaptar los conjuntos para poder ser utilizados en la red neuronal, para así poder ser entrenada con calidad.
3. Desarrollar y entrenar varios modelos de redes neuronales hasta encontrar uno con buenos resultados haciendo uso de técnicas de *Deep Learning*.
4. Evaluar el rendimiento del modelo de red neuronal y optimizarlo de manera que pueda mejorar su precisión todo lo posible.
    - Determinar la eficiencia del modelo neuronal gracias a la toma de métricas estándar.
    - Optimizar el modelo de manera que mejore su precisión, gracias al ajuste de hiperparámetros y técnicas de regularización que eviten el sobreajuste.

La última parte del TFG permitirá evaluar el rendimiento del modelo y optimizarlo todo lo posible basándose en métricas de referencia para poder compararlo con otros modelos ya existentes. En particular, se presentará en un segundo caso de uso la generación de estos escenarios destinados a obtener imágenes de escenarios específicos para mejorar la conducción autónoma.

2. Planificación y costes

2.1. Metodología de desarrollo

Este trabajo de fin de grado, tiene como objetivo estudiar y aportar soluciones de tratamiento de imagen con técnicas de inpainting. Para ello se partirá de diferentes problemas de menor a mayor complejidad que permitan el entrenamiento de modelos y la medición de resultados, sobre conjuntos de datos específicos.

Se ha acordado con el centro de trabajo la utilización de una metodología ágil basada en el marco de referencia *SCRUM*. De esta forma se puede aplicar la experiencia previa al haber trabajado con ella en la asignatura Planificación y Diseño de Sistemas Computacionales.

Frecuentemente, *SCRUM* se utiliza en equipos de desarrollo software, pese a ello, se puede aplicar a cualquier trabajo en equipo, ya que en este marco, un equipo único desarrolla, entrega y mantiene productos complejos. *SCRUM*, marca un conjunto de roles, herramientas y reuniones que se ponen en conjunto para poder administrar y estructurar correctamente un trabajo. [17]

Dentro del Equipo *SCRUM*, hay 3 roles que son autoorganizados y multifuncionales, es decir, que eligen la mejor forma de realizar su trabajo y que, a su vez, no dependen de otros que no formen parte del equipo. Estos roles son los siguientes:

- **Líder del equipo *SCRUM* (*SCRUM Master*)**. Entrena y monitoriza a los miembros del equipo para centrarse en la creación de incrementos de alto valor, se asegura de que todos los objetivos que tomen lugar sean productivos y se cumplan a tiempo, ayuda al *Product Owner* a buscar técnicas para definir de manera efectiva los objetivos del producto (*Product Goals*).
- **Dueño del producto (*Product Owner*)**. Es el propietario del producto, representa los intereses de las partes interesadas y es el responsable de gestionar el *backlog* del producto para así poder priorizar el trabajo.
- **Equipo de desarrollo (*Development Team*)**. Son los responsables del trabajo real de creación del producto. Han de tener las habilidades necesarias para lograr acercarse a la meta en cada Sprint. También se tienen que asegurar de que el *backlog* del producto sea transparente, visible y entendible.

Tabla 2.1

Actores implicados

Actor	Rol académico	Rol SCRUM
Álvaro García García	Tutor CIDAUT	Product Owner
Marcos V. Conde Osorio	Colaborador CIDAUT	SCRUM Master
Juan Carlos Benito Núñez	Compañero CIDAUT	SCRUM Master
Daniel Feijoo Piedrafita	Compañero CIDAUT	SCRUM Master
Manuel Barrio Solórzano	Tutor académico	SCRUM Master
Javier Abad Hernández	Autor	Desarrollador

Los actores implicados en este Trabajo de Fin de Grado, se pueden ver en la tabla 2.1 junto a sus respectivos roles.

El marco *SCRUM* tiene algunos eventos mencionados anteriormente que se van a explicar a continuación:

1. **Sprint:** los *sprints* son el corazón de *SCRUM*. Son eventos de duración fija acordada entre los miembros del equipo que pueden durar de una a cuatro semanas. En cada sprint se marcan unos objetivos que se han de cumplir y a lo largo del recorrido de este *sprint*, no se pueden realizar cambios que pongan en peligro estos objetivos.
2. **SCRUM diario:** es un evento de quince minutos para los desarrolladores del equipo. Su objetivo es inspeccionar el progreso hacia los objetivos del *sprint*, adaptando el *backlog* según sea necesario en cada jornada laboral.
3. **Sprint Review:** su propósito es inspeccionar el resultado del *sprint*. En esta reunión, el equipo *SCRUM* muestra los resultados a las partes interesadas para ver el progreso hacia el objetivo final y se decide qué hacer a continuación en el proyecto.
4. **Sprint Retrospective:** en este evento, se planifican las formas de aumentar la calidad y eficacia durante los *sprints*, analizando lo que se logró correctamente durante el *sprint*, cuáles fueron los problemas encontrados y como se resolvieron (en el caso de lograr resolverlos).
5. **Product Backlog:** es una lista de todas las características, funciones y lo que se necesita mejorar y corregir del producto. Está gestionado por el *Product Owner* y se va actualizando en función de las necesidades en el dominio del trabajo.
6. **Sprint Backlog:** resumiendo, es una lista de tareas seleccionadas del *Product Backlog* para las que el equipo *SCRUM* se compromete a completar durante el *sprint*. Gracias a estas tareas, se marca el objetivo general del siguiente *sprint*.

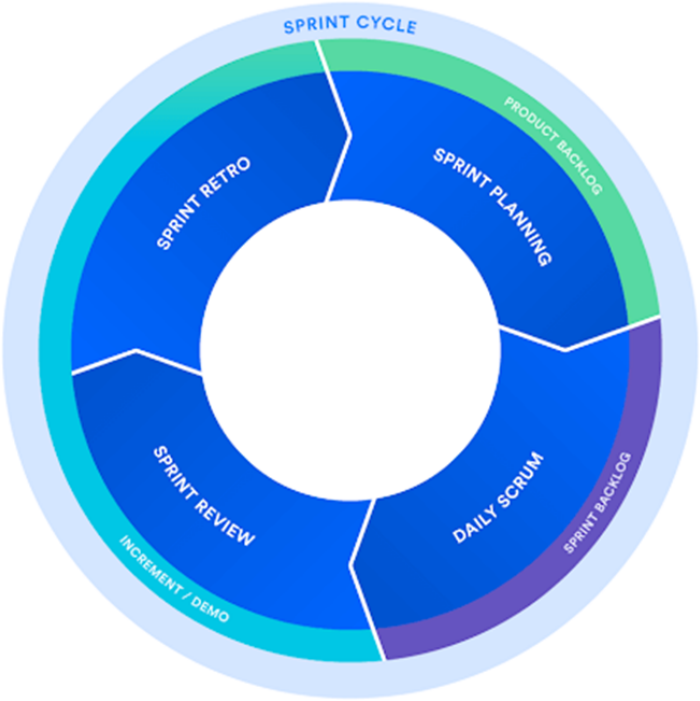


Figura 2.1: Ciclo de un sprint, obtenida de [17]

Tras explicar todos los puntos y detalles necesarios de la metodología utilizada, veremos a continuación la planificación y el proceso seguido a lo largo de este Trabajo de Fin de Grado.

2.2. Planificación del proyecto

En nuestra adaptación al marco *SCRUM*, se ha marcado una duración de los *sprints* a una semana laboral. Acordando revisar los viernes con la Fundación CIDAUT lo que se ha realizado a lo largo de la semana para ver si se han logrado los objetivos. Es decir, en esa reunión semanal, realizamos todas las tareas fundamentales: Fijar el *Product Backlog* en la planificación del *sprint*, mostrar los objetivos cumplidos en el *Sprint Review* que se habían fijado durante el *Sprint Backlog* anterior, y planificar durante el *Sprint Retrospective* los objetivos del siguiente *sprint* en función de lo logrado en el actual. Se puede ver el ciclo a través de todas estas reuniones a seguir en la Figura 2.1.

Gracias a la ayuda del *Scrum Master* y del *Product Manager*, se ha establecido una planificación mensual con ciertos objetivos a cumplir que se han ido dividiendo en pequeños *sprints* semanales.

En la tabla 2.2 se muestra la distribución de los *Sprints* con sus objetivos a lo largo del periodo dedicado en este trabajo de fin de grado. En ella se describen las tareas y si se han logrado o no. Por otro lado, se puede comprobar si respecto a la planificación mensual, se ha seguido la hoja de ruta marcada o si ha recibido alguna modificación según iba avanzando el proyecto.

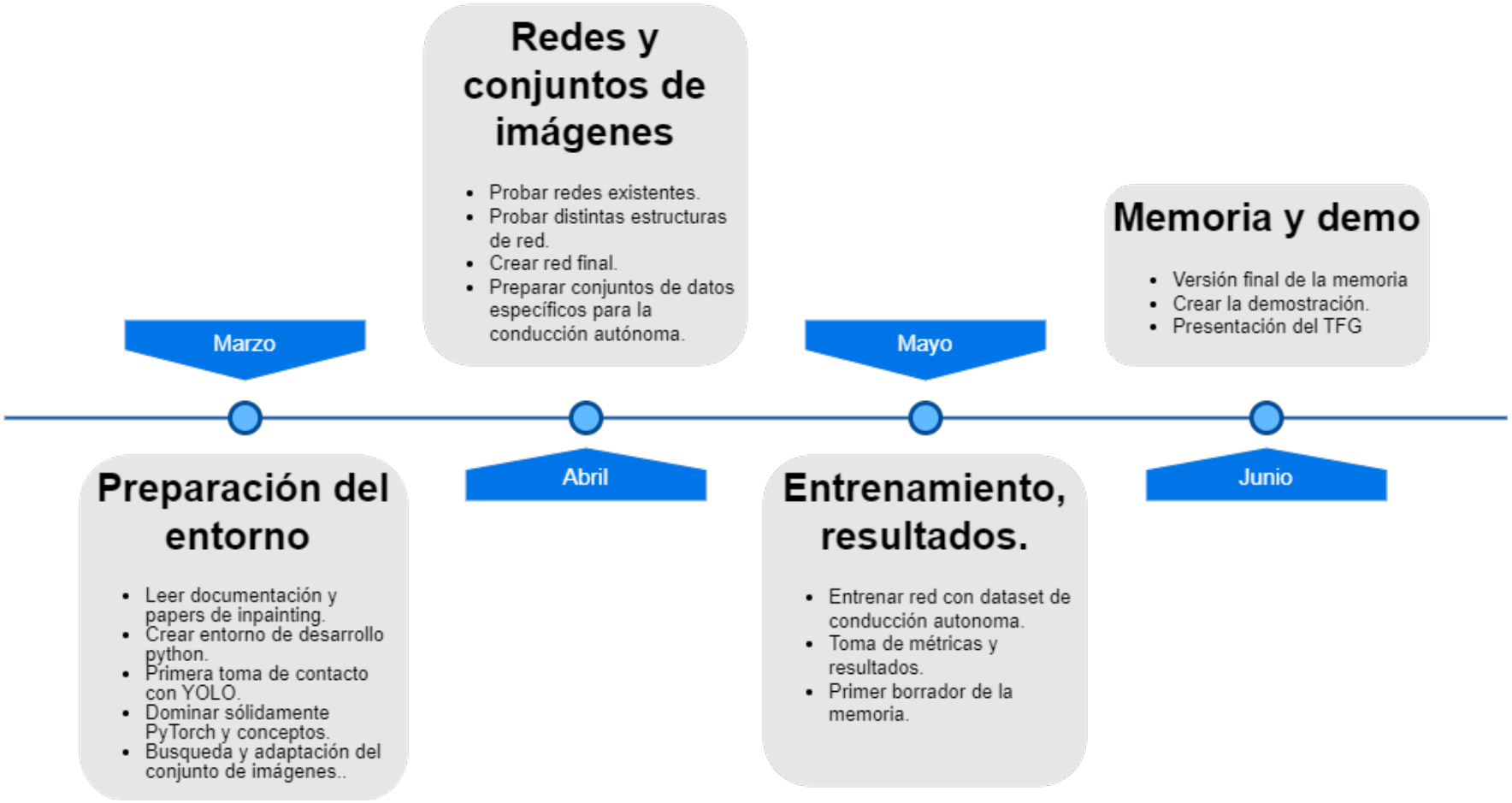


Figura 2.2: Planificación mensual del TFG [17]

Tabla 2.2

Resumen de Sprints y Objetivos Reestructurados

Sprint	Fecha inicio	Fecha fin	Objetivos	Logrado
Sprint 0	04-03-2024	08-03-2024	Leer documentación y papers de inpainting.	Sí
			Crear entorno de desarrollo en Python.	Sí
			Primera toma de contacto con YOLO.	No
			Dominar PyTorch y conceptos.	Sí
Sprint 1	11-03-2024	15-03-2024	Búsqueda y adaptación de imágenes.	Sí
			Crear red neuronal simple de prueba.	No
			Estudiar Dataset y Dataloader en PyTorch.	Sí
Sprint 2	18-03-2024	22-03-2024	Adaptar imágenes para usar con la red.	Sí
			Crear y entrenar redes sencillas.	Sí
			Probar Encoder-Decoder.	Sí
Sprint 3	25-03-2024	29-03-2024	Ajustar parámetros para mejorar resultados.	Sí
			Estudiar y probar redes GAN.	Sí
			Recopilar y analizar resultados.	Sí
Sprint 4	01-04-2024	05-04-2024	Probar redes existentes.	Sí
			Probar distintas estructuras de red.	Sí
			Crear red final.	Sí
Sprint 5	08-04-2024	12-04-2024	Preparar conjuntos de datos específicos para la conducción autónoma.	Sí
			Revisar Encoder-Decoder con bloques residuales y atención contextual.	Sí
			Probar la red UNet.	Sí
Sprint 6	15-04-2024	19-04-2024	Documentar resultados.	Sí
			Mejorar la red UNet y crear red final.	Sí
			Buscar imágenes de vehículos autónomos.	Sí
Sprint 7	22-04-2024	26-04-2024	Crear y adaptar imágenes para VKITTI2.	Sí
			Probar la red final con el dataset creado.	Sí
			Crear y adaptar imágenes para BDD100K.	Sí
			Entrenar la red con 6000 imágenes.	Sí
Sprint 8	29-04-2024	03-05-2024	Entrenar red con 6000 imágenes.	Sí
			Entrenar red con 12000 imágenes.	Sí
			Comenzar redacción del informe final.	Sí

Sprint	Fecha inicio	Fecha fin	Objetivos	Logrado
			Entrenar red con 24000 imágenes.	Sí
Sprint 9	06-05-2024	10-05-2024	Continuar redacción del informe.	Sí
			Recopilar imágenes para el informe.	Sí
			Crear scripts para métricas y representación de datos.	Sí
			Entrenar red con 48000 imágenes.	Sí
Sprint 10	13-05-2024	17-05-2024	Entrenar red con 60000 imágenes.	No
			Toma de métricas y resultados finales.	Sí
			Primer borrador de la memoria.	Sí
			Revisar y ajustar resultados.	Sí
Sprint 11	20-05-2024	24-05-2024	Preparar versión final de la memoria.	Sí
			Iniciar la preparación de la demo del proyecto.	Sí
			Revisar y ajustar resultados de la red.	Sí
Sprint 12	27-05-2024	31-05-2024	Iniciar la presentación de la defensa del TFG.	Sí
			Finalizar primera versión final de la memoria.	Sí
Sprint 13	03-06-2024	07-06-2024	Finalizar primer borrador de la presentación.	Sí
Sprint 14	10-06-2024	14-06-2024	Revisar y aplicar cambios en la memoria del TFG.	No
Sprint 15	17-06-2024	21-06-2024	Revisar y aplicar cambios en la memoria del TFG.	Sí
Sprint 16	24-06-2024	28-06-2024	Ultimar detalles de la demostración.	Sí
			Finalizar la presentación para la defensa.	Sí
Sprint 17	01-06-2024	07-06-2024	Documentar y cerrar el proyecto.	Sí
			Entregar versión final de la memoria.	Sí

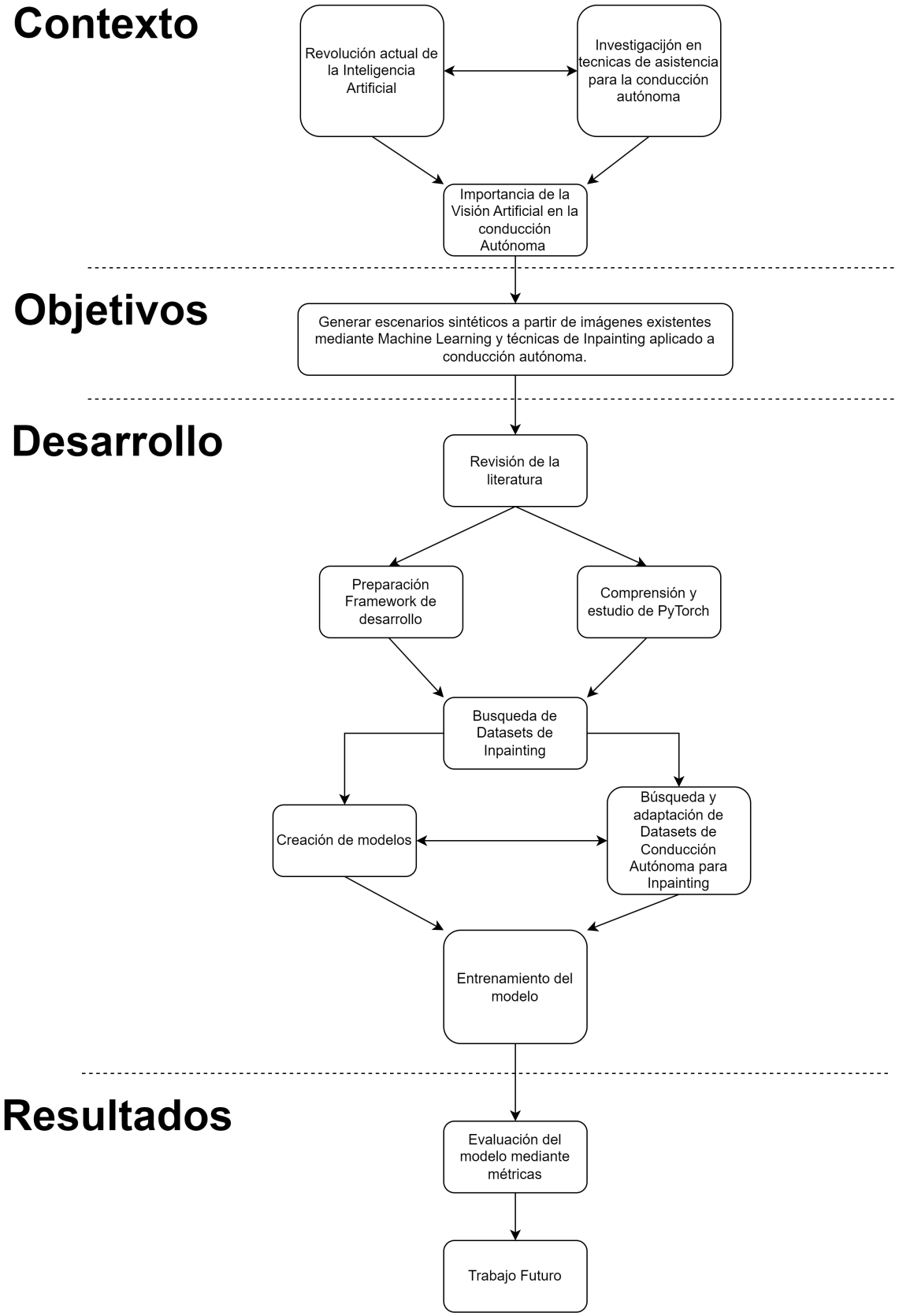


Figura 2.3: Planificación general del TFG

2.3. Plan de Costes

Para hablar de costes, primero tenemos que hablar de los recursos computacionales necesarios.
Hemos utilizado un *framework* basado en estaciones de trabajo Linux con GPUs Nvidia (Unidades de Procesamiento Gráfico) como el dispositivo estándar para el entrenamiento y desarrollo de modelos *Deep Learning*. La GPU, como sabemos, es un procesador especializado con memoria dedicada, que usualmente realiza las operaciones de punto flotante necesarias para renderizar gráficos. En este contexto, usamos este dispositivo porque está optimizado para procesar múltiples operaciones de forma paralela, que en su mayoría son multiplicaciones de matrices (las GPUs comerciales son 5 veces más rápidas que las CPUs).

Tabla 2.3

Especificaciones de Configuración en sistemas con GPU Nvidia

Configuración	**Almacenamiento**	**RAM**	**GPU**
Ordenador CIDAUT	4 TB SSD	32 GB	RTX 4090
Ordenador CIDAUT	2 TB SSD	16 GB	RTX 3060
Ordenador personal	500 GB SSD	16 GB	GTX 1050
Kaggle GPU	73 GB	16 GB	Tesla P100
Google Colab GPU	1008 GB	12 GB	Tesla T4

Los modelos de Deep Learning que se han desarrollado a lo largo del recorrido de este TFG, han usado las configuraciones descritas en la tabla 2.3. Los recursos ofrecidos por Kaggle [1] y Google (Colab)[2] son gratuitos tras el registro en cada uno de sus sitios web.

Al tener varias configuraciones a nuestra disponibilidad, tenemos la posibilidad de crear, entrenar varios modelos de forma paralela entre todas las configuraciones. Pese a ello, tanto con Google Colab como con Kaggle tenemos limitado el tiempo de uso de cada una de estas GPUs por sesión o por semana. En el caso de Google Colab solo nos permite utilizar su GPU durante 12 horas seguidas, tras ello se podría volver a ejecutar, mientras que Kaggle, solo nos permite usar una GPU por un máximo de 30 horas semanales.

Para determinar el coste del proyecto nos basamos en el precio de los recursos utilizados, que se muestran en la tabla 2.4.

[1] https://www.kaggle.com
[2] https://colab.research.google.com/?hl=es

Tabla 2.4

Comparativa de GPUs y Costes

GPU	**VRAM (GB)**	**Coste mercado actual ($)**	**Plataforma**	**Coste final ($)**
Tesla P100	16	3792	Kaggle	0
Tesla T4	16	582	Google Colab	0
RTX 4090	24	1756	CIDAUT	0
RTX 3060	12	317	CIDAUT	0
GTX 1050	4	211	Propio	0

Fuente: Datos obtenidos de Technical City[3].

3. Marco Teórico

El término Inteligencia Artificial (IA), fue acuñado en el año 1955 por el profesor emérito de Stanford John Macarthy. Se define como "la ciencia y la ingeniería para fabricar máquinas inteligentes". Abarca una gran cantidad de disciplinas diferentes, incluidas la informática, el análisis de datos y estadísticas, la ingeniería del hardware y del software, la lingüística, y la neurociencia, entre otros [18].

En particular, si hablamos de un problema abierto, como es la conducción autónoma, encontramos una revolución de la IA para comprender y modelar la percepción humana mientras se realiza una tarea altamente especializada. Por ello, la conducción es uno de los problemas más desafiantes de la década y ni siquiera está cerca de resolverse [19]. Uno de estos problemas atañe a los agentes basados en la visión. Conseguir que un coche pueda conducir sin intervención humana se puede abordar desde el punto de vista de la percepción y visión a través del desarrollo y entrenamiento de casos de uso, atendiendo a diferentes sistemas de interacción contextual, por ejemplo con imágenes.

Las principales soluciones que encontramos actualmente se centran en la percepción del entorno mediante información visual [20]. En ellas, se utilizan diferentes técnicas de visión por computadora, *Machine Learning*, redes neuronales, y algoritmos diversos para conseguir el tratamiento de imagen de forma que el sistema autónomo entienda el entorno o escena sin necesidad de intervención humana.

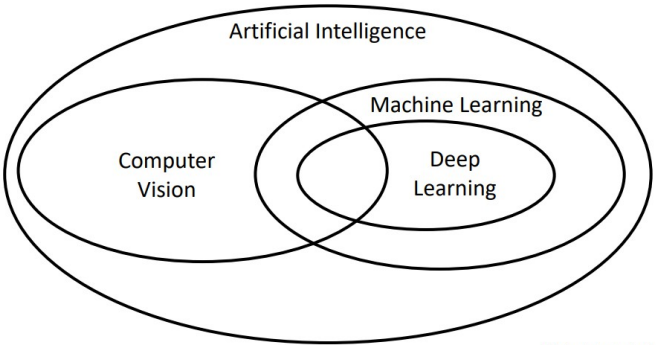


Figura 3.1: Interrelación entre diferentes técnicas de la IA

En este capítulo se introducen todas estas técnicas, dedicando especial atención a la generación de escenarios de conducción sintéticos usando *Deep Learning* y técnicas de Inpainting.

3.1. Visión por computadora *(Computer Vision)*

La Visión por Computadora es un campo de la Inteligencia Artificial en la que se entrena a los sistemas computacionales para que sean capaces de replicar la complejidad del sistema de visión humano [21], de manera que puedan interpretar y percibir el

mundo visual. Para ello utilizan imágenes digitales de cámaras y vídeos, que junto con modelos de *Deep Learning*, les permiten realizar ciertas tareas, como identificar, clasificar y segmentar imágenes [22].

1. **Clasificación**. Dada una imagen, podemos saber qué categoría general de objetos se encuentra en ella. De manera más técnica, es capaz de predecir a qué clase pertenece una imagen dada.

2. **Detección**. Una vez clasificados uno o varios objetos, se pueden detectar y localizar en la imagen gracias al dibujado de una caja delimitadora de coordenadas en las que está situado el objeto detectado.

3. **Seguimiento de objetos**. Tras haber detectado la posición de un objeto, esta tarea se aplica en vídeos en tiempo real o secuencias de imágenes, en las que se puede monitorizar y seguir ciertos objetos en escenarios concretos.

4. **Segmentación**. Es una ampliación de la detección de objetos, que se encarga de decir que píxeles exactos pertenecen al objeto detectado en la imagen, de manera que podemos ver de forma visual su área específica, sin tener que utilizar cajas delimitadoras.

5. **Otras tareas**. Hay muchas otras aplicaciones como la generación de imágenes, manipulación de características, el inpainting, la superresolución y otras muchas más.

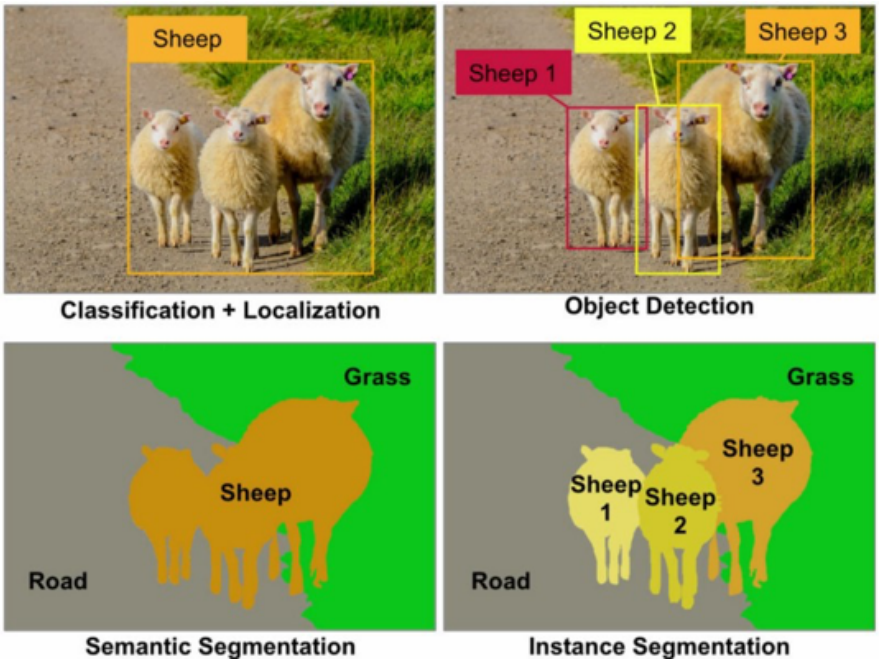


Figura 3.2: Tareas de Computer Vision. Tomado de [23]

En la conducción autónoma, la principal función es permitir que los vehículos sepan manejarse en el entorno gracias a la captura de imágenes y vídeo para que sean procesados en tiempo real y así realizar ciertas tareas que haría un ser humano, como pueden ser la lectura de señales de tráfico, detección de obstáculos (otros vehículos, animales, peatones, objetos, etc.); seguimiento del carril de conducción y otras muchas

que permitan que el vehículo sea capaz de conducir de forma autónoma sabiendo desenvolverse en situaciones complicadas de forma totalmente segura, autónoma y eficiente. (ver figura 3.3).

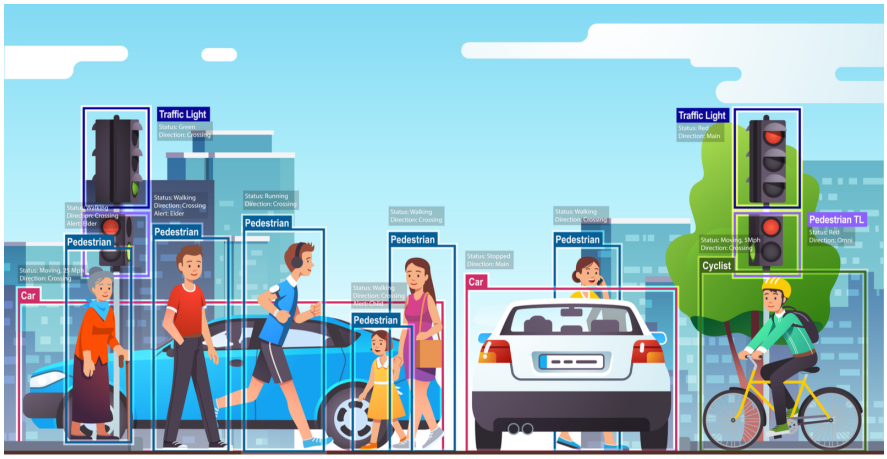


Figura 3.3: Representación de la detección de objetos en vehículos autónomos. Obtenido de [24]

Para seguir un orden lógico que permita abordar problemas de tratamiento de imagen, primero tenemos que entender que es el *Machine Learning*, el *Deep Learning* y los tipos de redes existentes.

3.2. Aprendizaje Automático (*Machine Learning*)

El *Machine Learning* es una disciplina del campo de la Inteligencia Artificial, que mediante el uso de algoritmos, da a los sistemas informáticos la capacidad de identificar patrones en datos masivos y gracias a esto aprender, es decir, un análisis basado en datos entrenados, de manera que puedan realizar tareas especializadas de forma autónoma [25].

Dentro del *Machine Learning* podemos destacar tres tipos de aprendizaje:

- **Supervisado**. Es un aprendizaje que hace uso de datos que previamente ya estaban etiquetados, es decir, que conocíamos a qué clase pertenecen. Tras ello, los utiliza para aprender sus características y poder distinguir entre ellos para clasificar posteriormente otros datos no etiquetados. Para estas tareas se utiliza lo siguiente: regresión lineal y máquinas de vectores de soporte (SVM).
- **No supervisado**. El aprendizaje no supervisado, no utiliza datos etiquetados. Para aprender, busca patrones para agrupar los datos en distintos conjuntos de datos. Algunos de los algoritmos aplicables son clustering y k-means.
- **Por refuerzo**. Se centra en cómo los agentes deben tomar acciones en un entorno para maximizar alguna noción de recompensa acumulativa. Para entenderlo correctamente se podría decir de forma informal que siguen el proceso utilizado en muchos casos por el ser humano: prueba-error.

Podemos utilizar diferentes enfoques de aprendizaje automático de acuerdo a los tipos mencionados anteriormente. En particular, se van a describir brevemente los algoritmos de *Machine Learning* más utilizados, mostrando brevemente en figuras sus implementaciones.

3.2.1. Algoritmos de aprendizaje supervisado

Regresión Lineal

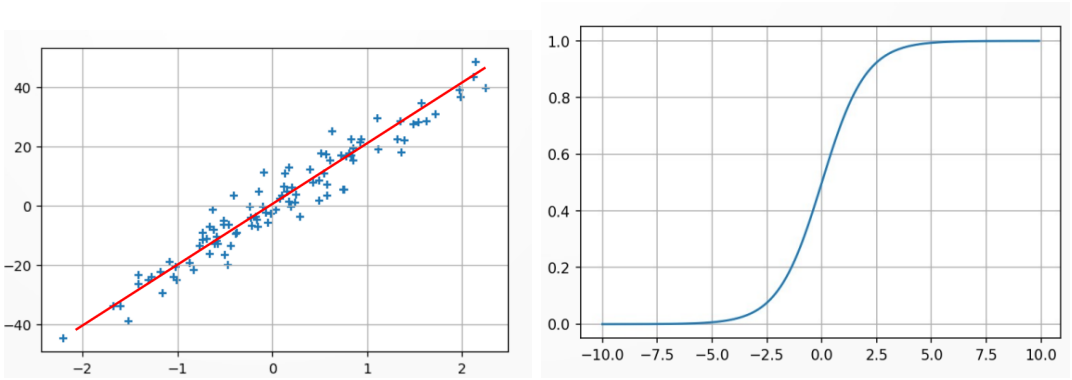
Se utiliza para poder predecir un valor en función de varias variables independientes. Para ello, utiliza dos métodos para predecirlo que son los siguientes:

- **Mínimos Cuadrados**. Utiliza el error cuadrado en todas las muestras y trata de minimizarlo.
- Trata de **ajustar los pesos** incrementándolos de forma proporcional al gradiente de la función de coste.

Regresión logística

En este caso, la diferencia con la regresión lineal está en que predice la probabilidad de que una instancia pertenezca a una clase concreta. Se divide en dos:

- **Binaria**. Clasifica las muestras en dos categorías, 0 o 1, utilizando la función logística[26] o sigmoide[26] de manera que se estimen las probabilidades.
- **Múltiple**. Utiliza varias técnicas como la clasificación por pares o *Softmax*[27] para poder manejar múltiples clases.



(a) Ejemplo de regresión lineal (b) Salida de la regresión logística

Figura 3.4: Regresión lineal vs. Regresión logística

Perceptrón Simple

Se utiliza para clasificar datos que son linealmente separables. Utiliza una función de activación que permite clasificar los datos en dos categorías y si finalmente el

conjunto de datos es linealmente separable, el algoritmo converge en un número finito de pasos.

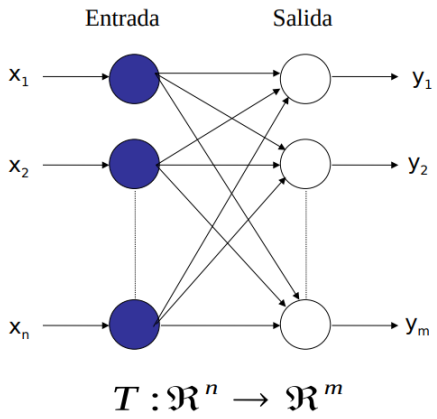


Figura 3.5: Arquitectura del perceptrón simple

Perceptrón Multicapa (MLP)

Su diferencia principal con el perceptrón simple, es que funciona con datos que no son linealmente separables, y tiene tres tipos de capas:

- **Entrada**: neuronas de entrada, en ellas no se produce procesamiento.
- **Ocultas**: neuronas cuya entrada proceden de una capa anterior y salida va a una posterior.
- **Salida**: neuronas cuya entrada viene de una capa anterior, pero su salida es la final de la red.

Para entrenar estas redes se utiliza el algoritmo de retropropagación, adaptando los pesos y propagando los errores a las capas ocultas inferiores.

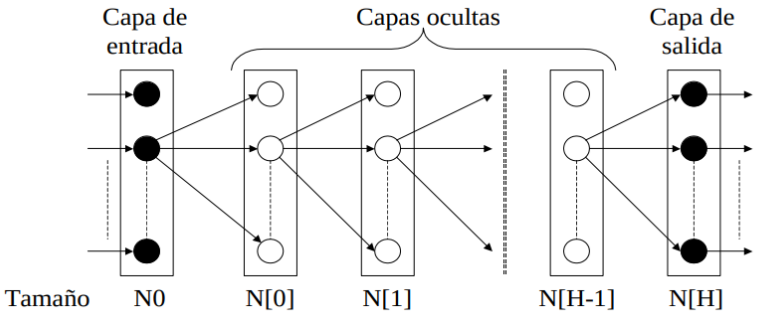
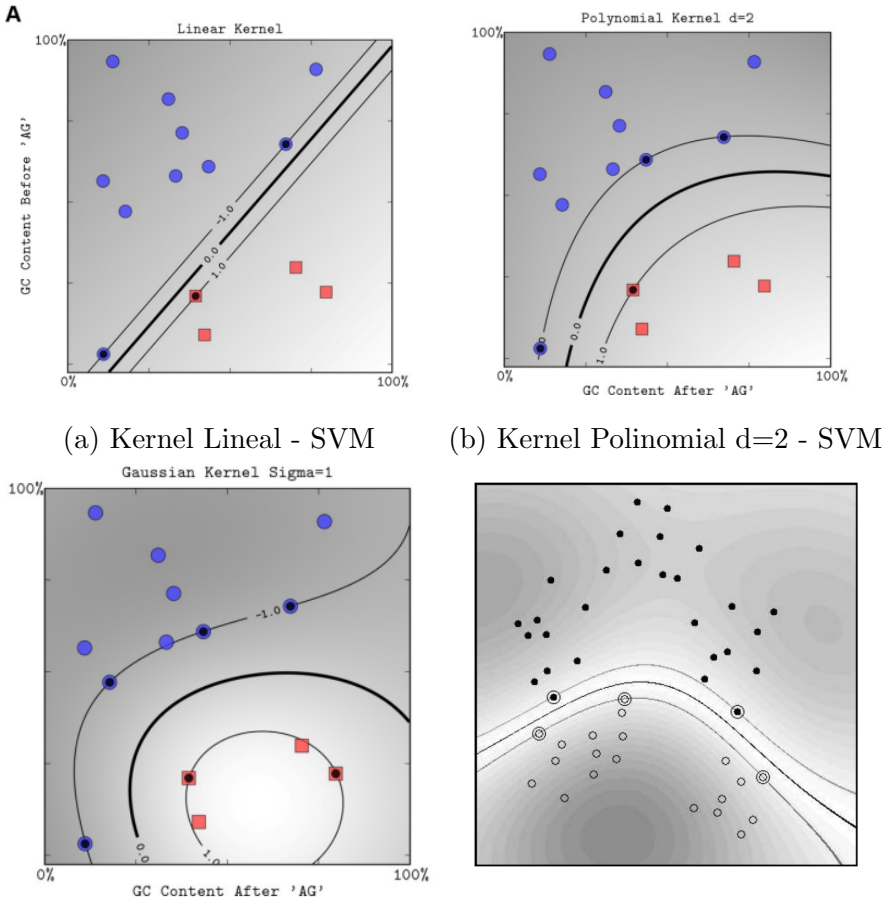


Figura 3.6: Arquitectura del MLP

Máquinas de Vectores Soporte (SVM)

Se basan en la idea de encontrar un hiperplano que separe las clases de datos. Para ello se trata de buscar el hiperplano que maximice la distancia a los puntos más cercanos de cada clase.



(a) Kernel Lineal - SVM (b) Kernel Polinomial d=2 - SVM

(c) Kernel Gausiano - SVM (d) Kernel Perceptrón - SVM

Figura 3.7: Aplicación visual de distintos kernels en una SVM

Necesitan las funciones núcleo para poder realizar la transformación de forma implícita y así trabajar en espacios con alta dimensionalidad sin realizar una transformación directa (se puede ver en la figura 3.8).

3.2.2. Algoritmos de aprendizaje no supervisado

K-means

Algoritmo de clustering que agrupa los datos en un número de conjuntos K diferentes según las características de cada uno. Para agruparlos se minimiza el MSE de cada objeto respecto al centroide del clúster. Es un algoritmo que se implementa rápido y de forma sencilla para el manejo de grandes conjuntos de datos, aunque depende de la elección del tamaño de los clústeres.

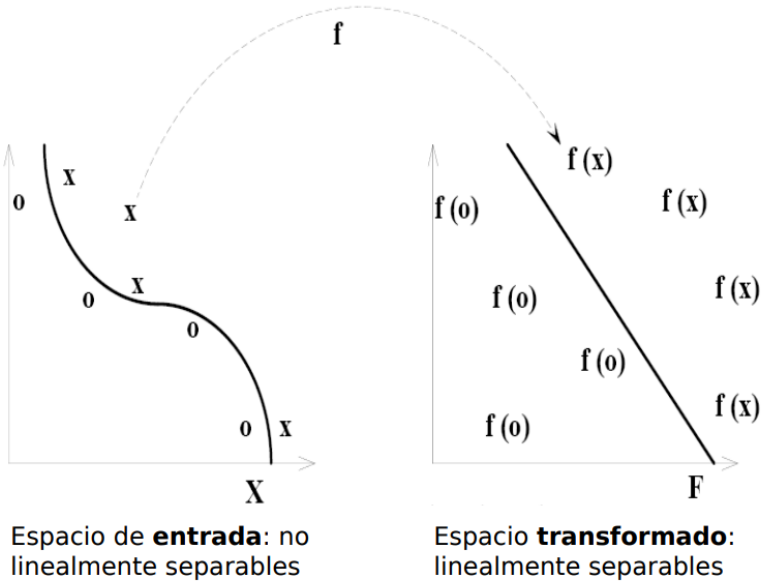


Figura 3.8: Transformación no lineal del espacio de entrada de un SVM

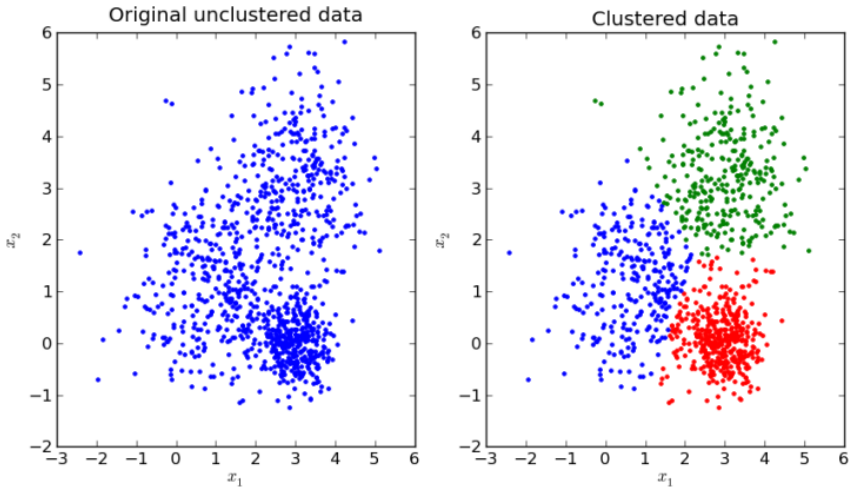


Figura 3.9: Paso de datos juntos a varias categorías

K-medoids

Aplica una idea parecida a *k-means*, pero no usa el centroide como punto central, sino el medoide, que es el punto más central de un clúster y es más robusto que *k-means*. Esta diferencia se puede ver en la figura 3.10.

DBScan

Es de los más usados después de *k-means* y busca los *clusters* en función de la densidad de los datos, para ello busca un punto núcleo si hay un número de puntos a menos de una distancia de él.

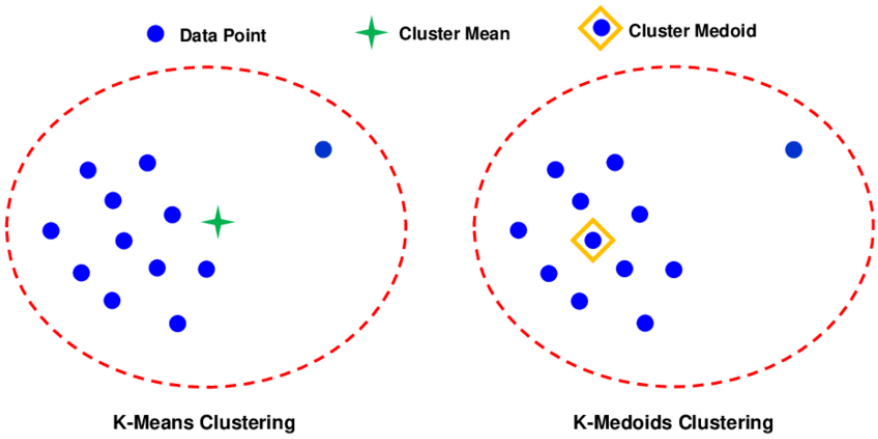


Figura 3.10: Representación gráfica de la diferencia entre los métodos de agrupación k-means y k-medoids. Tomado de [28]

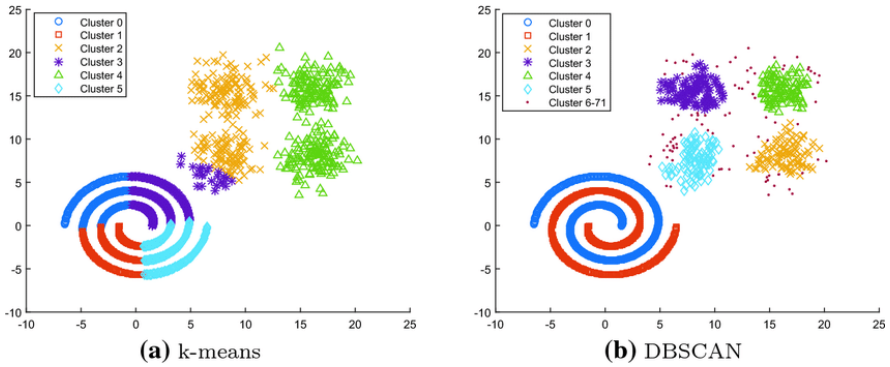


Figura 3.11: Resultados de la agrupación k-means y DBSCAN en el conjunto de datos en espiral. La subfigura b solo muestra los 6 conglomerados más grandes de los 72 obtenidos mediante el algoritmo DBSCAN. Tomado de [29]

3.3. Aprendizaje Profundo (*Deep Learning*)

El *Deep Learning* es un subconjunto del *Machine Learning* que utiliza redes neuronales multicapa (artificiales) capaces de aprender de datos no estructurados y conseguir resultados con una gran precisión. Su objetivo es tratar de emular el comportamiento del cerebro humano (de ahí lo de red neuronal), pudiendo aprender a partir de muchos datos. Su diferencia principal respecto al *Machine Learning*, es que puede procesar datos no estructurados como el texto, las imágenes o vídeos. Dos de las redes neuronales existentes que más se utilizan para abordar problemas con *Deep Learning* son las Redes Neuronales Recurrentes (RNN) 3.3.1 y las Redes Neuronales convolucionales (CNN) 3.3.2.

3.3.1. Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes son un tipo de red neuronal, que procesa de manera secuencial los datos, y las salidas procesadas, se utilizan como entrada más tarde

junto con otros datos nuevos. De esta manera, la red logrará memorizar información previa. [30] Asimismo, son capaces de usar la variable tiempo en sus conexiones, por lo que son idóneas para predicciones temporales y clasificar secuencias. Las RNN son muy flexibles, ya que nos permiten operar sobre las secuencias de entrada, salida o ambas en la arquitectura de la red. Hay varias arquitecturas de RNN que manejan secuencias de longitud variable (ver figura 3.12). También permiten trabajar con secuencias de entrada y salida, de manera que pueden ser configuradas para hacer distintas tareas con distintas arquitecturas:

- **Uno a muchos**. Como ejemplo se puede tomar el subtitulado de imágenes; en ella se toma una imagen con tamaño fijo y se produce una secuencia de palabras que describen esa imagen. (Ver ejemplo 2 - fig. 3.12).
- **Muchos a uno**. Predicción de acciones de un vídeo. Se toman muchos fotogramas de vídeo en lugar de una imagen y se entrega una única salida de lo que ha ocurrido. (Ver ejemplo 3 - fig. 3.12).
- **Varios a varios**. Sirve por ejemplo para subtitular un vídeo en tiempo real.
- **Variación de muchos a muchos**. El modelo genera una salida en cada paso del tiempo. Un ejemplo de esta tarea es la clasificación de vídeo fotograma a fotograma, donde se clasifica para cada fotograma con un cierto número de clases.

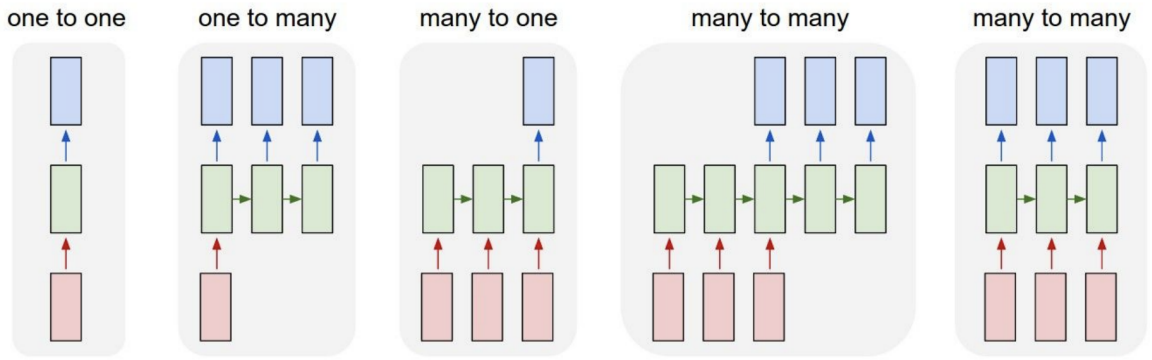


Figura 3.12: Diferentes tipos de RNN. Cajas rojas: vectores de entrada. Verde: capas ocultas. Azul: Capas de salida. Obtenido de CS231n[31]

Su aprendizaje puede ser de dos tipos, por épocas o en tiempo real.

- **Por épocas**: solo al final de cada una de las series aparece la salida deseada. En ese momento se modifican los pesos.
- **Tiempo Real**: para cada entrada de la secuencia se dispone de la salida deseada y la modificación de los pesos es en cada entrada.

Este tipo de redes puede ser de dos tipos: parcialmente recurrentes y totalmente recurrentes. A continuación, veremos una pequeña descripción de cada una de ellas y entraremos más en detalle en las parcialmente recurrentes.

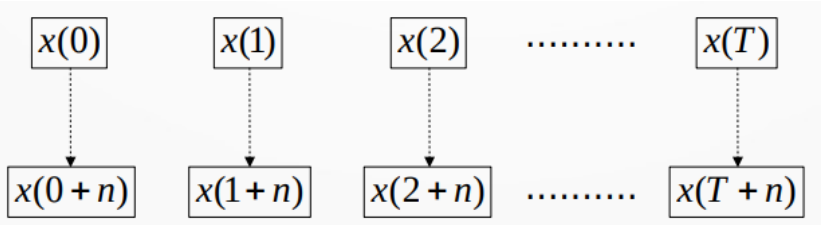


Figura 3.13: RNN: Aprendizaje en tiempo real

Redes neuronales parcialmente recurrentes

Son redes multicapa con conexiones recurrentes. Estas conexiones permiten recordar el estado anterior de ciertas neuronas de la red. Tienen unas neuronas especiales, conocidas como neuronas de contexto. Estas neuronas funcionan como una memoria de la red, almacenando el estado de las neuronas en el instante anterior. Se puede ver su estructura en la figura 3.14.

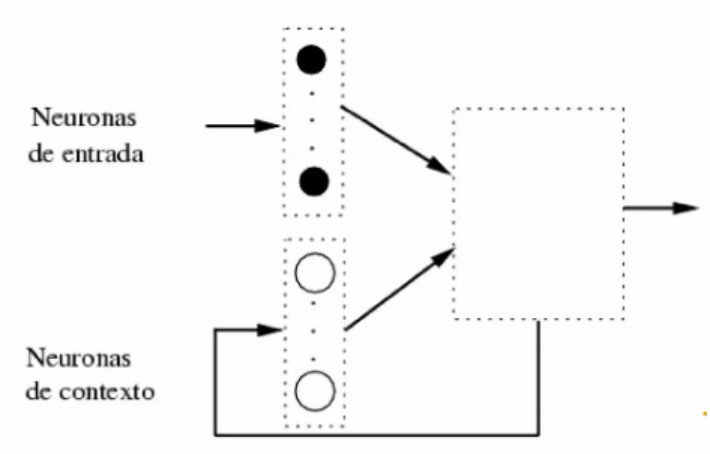


Figura 3.14: Redes Parcialmente Recurrentes: Estructura

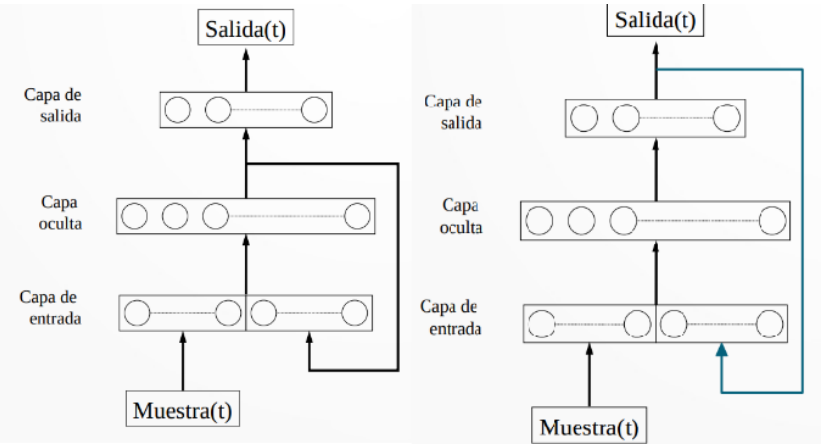


Figura 3.15: Redes Parcialmente Recurrentes: Elman (izquierda), Jordan (derecha)

Las dos redes más conocidas de este tipo son la Red de Elman[32] y la red de Jordan[33][34]. La red de Elman, tiene conexiones hacia atrás desde la capa oculta a la

capa de contexto, lo que le permite tener en cuenta la información temporal. Para calcular la salida tenemos que dividirlo en dos, en la de la capa oculta y la capa de salida.

$$y(t) = F\left(\sum_{j=0}^{N} w_{ij} y_j(t-1) + x_i(t)\right)$$

En la fórmula anterior, se representa el cálculo de la salida de la capa oculta en el tiempo (t), donde F es la función de activación, w_{ij} son los pesos de la red, $y_j(t-1)$ es la salida de la neurona (j) en el tiempo $t-1$, y $x_i(t)$ es la entrada (i) en el tiempo (t).

$$y(t) = F\left(\sum_{j=0}^{N} w_{ij} y_j(t-1) + \sum_{k=0}^{N} w_{jk} x_k(t)\right)$$

Esta fórmula es similar, en este caso corresponde a la salida de la capa de salida. Podemos ver que incluye la salida de la capa oculta del paso anterior, como la entrada actual. Para su entrenamiento utiliza el aprendizaje por épocas en tiempo real, de manera que modifica los pesos de la red al final de la secuencia o en cada entrada.

La red de Jordan es similar a la red de Elman. En este caso, las conexiones de retroalimentación las tiene desde la capa de salida hacia la capa de entrada, mientras que en Elman era desde la oculta. Utiliza la salida anterior como parte de la entrada para la siguiente iteración, por lo que se tiene en cuenta la información temporal previa. A la hora de entrenar, utiliza el algoritmo de retropropagación en el tiempo, de manera que pueda utilizar el término momento para así evitar caer en mínimos locales. Como en la red anterior se pueden calcular mediante las siguientes fórmulas su salida en función del tiempo. La capa oculta utiliza la salida anterior y la entrada actual para calcular la salida de la capa oculta.

$$y(t) = F\left(\sum_{j=0}^{N} w_{ij} y(t-1) + \sum_{k=0}^{N} w_{jk} x(t)\right)$$

Por otro lado, la capa de salida toma la salida de la capa oculta y la utiliza para calcular la salida de la capa de salida final de la red.

$$\hat{y}(t) = F\left(\sum_{i=0}^{N} w_i y(t)\right)$$

Tanto en Jordan como en Elman, estas fórmulas sirven para entender como se procesa la información de manera secuencial y como se actualizan los pesos durante el entrenamiento de la red.

Redes neuronales totalmente recurrentes

Las redes totalmente recurrentes, por lo contrario, no tienen ningún tipo de restricción de conectividad. En cada neurona se recibe como entrada la activación de todas

las neuronas y la suya propia. Su aprendizaje es más complicado, ya que los pesos de cada conexión se adapta en función del mismo. Aplica la retropropagación en el tiempo [35] que consta de las siguientes características:

- Consta de una sola capa.
- Se desenrollan en un diagrama temporal (mismas neuronas en diferentes instantes)
- Da lugar a una capa replicada en el tiempo
- Minimización de una función de coste
- Da la imagen de un MLP (en la misma capa).

En ella, los pesos coinciden para todas las capas de la red multicapa, de manera, que para poder calcular el gradiente respecto al peso, se ha de hacer de manera individual para cada capa y más tarde sumarlos.

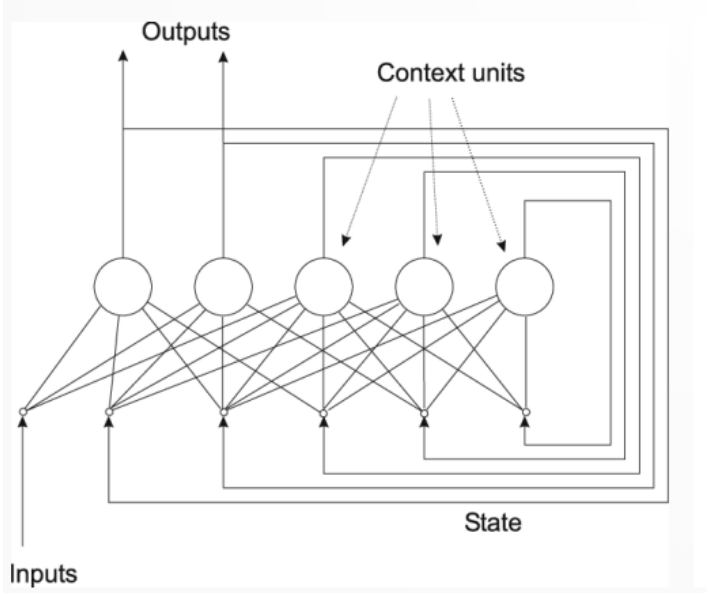


Figura 3.16: Redes Totalmente Recurrentes: Estructura

Redes neuronales LSTM (*Long Short-Term Memory*)

Tanto Elman como Jordan, o incluso las Totalmente Recurrentes, no dan buenos resultados cuando las secuencias son relativamente grandes, esto se debe al problema de la Evanescencia del Gradiente. Con este problema la magnitud del gradiente se va reduciendo de una capa a otra y llega a ser imperceptible en capas muy profundas. Es por ello que aparecen las LSTM [36]. Las LSTM (3.18) son una extensión de las RNN con las que se evita el problema del desvanecimiento del gradiente, de manera que la red puede tener memoria a largo plazo gracias a la adición de una "cinta transportadora", de esta manera, recuerda entradas anteriores. En la figura 3.19, se puede ver una comparación entre una Red Neuronal Recurrente normal y entre la LSTM.

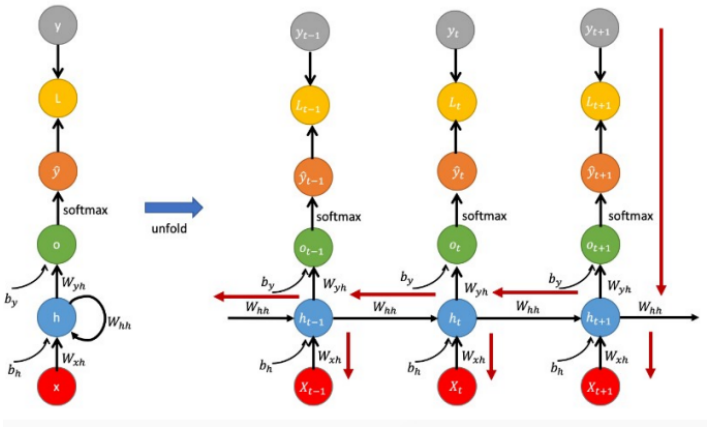


Figura 3.17: Redes Totalmente Recurrentes: Retropropagación

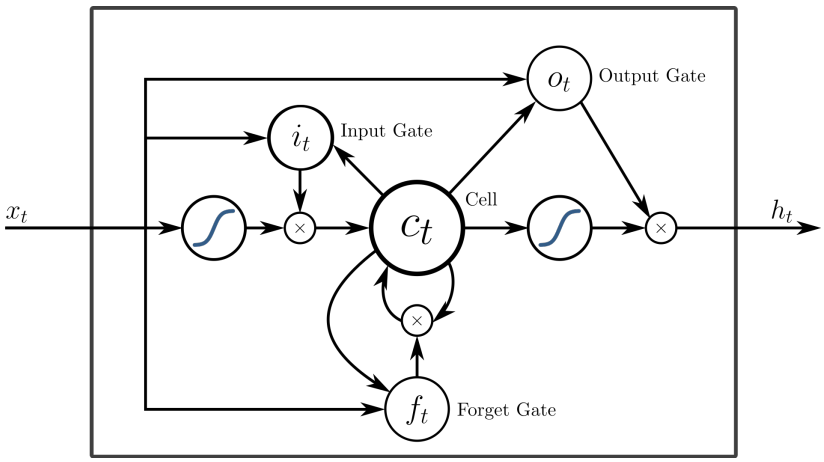


Figura 3.18: Estructura interna de una LSTM. Obtenida de Wikipedia [37]

La red LSTM no garantiza que se solucione el problema del desvanecimiento del gradiente, pero sí que ayuda a que el modelo aprenda de dependencias con una longitud pasada amplia.

3.3.2. Redes Neuronales Convolucionales (CNN)

Las Redes Convolucionales, son similares a las redes neuronales ordinarias (RNN). Al igual que las RNN, están formadas por pesos y sesgos *(bias)* aprendibles [38]. Recibe entradas y le aplica el producto escalar. Siguen teniendo su función de pérdida y otros puntos que también tienen las redes neuronales normales. La verdadera diferencia está en que las Redes Convolucionales asumen que las entradas son imágenes o vídeos, que permiten la codificación de un conjunto de propiedades de las mismas en la arquitectura para así poder reducir los parámetros de la red eficientemente.

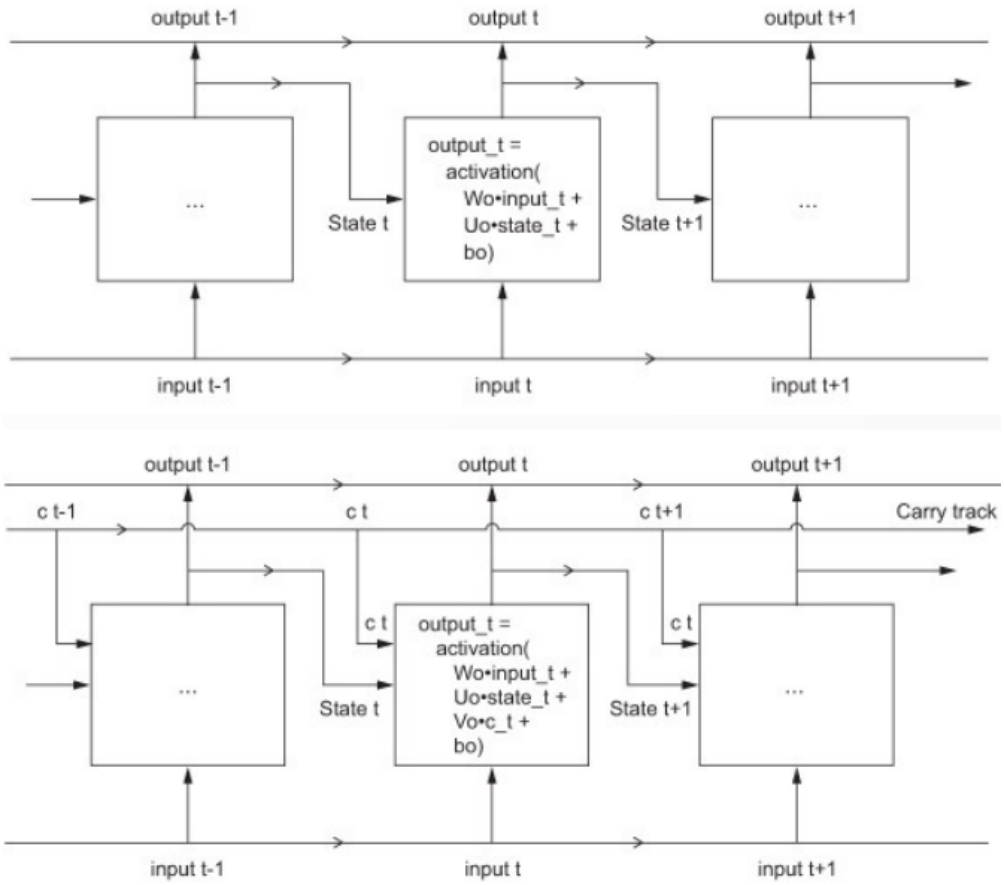


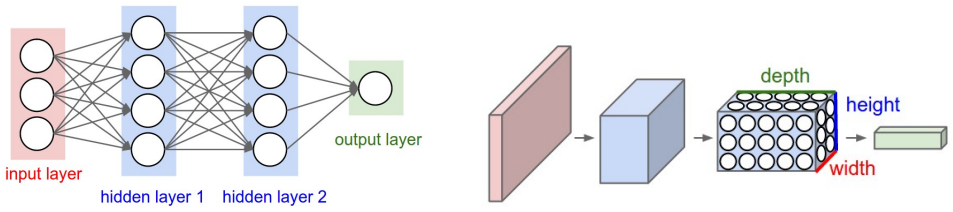
Figura 3.19: RNN vs. LSTM

Arquitectura de una red CNN

Para poder explicar correctamente la arquitectura de una CNN, volveremos a tratar por encima las RNN con un ejemplo. Imaginemos que queremos utilizar una red neuronal recurrente con una imagen RGB cuadrada de 32 de *alto*×*ancho*. En una RNN, la primera capa oculta tendría 32×32×3 = 3072 pesos. Parece una cantidad manejable, pero si la imagen cuadrada fuese de 200 (*alto*×*ancho*) en ese caso serían, 120000 pesos. Si tuviéramos varias neuronas de este tipo, habría sobreajuste casi instantáneamente. Para evitar este problema, existen las CNN, que gracias a que espera como entrada objetos que se visualizan como matrices y no necesariamente como vectores (las imágenes al fin y al cabo son vectores de píxeles). Sus capas tienen neuronas que para imágenes están compuestas de 3 dimensiones que son ancho, alto y profundidad (número de canales). A esta entrada de 3 dimensiones se le conoce como Tensor, y se le aplican ciertas configuraciones a través de capas, de manera que su salida siga siendo un elemento de 3 dimensiones.

A continuación se describe una pequeña diferencia respecto de como actúan una RNN y una CNN para una misma imagen.

Una vez introducidos en la arquitectura básica de las CNN, se han de tratar más en profundidad. Como sabemos, en *Computer Vision* las CNN están formadas por bloques, donde el más importante es la capa convolucional, en la que tenemos *kernels* (conocidos



(a) RNN (b) CNN

Figura 3.20: RNN vs. CNN sobre una imagen del mismo tamaño

también como filtros) que nos permiten aprender diferentes representaciones de imágenes y extraer características que logran que sea más fácil clasificar estas imágenes. La figura 3.21 muestra como las representaciones más profundas de la imagen principal, pueden lograr que la CNN aprenda y clasifique más fácilmente.

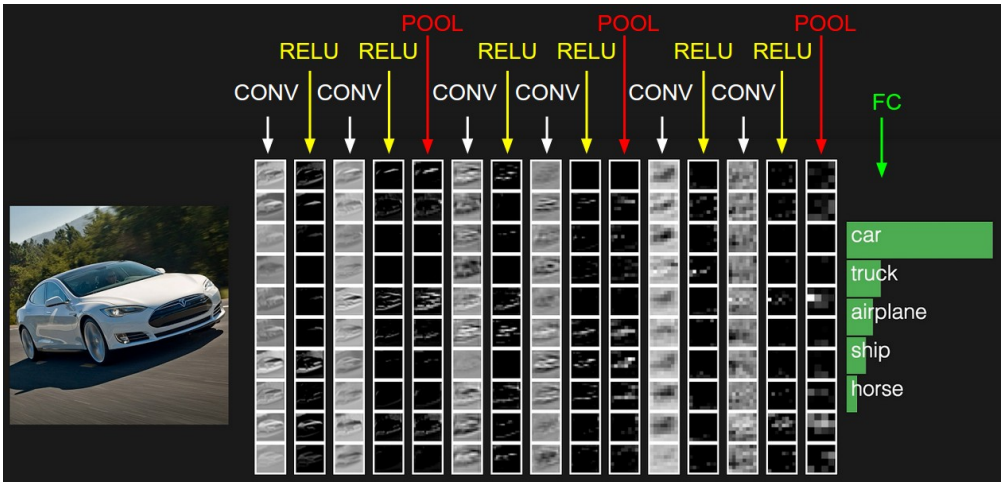


Figura 3.21: Arquitectura CNN para la identificación de un vehículo. Obtenido de CS231n

A continuación, se describen los bloques principales que son necesarios para formar cualquier CNN [38], y las arquitecturas de red más comunes para los principales problemas a solucionar mediante *Computer Vision*. Para explicarlo, se ha tomado como referencia un artículo de introducción a redes convolucionales de GeeksForGeeks [39].

1. La **imagen de entrada** es la capa de entrada de nuestro modelo. En las redes convolucionales, de manera general, se suele esperar tener como entrada una imagen, o una secuencia de las mismas. En el caso de la arquitectura anterior, esta primera capa contiene una imagen con un ancho de 32, un largo de 32 y una profundidad de 3, que en este caso es el número de canales RGB. Esas imágenes se han de normalizar al intervalo [0,1]. Véase la Figura 3.22 como ejemplo.

2. Las **capas convolucionales** son las capas que se usan para extraer las características de la imagen de entrada. Para ello aplica ciertos filtros o *kernels* que van recorriendo la imagen poco a poco. A continuación, se obtiene un mapa de características que genera una nueva imagen. Sobre la arquitectura presentada anteriormente, si se le aplicasen un total de 12 *kernels*, tendríamos una nueva

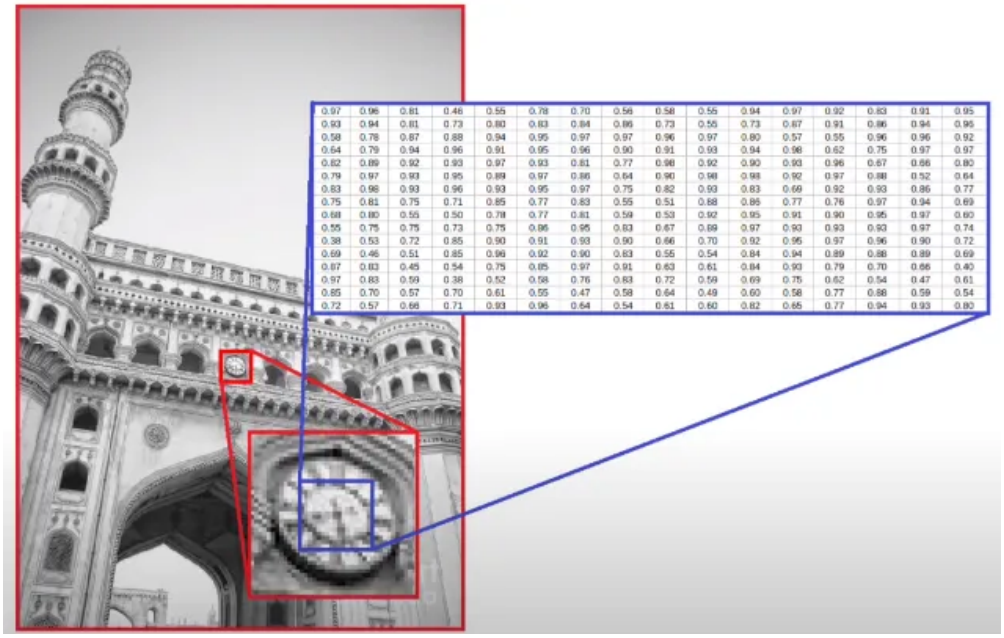


Figura 3.22: Visión de una imagen por un computador. Obtenido de NPTEL[40]

imagen o salida, con una dimensión de 32 de alto, 32 de ancho y 12 de profundidad.

3. Las **Capas de activación** se utilizan para agregar la característica de **no linearidad**. Se suelen aplicar tras la aplicación de una capa convolucional, y mantiene las mismas dimensiones que tenía de entrada. Las más habituales son la función de activación ReLU: $\max(0, x)$ [41], Sigmoide y Softmax (ya presentadas anteriormente).

4. Las **capas de pooling** cuya función es la de reducir la resolución de la imagen (conocido como *downsampling*), se insertan cada cierto tiempo en las redes, logrando así reducir memoria, parámetros y evitar el sobreajuste. Esta reducción afecta a la dimensión de alto y de ancho. Hay dos formas en las que se puede realizar el pooling, aplicando *Max Pooling* o *Average Pooling* [42]. Habitualmente, suele ser *Max Pooling* de 2x2, que en la arquitectura anterior haría que tuviéramos unas dimensiones de 16 de ancho, 16 de alto y 12 de profundidad.

5. **Capas totalmente conectadas**. Son capas en las que, de la misma manera que en las redes neuronales recurrentes, toman como entrada todas las de la capa anterior y están conectadas a cada neurona. Se utilizan para realizar finalmente tareas de clasificación y/o regresión.

6. **Capa de salida.** Como capa final, se le pasa a la salida de la capa totalmente conectada una función como Softmax o Sigmoide que nos da una puntuación de forma probabilística de pertenencia a cada clase.

La figura 3.21 representa correctamente los pasos explicados con anterioridad, que habitualmente suele ser la estructura más básica de cualquier red CNN. Asimismo, en la siguiente sección se explicarán varios tipos de redes CNN que se utilizarán en este Trabajo de Fin de Grado.

3.4. Inpainting

El inpainting es el proceso de restauración de imágenes en el que se restauran los píxeles faltantes de una imagen digital de una forma realista y manteniendo el contexto de la imagen original. Realmente, este término no es nuevo. Los artistas hace años ya restauraban a mano pinturas o fotografías dañadas con arañazos, grietas, manchas, etc. tratando de mantener su calidad original. (En la figura 3.23 se puede ver como se realizaba.)

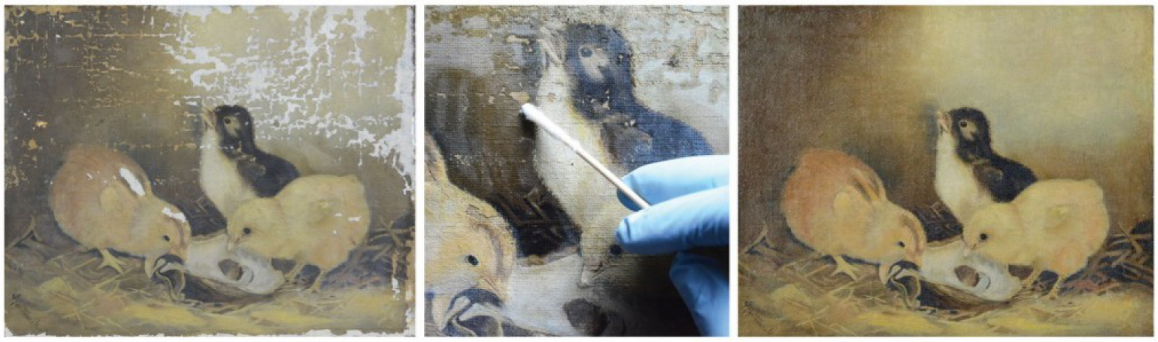


Figura 3.23: Restauración de una imagen antes de la informática. [43]

Algunas aproximaciones utilizadas son las siguientes:

- **Síntesis de Textura Basada en Ejemplares**: Utiliza parches de imagen para poder reconstruir las áreas dañadas, gracias a la autosimilitud se estiman los valores de los píxeles faltantes. Podemos ver el origen de esta técnica en la siguiente cita [44].
- **Síntesis de Estructura Basada en Ejemplares**: En este caso, se busca continuar con la estructura marcada por los píxeles vecinos para poder guiar la posterior reconstrucción. Si se quiere ver los estados del arte, consultar las siguientes referencias: [45][46][47][48]
- **Métodos de difusión**: Propagan, mediante ecuaciones diferenciales parciales, características locales de áreas cercanas hacia las dañadas. Si se quiere ver los estados del arte, consultar las siguientes referencias: [45][49][50][51][52][53][54]
- **Métodos de Representación Dispersa**: Estos métodos asumen que las imágenes tienen señales naturales que permiten una descomposición dispersa. Es decir, funcionan como un puzle, toman la imagen dañada y buscan de entre muchas opciones posibles la que podría ser la pieza más eficiente y precisa. Si se quiere ver los estados del arte, consultar las siguientes referencias: [55][56][57][58][59][60]

Los métodos presentados asumen da alguna manera que la forma de las regiones conocidas y desconocidas tienen una representación similar, y solo da buenos resultados en imágenes con pequeñas áreas a restaurar. Técnicamente, para reconstruir la imagen dañada, se puede usar la interpolación de los píxeles vecinos, logrando que no se reconozca si la imagen ha sido reconstruida o si es original. Todo esto es posible gracias a la información conocida previamente para rellenar cada una de las regiones desconocidas. [61].

Con los avances de la capacidad de cómputo, estas técnicas han ido avanzando de una forma notable. Por ejemplo, cada vez hay más software de edición de imagen para la restauración y modificación de las mismas. Llegando hasta el punto actual, en el que el inpainting mediante *Computer Vision* se ha convertido en el **SOTA** de la restauración de imágenes. Anteriormente, existían otros métodos más tradicionales que no aplicaban *Deep Learning* y que también se consideraban *Computer Vision*. Con la aparición de las CNN y uno de sus subtipos de redes, las GAN, se estableció un nuevo marco de referencia para la mejora de imagen mediante el inpainting. Estas redes utilizan las convoluciones para capturar abstracciones de la imagen, que gracias a la perdida adversarial, se pueden obtener abstracciones de datos con una alta dimensionalidad, una reconstrucción y posterior mejora de la imagen reconstruida.

3.4.1. Tipos de redes utilizadas

Dentro del *Deep Learning*, existen varios estructuras de redes CNN utilizadas. Entre ellas, las utilizadas como estado del arte en técnicas de inpainting suelen ser principalmente las tres siguientes.

Encoder-Decoder

Estas redes dividen su estructura en dos partes, el Encoder y el Decoder (codificador y decodificador). Cada una de estas dos partes tiene su propósito.

- **Encoder**. Se encarga de comprimir los datos de entrada, reduciendo su dimensionalidad y el peso, logrando obtener una representación con los datos más relevantes.
- **Decoder**. Su función es la contraria, en este caso, se toma como entrada la representación de bajo nivel de la salida del encoder, una vez tomada, va aumentando la dimensionalidad hasta obtener una representación de alto nivel similar a la original.

Este tipo de redes, permiten la adición de mecanismos de atención [62][63] para que la red se concentre en partes específicas de la entrada. Aparte de estos mecanismos, también se les puede añadir conexiones residuales (*skip-conections*)[64][65] y bloques residuales[64] para intentar mejorar su rendimiento.

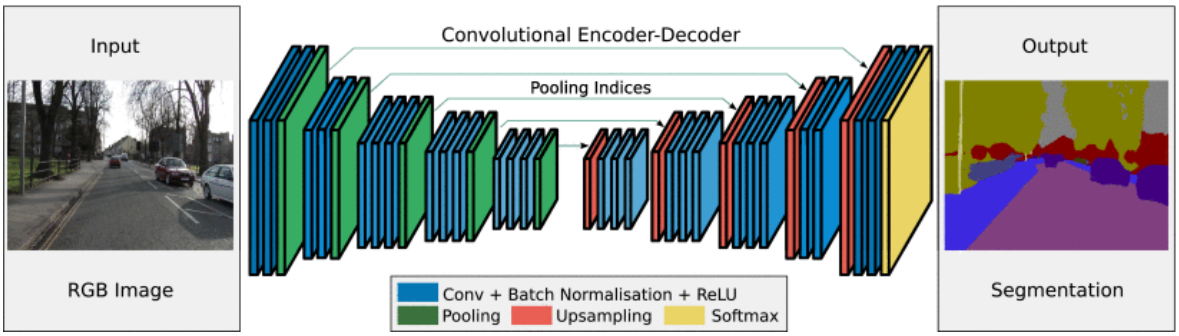


Figura 3.24: Arquitectura SegNet. Tomado de [66]

Gracias a los papers "*Understanding Geometry of Encoder-Decoder CNNs*"[67] y "*Understanding How Encoder-Decoder Architectures Attend*"[68] se puede ver en detalle el funcionamiento interno de estas redes con lo mencionado anteriormente (aplicación de bloques residuales y mecanismos de atención). En la figura 3.24 podemos ver la arquitectura de la red *SegNet* que utiliza la geometría de las redes Encoder-Decoder.

Generative Adversarial Networks (GAN)

Las redes GAN [69] se componen de dos modelos Convolucionales con una red generadora y una discriminadora, no uno único como es habitual. La red discriminadora observa la imagen original y la compara con la imagen generada por la red generadora. Es decir, evalúa si los datos que recibe son reales o generados. La red generadora se encarga de crear datos similares a los reales, su objetivo es crear una imagen lo más real posible para tratar de engañar a la red discriminadora. Estas dos redes se entrenan de forma simultánea con un proceso adversarial. La red Generadora intenta que la Discriminadora cometa los máximos errores posibles, mientras que esta última simplemente trata distinguir lo real de lo falso. Este proceso es siempre así hasta que finalmente la red Generadora recrea con tal precisión que la red Discriminadora ya no es capaz de distinguir que es real y que no. En la figura 3.25 podemos ver un diagrama de como es la estructura de este tipo de redes con la que podremos entender correctamente su funcionamiento.

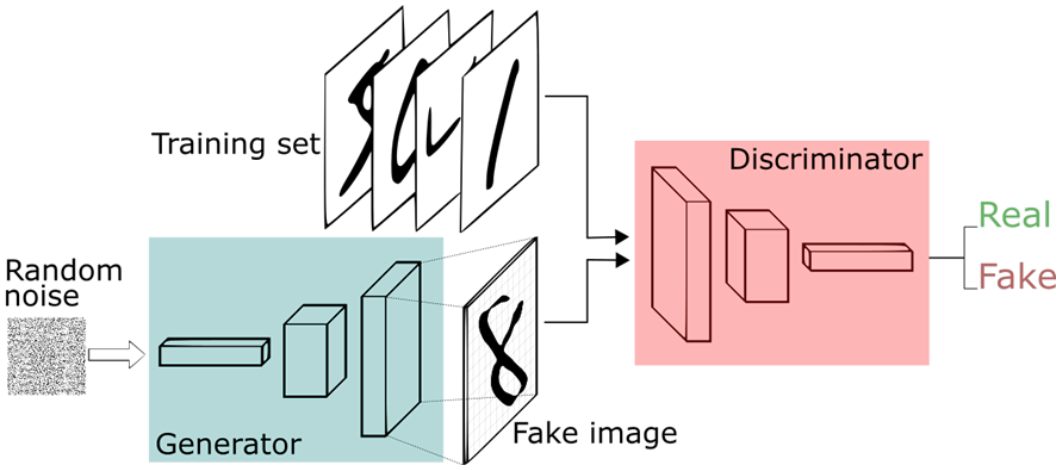


Figura 3.25: Funcionamiento de una red GAN. Tomado de [70]

U-Net

La U-Net se dio a conocer por primera vez en el paper *"U-Net: Convolutional Networks for Biomedical Image Segmentation"* [71]. Es una red CNN cuya estructura es similar a la de Encoder-Decoder, pero tiene forma de "U". Sigue el mismo camino que la Encoder-Decoder, pero tiene unas características concretas. Su arquitectura consta de las mismas dos partes que la Encoder-Decoder y cada una hace lo siguiente.

- **Encoder**: Mediante la aplicación de convoluciones, funciones de activación y *poolings*, en repetidos casos, se logra contraer la imagen (fase de *downsampling*,

capturar contexto de la misma y reduce la resolución espacial. Concretamente, con el uso de capas convolucionales se extraen las características más relevantes de la imagen de entrada y con el *pooling* se reduce la dimensión espacial. Pese a ello, se pierde cierta información de bajo nivel.

- **Decoder**: Durante esta etapa, se aplica la expansión, mediante deconvoluciones, es decir, *upsampling*, en el que se aumenta la resolución espacial.

Se puede pensar que las redes Encoder-Decoder y la U-Net, parecen lo mismo. Pero la UNet, durante el decoder, también toma características aprendidas durante la etapa de codificación, de manera que la red es capaz de reconstruir de una manera más detallada la imagen, combinando características de alto y bajo nivel. Para ello concatena la salida del *Encoder* con la entrada del *Decoder* de su mismo nivel.

En la figura 3.26 podemos ver la estructura de la arquitectura de una U-Net, en la que se distingue perfectamente el proceso en forma de **U** que sigue la red para hacer las dos fases, la de *downsampling* y *upsampling* junto con la concatenación de ambas.

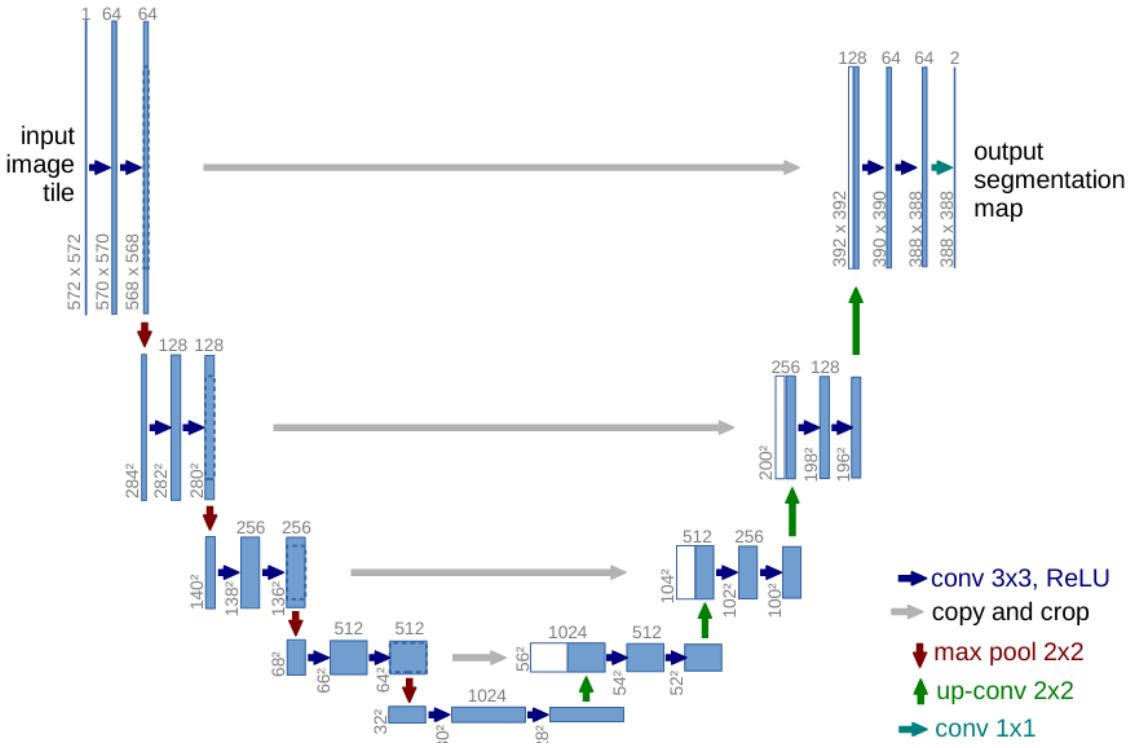


Figura 3.26: Arquitectura de una red U-Net. Tomado de [71]

3.4.2. Estado del arte (SOTA)

En esta sección se resume el SOTA para cada uno de los principales conjuntos de datos más utilizados en inpainting. Se verán sus principales características y/o ventajas y finalmente, enseñaremos su estructura. Se pueden consultar en detalle varias de las métricas mencionadas en el anexo A.

Paris StreetView

El SOTA para este dataset, de H. Liu *et al.* [72], propone una capa de atención semántica coherente conocida como *CSA* para mantener la estructura contextual y predecir partes faltantes de las imágenes. Se compara para mostrar su mejora respecto a otras redes que utilizan atención contextual.

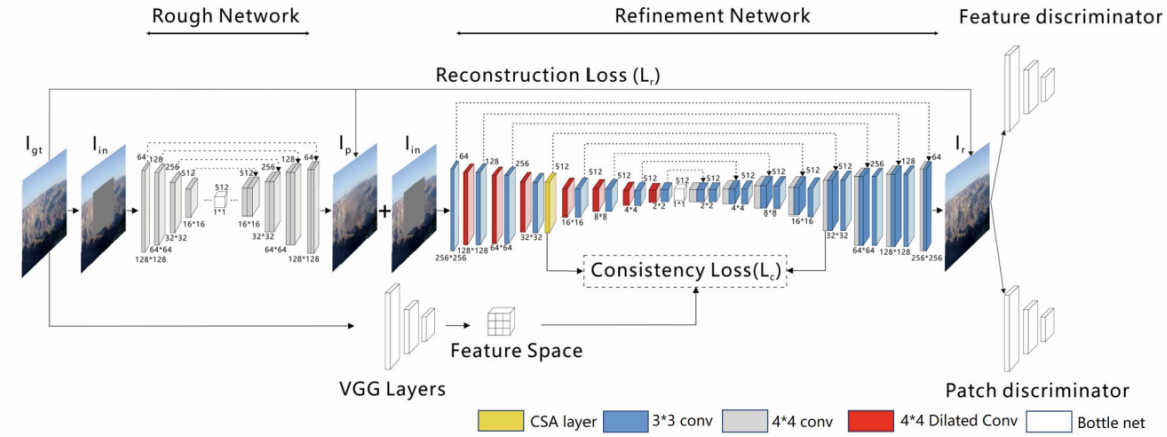


Figura 3.27: Arquitectura de red CSA [72]

Places2

Para este dataset, la red *CM-GAN* [73] creada en 2022, propone una nueva arquitectura de red generativa que mejora la síntesis de estructuras y detalles locales. Para ello, utiliza una modulación global seguida de una espacial para crear de forma coherente y el área faltante en cada imagen.

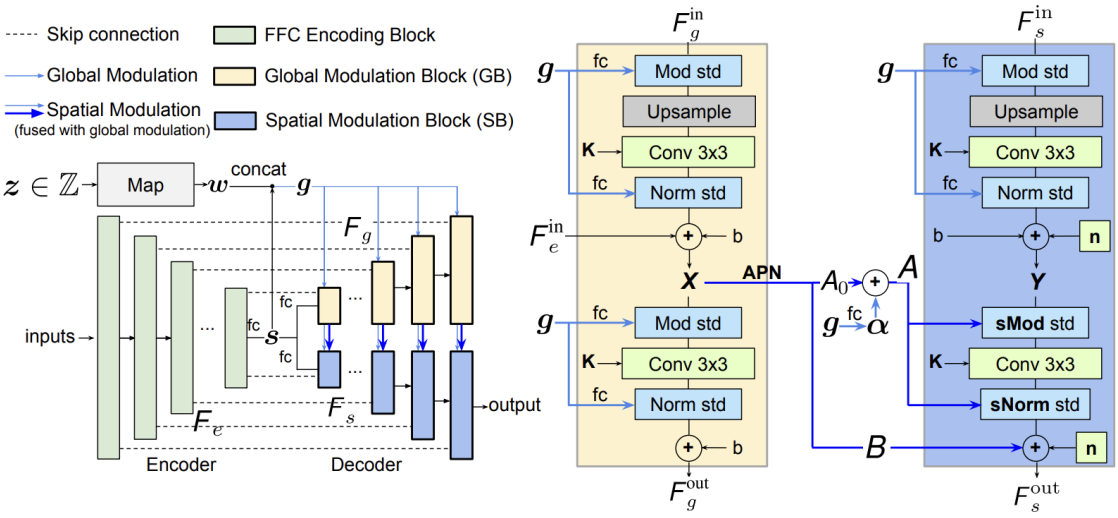


Figura 3.28: Arquitectura CM-GAN [73]

CelebA-HQ

Para este dataset de rostros humanos, la red se basa en el uso de transformers [74] para el relleno de grandes áreas faltantes. La red MAT [75] mediante el uso de convoluciones y atención contextual, logra una gran eficiencia y producción de múltiples resultados a la hora de rellenar la imagen. Su estructura es la siguiente.

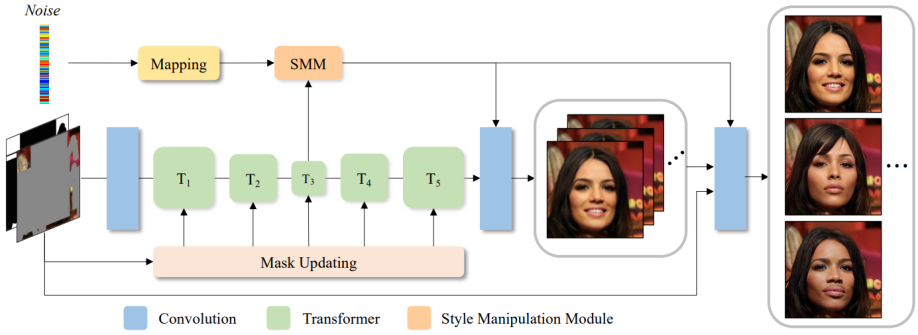


Figura 3.29: Arquitectura de la red MAT. [74]

ImageNet

En 2023, Jeevan *et al.* [76] crearon una red supervisada, que se situó como el SOTA para el dataset de *ImageNet*. La red *WavePaint* [76] destaca en su eficiencia con tan solo 5 millones de parámetros. Así, supera en resultados a otras redes que hasta ese momento eran el SOTA como *LaMa* con 27 millones y *CoModGAN*[77] con 109 millones respectivamente.

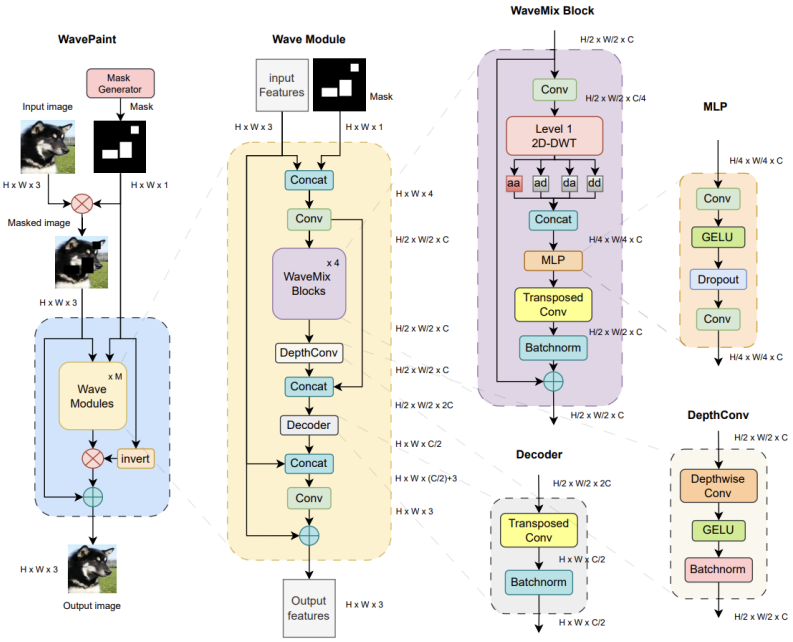


Figura 3.30: Estructura detallada de la red WavePaint [76]

3.5. Aportación personal

Dentro del ámbito de la visión artificial, este Trabajo de Fin de Grado propone una solución con *Deep Learning* mediante el uso de redes convolucionales y recurrentes, con el objetivo de encontrar una red que obtenga buenos resultados con técnicas de inpainting (3.31). Siendo justificable y medible gracias a la toma de métricas utilizadas en el campo de *Computer Vision* (CV) como son el error cuadratico medio (MSE), *Peak Signal-to-Noise Ratio* o Proporción Máxima de Señal a Ruido (PSNR) y *Structural similarity index measure* o en español medida del índice de similitud estructural (SSIM). Para más detalles sobre estas y otras métricas, lo que representan y como funcionan, consultar el Apéndice A.

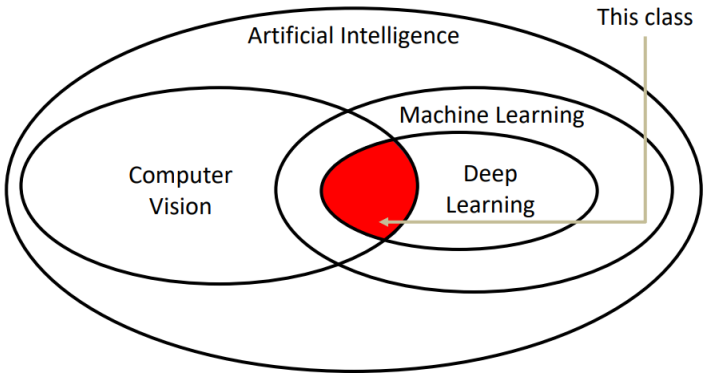


Figura 3.31: Donde se sitúa el inpainting en el saco de la IA

4. Framework

Este capítulo trata la infraestructura tanto a nivel de hardware como a nivel de software utilizada como base de nuestro proyecto de inpainting. En él hablaremos del sistema en general, el entorno de desarrollo especifico, librerías y dependencias utilizadas. Finalmente, trataremos cuales son las instrucciones de despliegue.

4.1. Configuración del Sistema

Esta configuración se ha probado tanto en sistemas Linux [78] como sistemas Microsoft Windows [79], pero por eficiencia y gestión de recursos el sistema operativo utilizado ha sido Linux, con la distribución **Ubuntu 22.04 LTS** [80]. Esta configuración anterior se ha utilizado en varios sistemas, pero principalmente, se ha hecho uso de dos. Sus características se van a ver en la siguiente tabla:

Tabla 4.1

Especificaciones del Hardware

CPU	**Cores**	**RAM**	**GPU**	**VRAM**
i7-12700F	12	32 GB	RTX 4090	24 GB
i7-12700F	12	16 GB	RTX 3060	12 GB

4.2. Entorno de desarrollo

4.2.1. Lenguaje de Programación

Para el desarrollo de este Trabajo de Fin de Grado, se ha utilizado **Python 3.10** [81] como lenguaje de programación. Es uno de los lenguajes más populares y versátiles para *Machine Learning* y *Computer Vision* gracias a todas las bibliotecas que se han desarrollado para ello.

También se ha elegido debido al conocimiento y aplicación anterior en este ámbito durante la asignatura Minería de Datos [82] del grado.

4.2.2. Entornos Virtuales

Para gestionar las dependencias correctamente y asegurar que este Trabajo pueda ser reproducido correctamente, se ha creado un directorio virtual utilizando el módulo **venv** [83] que nos permite crearlos de manera aislada. En ellos podemos gestionar todos los paquetes y dependencias que queramos sin afectar al de otros proyectos, ya que con esto evitamos tocar en las dependencias del sistema.

También sirve para poder tener varios directorios virtuales con distintas versiones de paquetes o librerías según lo que necesitemos hacer en ese momento, evitando entre ella conflictos.

4.3. Dependencias y Librerías

Presentamos las librerías y distintas dependencias que son necesarias para poder abordar este Trabajo de Fin de Grado.

4.3.1. Librerías de visión artificial

Hay varias librerías utilizadas para el tratamiento de imágenes en Python, en este caso se han utilizado:

- **OpenCV (v4.8.1.78)**: Es una librería que, como su nombre indica OpenComputerVision, es de código abierto y está destinada al uso de visión artificial, se usa para preprocesamiento de imágenes y muchas otras utilidades. [84]
- **Pillow (v10.0.1)**: Al igual que OpenCV, sirve para lectura y guardado de imágenes en distintos formatos y para el preprocesamiento de imágenes. [85]
- **MatPlotlib (v3.7.3)**: Matplotlib es una librería Python *open source* que permite crear visualizaciones de datos [86]. Se utiliza para mostrar resultados y métricas.
- **Ultralytics (v8.1.8)**: Ultralytics es una librería de Python que implementa de manera eficiente el modelo preentrenado YOLO (*You Only Look Once*) para detección y segmentación de objetos en tiempo real (ver Anexo C).

4.3.2. Frameworks de Deep Learning

Para poder construir y entrenar modelos de *Deep Learning*, se ha utilizado la librería PyTorch [87], que es un framework robusto y eficiente con abstracciones de alto nivel en redes neuronales complejas.

4.3.3. Gestión de Dependencias

Se ha utilizado el gestor de paquetes de Python `pip` y las librerías correspondientes con el archivo ***requirements.txt***.

- **pip (22.0.2)**: Es el gestor de paquetes de Python, permite instalar y gestionar las dependencias y librerías.
- **requirements.txt**: Es un archivo de texto, que contiene la lista de todas las librerías que se necesitan en el proyecto. Una vez preparado ese archivo, con el comando "*pip install -r requirements.txt*" se instalarán estas dependencias una a una.

4.4. Instrucciones de despliegue

A continuación, se describen las instrucciones de instalación del sistema sobre la infraestructura física descrita anteriormente.

4.4.1. Despliegue y ejecución

Se debe de tener un driver para la correcta conexión del sistema con la gráfica, en este caso se ha utilizado la versión **550.54.15** [88], con soporte para **CUDA 12.4** [89] Es necesario tener instalado **Python 3.10** y todas las dependencias del *requirements.txt*. Es importante comprobar si el intérprete de Python detecta correctamente la gráfica, o si no logra una conexión con la misma.

4.4.2. Monitoreo

Se utilizarán las librerías **MatplotLib** [90] y **wandb** (*Weights and Biases*). Ambas se utilizan para visualizar resultados mostrados en algún tipo de *plot*, la diferencia es que MatplotLib necesita un entorno gráfico, mientras que *wandb* no, ya que permite visualizar las ejecuciones, métricas y monitoreo de todo tipo de datos del sistema desde su web [91].

5. Conjuntos de imágenes (*datasets*)

En este capítulo se describirá y se podrán ver los conjuntos de imágenes utilizados, ya sean generalistas o específicos para la conducción autónoma. Estos *datasets* se han usado para entrenar y probar tanto los modelos de redes para inpainting desechadas como el modelo final. También se presentarán las modificaciones que han sido realizadas en cada uno de ellos y por último, las características de cada uno.

5.1. Conjuntos utilizados

Se han utilizado tres conjuntos de datos. De los cuales, los más utilizados debido al uso de imágenes reales han sido el primero y el último. A continuación, se presenta en profundidad las características de cada uno de ellos.

Figura 5.1: Una única imagen - 3 objetos removidos. Dataset *DEFACTO*.

5.1.1. DEFACTODataset

El conjunto de datos *DEFACTODataset*[92], ha sido extraído de la plataforma Kaggle, y a su vez recreado de un subconjunto de imágenes de *MSCOCO*[93]. *DEFACTO*, consta originalmente de 25001 imágenes en formato TIF, con sus correspondientes anotaciones. Se dividen en las carpetas *inpaint_mask* con sus, 25001 máscaras correspondientes. La carpeta *prove_mask* sigue la misma estructura. Por último, la carpeta *graph* contiene 25001 archivos JSON que referencian el nombre de la imagen con el inpainting ya aplicado y su correspondiente referencia a la imagen original en el dataset *MSCOCO*. [1]

Debido al gran tamaño del conjunto de datos *MSCOCO*, se ha optado por utilizar el conjunto *COCO minitrain* [94] para obtener las imágenes originales, ya que estaba compuesto de 25000 imágenes.

5.1.2. Virtual KITTI V2

El conjunto de datos *Virtual Kitti V2* [95] no utiliza imágenes reales, sino creadas de forma artificial aplicado a la conducción. VKITTY2 contiene imágenes con la perspectiva frontal de un vehículo. Hay un total de 2156 frames de imágenes como el de la figura 5.2, que están extraídas de cinco escenas distintas por dos cámaras diferentes. Esas escenas están recreadas en muchos escenarios diferentes, como puede ser: noche, día, escenario lluvioso, nevado, con niebla, etc. Algunos de estos escenarios se presentan en la figura 5.4.
A su vez, nos ofrecen esas imágenes con rotaciones (ver figura 5.5), de 15 a 30 grados, tanto a la izquierda como a la derecha. Resumiendo, tenemos 2156 imágenes, para cada uno de los escenarios mencionados y sus correspondientes imágenes segmentadas tanto por clase como por instancias (ver fig. 5.3).

Figura 5.2: Una escena de VKITTI2.

[1]En muchos casos, de una única imagen original, hay varias correspondencias en las 25000 de *DEFACTO*, ya que "inpaintea"distintos objetos de una misma imagen. Como se puede ver en la figura 5.8

Figura 5.3: Segmentación de clases y de instancias en VKITTI2.

Figura 5.4: Variaciones artificiales del clima o la hora del día en VKITTI2.

Figura 5.5: Variaciones artificiales de la rotación de la cámara en VKITTI2.

5.1.3. BDD100K

Es un conjunto de datos de vehículo autónomo que, como podemos ver en *arXiv* [96], está formado por 100000 vídeos y más de 10 tareas para evaluar el progreso de algoritmos en el reconocimiento de imágenes para la conducción autónoma.

En nuestro caso concreto, se ha tomado el subconjunto de datos *100k images* que está formado por frames extraídos cada 10 segundos del su video original. Estas 100000 imágenes JPG, se dividen en 70000 para entrenamiento, 20000 para validación y 10000 para prueba.

A su vez, se han tomado varios subconjuntos más, como son *Instance Segmentation* y *Drivable Area* (área conducible) (ver figuras 5.6 y 5.7). Más adelante veremos la utilidad de estos dos subconjuntos de datos.

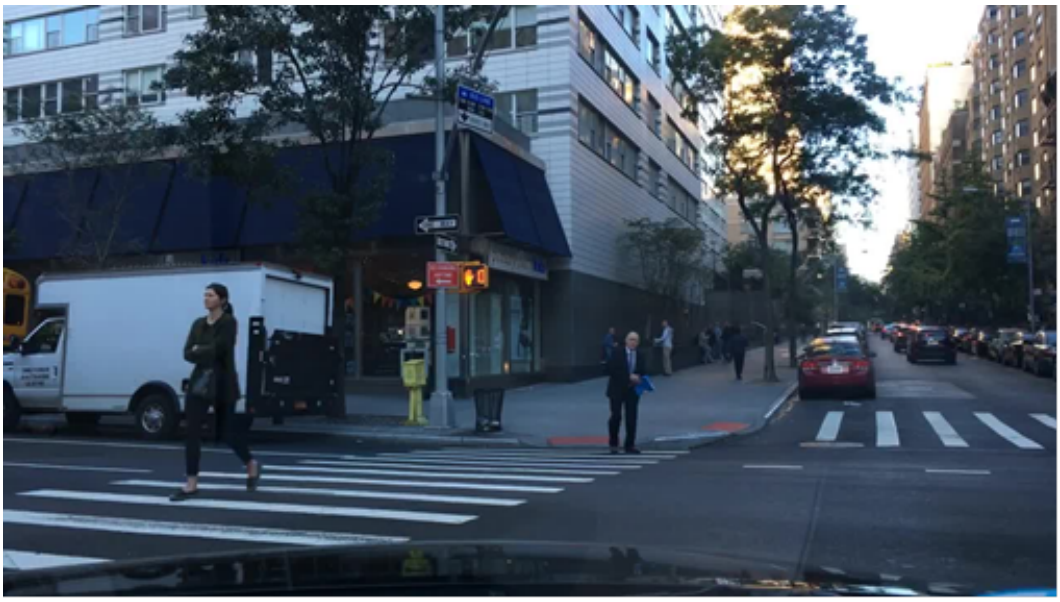


Figura 5.6: Una escena de BDD100K.

Del área conducible, tenemos el mismo número de imágenes que las originales, es decir, su par. Por otro lado, en para las imágenes de *Instance Segmentation*, tenemos un total de 10000 imágenes nada más.

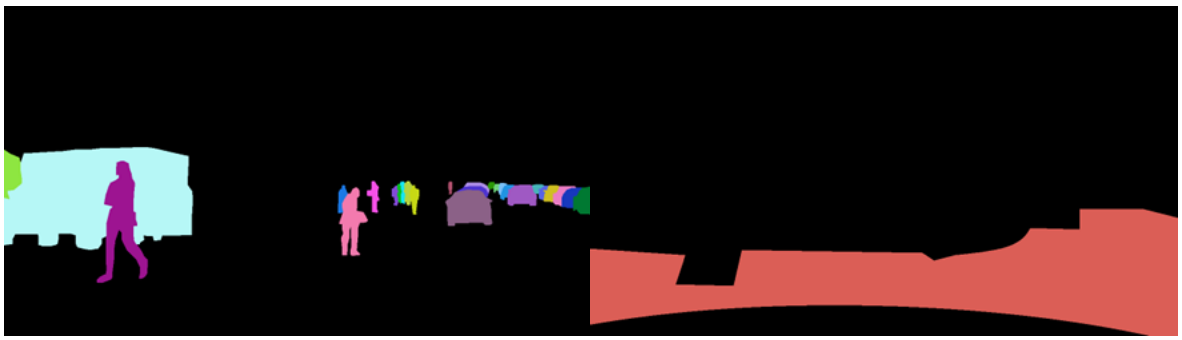


Figura 5.7: Segmentación de instancias y Drivable Area en BDD100K

5.2. Modificaciones

A los conjuntos presentados anteriormente, hay que aplicarles ciertas modificaciones para poder adaptarlos a nuestra tarea, el inpainting. Han de tener una estructura concreta que forma un trío de imágenes. Todas las modificaciones y problemas que hemos tenido durante este proceso, se van a explicar aquí para cada uno de los tres datasets.

5.2.1. DEFACTO

Para tomar las imágenes *Ground Truth* (imágenes objetivo), se ha utilizado el dataset reducido *COCO minitrain*.

Esta elección tuvo sus pros y sus contras, ya que reducía considerablemente el número de imágenes con el que trabajar, pero, por otro lado, las 25000 imágenes, no coincidían con las de *DEFACTO*.

Por ello, se han mantenido solo las coincidentes, siendo un total de 5121 imágenes inpainteadas, con sus máscaras y *ground truth* correspondientes.

De esas imágenes solo se tomó el mayor subconjunto con las mismas dimensiones, en este caso, $640x480$ de alto por ancho. De esta manera, el subconjunto final para entrenar la red inicial constaba de 1426 imágenes con sus máscaras y *ground truth*.

Una vez seleccionadas las 1426 imágenes, se ha aplicado la máscara binaria que incluía el dataset, invirtiéndola y multiplicándola por la imagen. De manera que se obtiene la estructura deseada para el entrenamiento futuro de la red, formada por tres carpetas, que son `inpainted_images`, `masked_images` y `merged_mask`.

Tomaremos como imágenes *ground truth* las imágenes inpainteadas tomadas de *DEFACTO*.

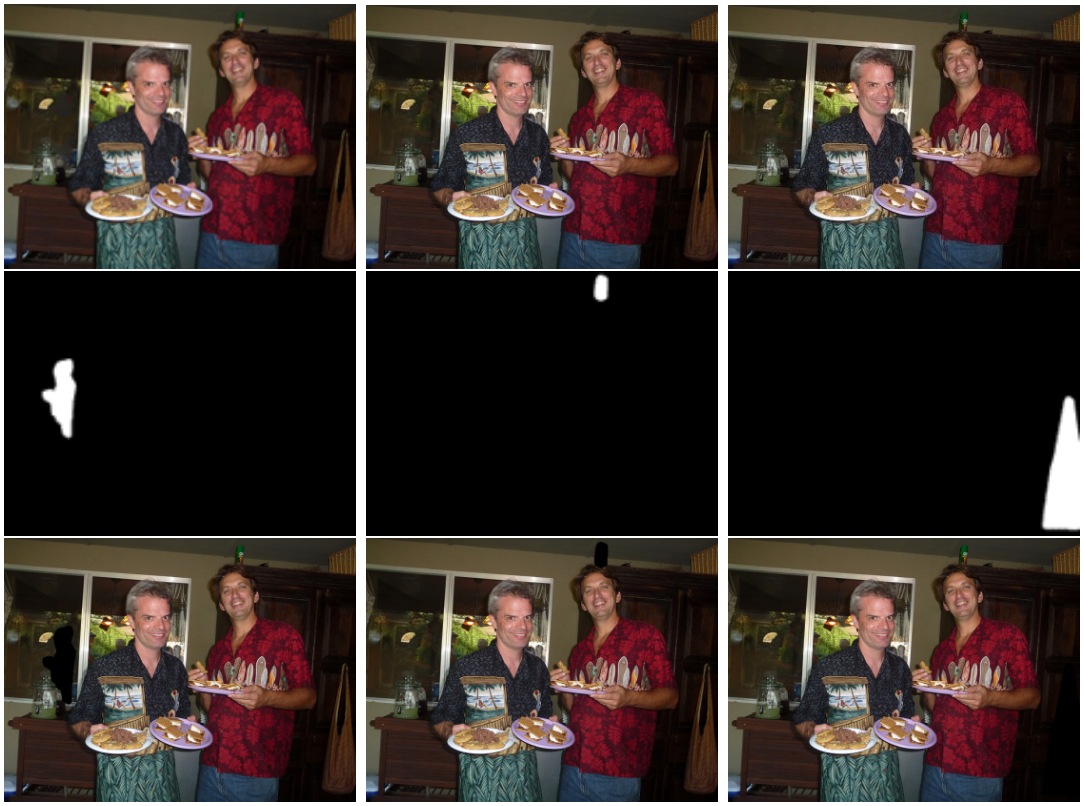


Figura 5.8: *Ground truth*, máscara binaria, y mascará aplicada a imagen original.

5.2.2. VKITTI2

Como VKITTY2 nos ofrecía varios tipos de segmentaciones, gracias a la de clases, se han extraído todos los vehículos y su máscara correspondiente.

Figura 5.9: Extracción de clase vehículo segmentada en *VKITTI2*.

Una vez extraídos todos, se han seleccionado cada uno por separado gracias a la segmentación de instancias.

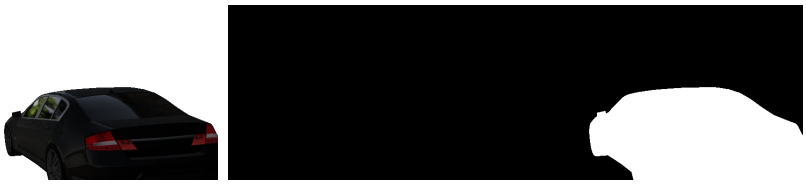


Figura 5.10: Extracción de vehículos únicos.

Después, se ha extraído la máscara de las carreteras en cada imagen original de la segmentación por clases.

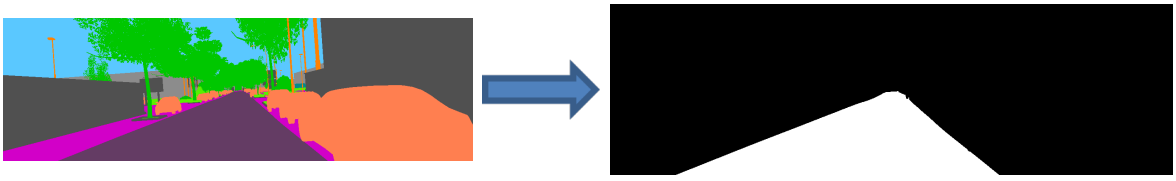


Figura 5.11: Extracción de la clase carretera.

Por último, se han añadido de forma aleatoria de uno a seis vehículos extraídos anteriormente de forma artificial en las imágenes de origen, con la condición de que toquen la zona correspondiente a la calzada, usando su máscara extraída también con anterioridad. De esta forma obtenemos una imagen con esos vehículos pegados encima de la imagen de manera artificial.

A partir de esta imagen creada (ver figura 5.12), se han guardado 3, la *ground truth* (la original anteriormente), la máscara binaria de los vehículos artificiales, y por último, esa máscara invertida multiplicada por la imagen inpainteada (ver figura 5.13).

Figura 5.12: Imagen con vehículos añadidos de forma artificial.

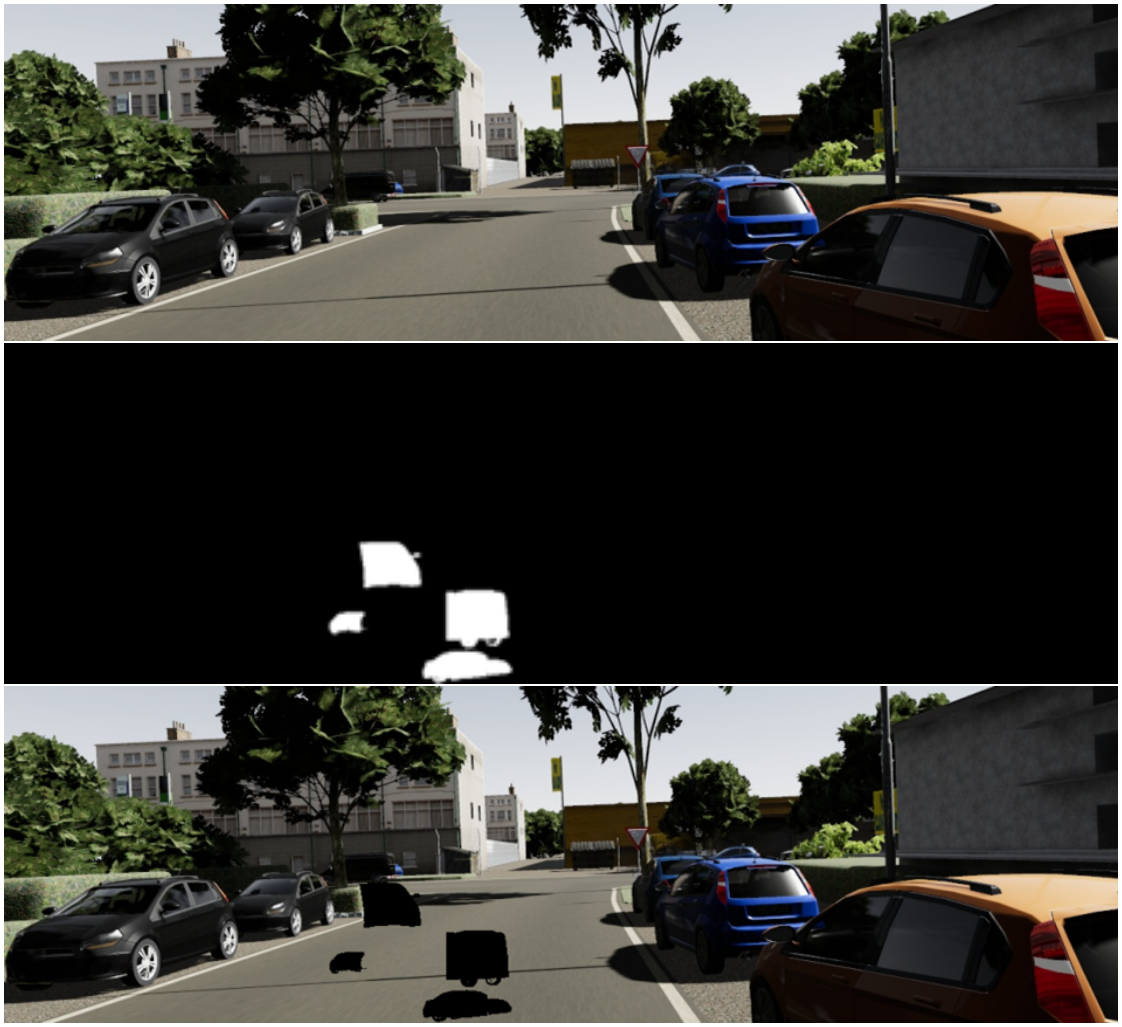


Figura 5.13: Imagen inpainteada, máscara binaria y mascará invertida aplicada por encima.

Siguiendo el patrón anterior, se logró obtener de manera aproximada un total de 11000 tríos de imágenes.

5.2.3. BDD100K

De las 10000 imágenes que contienen la segmentación, se han tomado por separado cada una de las instancias (habitualmente vehículos y personas).
Ahora, en lugar de tomar todas las instancias, se filtró para extraer únicamente las que estaban completas.
Para ello, se extrajeron las instancias que estaban aisladas (que no se tocaban entre sí con otras).
Esto daba lugar a un número muy bajo de instancias para el total de imágenes que necesitaba, siendo de menos de 1000 instancias.

Finalmente, quedaban por extraer las instancias completas que no estaban aisladas, y para ello, se buscaron los conjuntos de instancias que se tocaban entre sí, y se guardó solo la de mayor área, ya que según la perspectiva de la cámara debería de ser la instancia más cercana y que, por lo tanto, no tendría ningún obstáculo por delante.

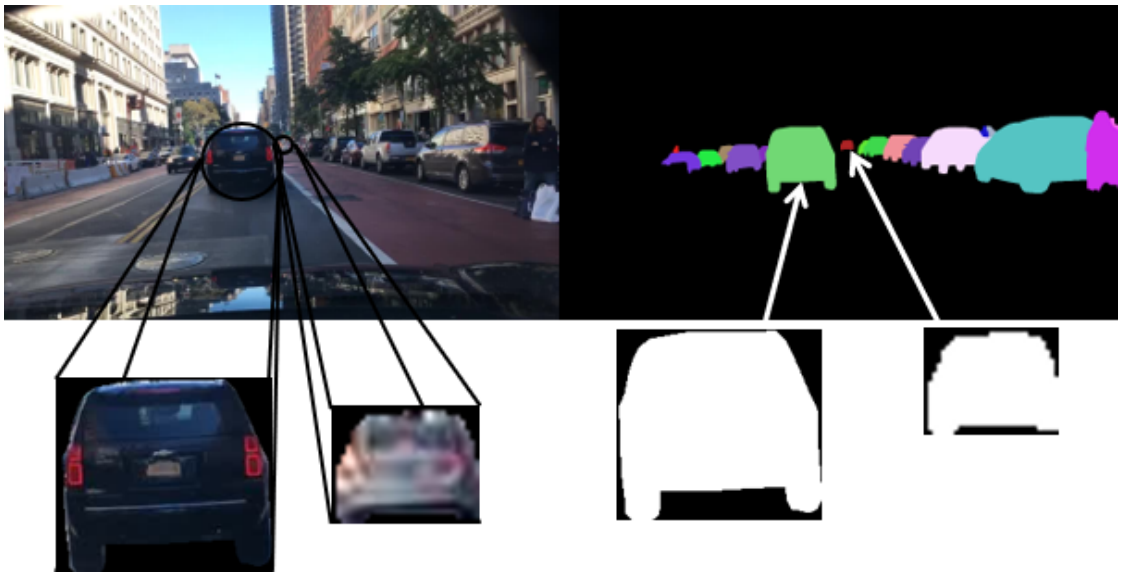


Figura 5.14: Selección de vehículos válidos.

Pese al planteamiento anterior, entre las instancias tomadas, hay algún falso positivo. Por ejemplo, ciertos vehículos de la clase camión, pueden ser más grandes que alguno de la clase coche. En este caso concreto, el área del coche podría ser menor que la del camión, aun estando más cerca de nuestra perspectiva. Podemos ver un caso similar en la siguiente figura con un humano y una furgoneta.

Figura 5.15: Falso positivo en la selección de vehículos válidos.

Con lo mencionado anteriormente hemos obtenido un total de 11855 pares de imágenes.

El último paso para la creación del dataset que necesitamos, es similar al realizado con *VKITTI2*. Es decir, se han colocado de uno a seis vehículos de forma artificial en las imágenes originales, utilizando el área de carretera conducible (*road area* - máscaras de la carretera). En este caso, se ha tenido que adaptar la máscara de la carretera a un único canal, de manera que sea binaria (ver figura 5.16).

Figura 5.16: Modificación de la máscara de la carretera en BDD100K.

Una vez con la máscara de la carretera binaria, se han filtrado las instancias cuya área fuera lo suficientemente grande como para poder colocarlas de manera visible y clara. Estas instancias se han colocado en función de su tamaño, de manera que cuanto menor sea el objeto, más lejos se ha colocado. De forma más técnica, se ha aumentado o disminuido la posición en el eje Y del área transitable en el que se coloca cada instancia.

Con todo esto, ya tenemos las imágenes preparadas que, objetivamente, son mucho más naturales gracias al procedimiento anterior, en el que se ha evitado que las instancias añadidas, no estén completas y con tamaños irreales (ver figura 5.17).

Figura 5.17: Imagen con vehículos añadidos de forma artificial.

Por último, partiendo de esta imagen generada, extraeremos la que contiene las máscaras de las instancias aplicadas y las máscaras binarias. Como imagen *ground truth*, se toma la original del dataset *BDD100K*. Podemos verlo en la siguiente figura.

(a) Ground Truth (b) Máscara binaria (c) Máscara aplicada

Figura 5.18: División final de imágenes en BDD100K.

5.3. Utilidad de cada dataset

Para todos los conjuntos de datos han realizado ciertas adaptaciones para poder utilizarlos en el entrenamiento de nuestras redes de inpainting. En todos los casos, únicamente necesitamos la imagen *ground truth*, a la que tiene que llegar la red para inpaintear. También la imagen con la máscara aplicada por encima de los objetos a querer eliminar y por último, la imagen de únicamente la máscara binaria.

En el apartado anterior, para cada conjunto, se ha mostrado la imagen con las instancias añadidas de forma artificial (figuras 5.8, 5.13 y 5.18). Esto ha servido para representar en todo momento de donde procedían las imágenes que se usarían para entrenar la red. Más allá de eso, su utilidad es nula.

Si queremos entrar de forma más específica en la utilidad de cada uno de conjuntos anteriores, tenemos que recordar los dos objetivos principales de este trabajo. El primero es crear una red para inpainting y el segundo entrenará específicamente para la conducción autónoma. Para el primero hemos aprovechado el dataset ***DEFACTO***, que gracias a sus imágenes generalistas, sirve para crear la red final. Haciendo distintas pruebas, creando distintos modelos con distinto número de parámetros y aplicando distintas regularizaciones, etc.

Una vez conseguidos buenos resultados para esas imágenes, nos hemos a centrar en el segundo objetivo principal. Para ello, necesitamos que las imágenes sean de carreteras con distintos objetos a remover, como pueden ser vehículos o personas. Por ello, se comenzó a elegir y a buscar conjuntos de datos de imágenes de vehículos.

Virtual Kitti 2, ofrecía una muy buena distribución de los datos, con segmentación de objetos por clases e instancias. Esto permitía aprender a trabajar con este tipo de datos, a utilizar la segmentación ya presente en el dataset para extraer correctamente de ahí distintos vehículos y otras instancias que más tarde se colocarían intencionadamente de forma artificial sobre las carreteras.

Una vez ya familiarizado con este tipo de conjuntos, se utilizó ***BDD100K*** para entrenar el modelo final de inpainting de forma específica para el vehículo autónomo. Simplemente, una vez creado, se van a utilizar 60 mil imágenes para entrenamiento y las 6921 restantes para prueba.[2]

[2]En tareas de Aprendizaje Automático, se suele dividir el conjunto en tres partes: entrenamiento, validación y prueba. En este caso, al ser Visión Artificial, cuyo aprendizaje requiere muchísimo mayor tiempo, se ha optado por eliminar el conjunto de validación, ya que se utiliza para ajustar hiperparámetros que en nuestro caso se realizará con otros métodos.

5.4. Carga de los datos

Una vez preparados los datasets para ser entrenados posteriormente en un modelo, hemos de realizar ciertos pasos, como la transformación de imágenes a tensores para usar redes neuronales y la división en conjuntos de entrenamiento y prueba, de manera que todo pueda ser procesado correctamente, y tengamos imágenes para validar el entrenamiento. A todo esto anterior se le conoce como preprocesamiento y división del conjunto de datos.

Es imprescindible que para que las imágenes estén emparejadas entre sí, tengan el mismo nombre.

```
# path to the folder with images (.jpg, .jpeg, .png, .tif)

def cargar_paths_y_contar_imagenes(ruta_mask, mask_only, ruta_inpainted):
    tipos_imagen = ["*.jpg", "*.jpeg", "*.png", "*.tif", "*.tiff"]
    tipos_imagen.extend([tipo.upper() for tipo in tipos_imagen])  # Appendea las versiones en mayusculas

    mask_paths = []
    mask_only_paths = []
    inpainted_paths = []

    for tipo in tipos_imagen:
        mask_paths.extend(glob(os.path.join(ruta_mask, tipo)))
        mask_only_paths.extend(glob(os.path.join(mask_only, tipo)))
        inpainted_paths.extend(glob(os.path.join(ruta_inpainted, tipo)))


    return mask_paths, mask_only_paths, inpainted_paths

# Uso de la funcion
RUTA_MASK = "RUTA CON LAS IMAGENES CON LA MASCARA APLICADA"
MASK_ONLY = "RUTA CON LAS IMAGENES DE LA MASCARA BINARIA"
RUTA_INPAINTED = "RUTA CON LAS IMAGENES GROUND TRUTH"

mask_paths, mask_only_path, inpainted_paths = cargar_paths_y_contar_imagenes(RUTA_MASK, MASK_ONLY, RUTA_INPAINTED)
```

Extracto de código 5.1: Especificación de rutas y carga de paths

Como vemos en el código anterior, lo primero es especificar las rutas en las que ese encuentran las imágenes. Tras ello, se buscarán y cargarán los directorios correspondientes. Que se han de dividir en dos subconjuntos: de entrenamiento y test y/o validación, para ello utilizamos la función `train_test_split` y como podemos ver en el código 5.2, se ha tomado como ejemplo la división en 90/10 de entrenamiento/prueba.

```
TRAIN_MASK_IMGS, TEST_MASK_IMGS = train_test_split(MASK_PATHS, test_size=0.1))
TRAIN_MASK_ONLY, TEST_MASK_ONLY = train_test_split(MASK_ONLY_PATH, test_size=0.1)
TRAIN_INPAINT_IMGS, TEST_INPAINT_IMGS = train_test_split(INPAINTED_PATHS, test_size=0.1)
```

Extracto de código 5.2: División en conjunto de entrenamiento y prueba

Posteriormente, tenemos que adaptar estas imágenes a tensores para que la red sea capaz de trabajar con ellos. Para ello hemos de definir nuestra clase `ImageDataset`. (código 5.3)

```
class ImageDataset(Dataset):
    def __init__(self, mask_images_list, mask_only_list,
    inpainted_images_list, test=False):
        self.mask_imgs = sorted(mask_images_list)
        self.mask_only = sorted(mask_only_list)
        self.inpainted_imgs = sorted(inpainted_images_list)
        self.test=test

    def __len__(self):
        return len(self.mask_imgs)

    def __getitem__(self, idx):
        mask_img_path = self.mask_imgs[idx]
        mask_only_path = self.mask_only[idx]
        inpainted_img_path = self.inpainted_imgs[idx]

        # Load the mask image
        mask_rgb = Image.open(mask_img_path).convert('RGB')
        mask_rgb = np.array(mask_rgb)
        mask_rgb = (mask_rgb / 255).astype(np.float32)
        assert mask_rgb.shape[-1] == 3
        mask_rgb = torch.from_numpy(mask_rgb)
        mask_rgb = mask_rgb.permute((2, 0, 1))


        mask_bw =  Image.open(mask_only_path).convert('RGB')
        mask_bw = np.array(mask_bw)
        mask_bw = (mask_bw / 255).astype(np.float32)
        mask_bw = torch.from_numpy(mask_bw)
        mask_bw = mask_bw.permute((2, 0, 1))

        # Load the inpainted image
        inpainted_rgb = Image.open(inpainted_img_path).convert('RGB')

        inpainted_rgb = np.array(inpainted_rgb)
        inpainted_rgb = (inpainted_rgb / 255).astype(np.float32)
        assert inpainted_rgb.shape[-1] == 3
        inpainted_rgb = torch.from_numpy(inpainted_rgb)
        inpainted_rgb = inpainted_rgb.permute((2, 0, 1))

        return mask_rgb, mask_bw, inpainted_rgb


# Usage:
# dataset = ImageDataset(MASK_PATHS, INPAINTED_PATHS)
```

Extracto de código 5.3: Clase ImageDataset

En el código anterior, podemos ver que partimos de las listas de imágenes que contienen las imágenes, sus máscaras, y la mezcla superpuesta de ambas.
Una vez cargadas, especificamos que son imágenes RGB, de esta manera nos aseguramos de que tienen 3 canales de entrada para su correcta lectura. Tras ello, las convertimos a un array de `NumPy` y las normalizamos al intervalo **[0,1]**, ya que en Redes

Neuronales es necesaria para obtener mejor eficiencia. Posteriormente, en las imágenes RGB (todas menos la máscara binaria), verificamos que la última dimensión, correspondiente al número de canales (o profundidad) sea igual a 3. La transformamos a un tensor de PyTorch, (tipo de datos estándar utilizado para operaciones y entrenamientos de modelos neuronales). Finalmente, reordenamos las dimensiones del tensor al estándar que espera PyTorch.

Realmente, estos pasos son comunes a la hora de ser procesadas por redes neuronales que utiliza PyTorch, ya que utiliza el orden de dimensiones siguiente (canales, alto, ancho) mientras que inicialmente, al leerlo con NumPy, tenemos (alto, ancho, canales).

A continuación, con el código 5.4 pasamos a `ImageDataset` 5.3 los conjuntos de entrenamiento y de prueba para que haga las transformaciones pertinentes y cargue las imágenes de ambos conjuntos.

```
1 train_dataset = ImageDataset(TRAIN_MASK_IMGS, TRAIN_MASK_ONLY,
    TRAIN_INPAINT_IMGS, test=False)
2 test_dataset  = ImageDataset(TEST_MASK_IMGS, TEST_MASK_ONLY,
    TEST_INPAINT_IMGS,  test=True)
```

Extracto de código 5.4: Aplicación de ImageDataset a conjuntos de entrenamiento y prueba

Una vez preprocesados y transformados los datos de una manera uniforme y estructurada, podríamos entrenar nuestro modelo con los objetos Dataset, pero tenemos un problema. En este momento estaríamos pasando absolutamente todos los tensores por memoria, lo que es muy poco eficiente, ya que es un número muy alto de datos a cargar en memoria, concretamente en la memoria RAM.

```
1 train_loader  = DataLoader(dataset=train_dataset, batch_size=10, shuffle=
    True, num_workers=1, pin_memory=True, drop_last=True)
2 test_loader  = DataLoader(dataset=test_dataset, batch_size=5, shuffle=
    False, num_workers=1, pin_memory=True, drop_last=False)
```

Extracto de código 5.5: Carga de los loaders para los datasets de entrenamiento y prueba

Para evitar lo anterior, se utiliza la clase `DataLoader` como en el código 5.5. Con ella, se divide el objeto `Dataset`, en varios lotes, que se cargarán en memoria de forma paralela. Especificaremos el objeto `Dataset` a dividir, el número de lotes con `batch_size`, el número de subprocesos en el que se podrán cargar los datos de forma paralela con `num_workers` y el mezclado de datos con el parámetro `shuffle`.

De esta manera, se cargan en pequeños lotes los datos, con los que entrenaremos iterativamente el modelo. A su vez nos aportará ciertos beneficios, como que los pesos se actualicen más frecuentemente. De manera que la convergencia sea más rápida durante el entrenamiento posterior y se aproveche el procesamiento en paralelo de una GPU.

6. Modelo

En este capítulo se introducen las diferentes redes neuronales que se han realizado, del tipo que son, sus esquemas y como han sido entrenados. Además, se explica más en profundidad la red final, con sus parámetros (consultar Anexo B para comprender cada término), y otros puntos importantes a destacar de la red. Finalmente, mostraremos como han sido entrenadas, ya que el entrenamiento ha seguido siempre la misma estructura y su posterior optimización.

6.1. Modelos de redes neuronales utilizados.

En esta sección podemos ver reflejado, mediante el uso de tablas, el distinto progreso logrado a lo largo del tiempo, desarrollando distintas redes desde cero junto con una ligera explicación de lo que incluyen. Cada tabla hace posible que este progreso se vea de una manera más visual.

6.1.1. Red 1

La red 6.1 es muy simple, solo tiene alguna capa convolucional y como activación la Tangente Hiperbólica al final. La primera capa extrae características de bajo nivel. En las capas de la 3 a la 5, extrae otras más complejas. Y finalmente, con la *Tanh*, se realiza una normalización de la salida entre -1 y 1.

Tabla 6.1

Red 1

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	896	896
ReLU-2	[1, 32, 720, 1280]	0	0
Conv2d-3	[1, 32, 720, 1280]	9,248	9,248
ReLU-4	[1, 32, 720, 1280]	0	0
Conv2d-5	[1, 32, 720, 1280]	867	867
Tanh-6	[1, 3, 720, 1280]	0	0
Total params: 11,011			
Trainable params: 11,011			
Non-trainable params: 0			

6.1.2. Red 2

A esta segunda red 6.2, se le añaden más capas convolucionales que a la primera. Por ello, captura características más complejas. Van aumentando de 3 a 64 canales (o filtros) para extraer más detalles y finalmente se hace el recorrido inverso antes de la activación de la Tangente Hiperbólica. Esto da algo más de profundidad que la red anterior.

Tabla 6.2

Red 2

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	2,368	2,368
ReLU-2	[1, 16, 720, 1280]	0	0
Conv2d-3	[1, 16, 720, 1280]	25,120	25,120
Conv2d-4	[1, 32, 720, 1280]	100,416	100,416
Conv2d-5	[1, 64, 720, 1280]	200,768	200,768
Conv2d-6	[1, 64, 720, 1280]	100,384	100,384
Conv2d-7	[1, 32, 720, 1280]	25,104	25,104
Conv2d-8	[1, 16, 720, 1280]	2,355	2,355
Tanh-9	[1, 3, 720, 1280]	0	0
Total params: 456,515			
Trainable params: 456,515			
Non-trainable params: 0			

6.1.3. Red 3

Esta tercera red 6.3 introduce la arquitectura de Encoder-Decoder, de manera que se aplican capas de *maxpooling* para reducir la resolución espacial de las imágenes, logrando eliminar información redundante a la hora de aumentar el número de canales. Asimismo, con la Convolución Transpuesta se aumentan de vuelta esa resolución para volver a lo original.

Tabla 6.3

Red 3

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	2,368	2,368
ReLU-2	[1, 16, 720, 1280]	0	0
MaxPool2d-3	[1, 16, 720, 1280]	0	0
Conv2d-4	[1, 16, 360, 640]	25,120	25,120
MaxPool2d-5	[1, 32, 360, 640]	0	0
Conv2d-6	[1, 32, 180, 320]	100,416	100,416
ConvTranspose2d-7	[1, 64, 180, 320]	16,448	16,448
Conv2d-8	[1, 64, 360, 640]	100,384	100,384
ConvTranspose2d-9	[1, 32, 360, 640]	4,128	4,128
Conv2d-10	[1, 32, 720, 1280]	25,104	25,104
Conv2d-11	[1, 16, 720, 1280]	2,355	2,355
Tanh-12	[1, 3, 720, 1280]	0	0
Total params: 276,323			
Trainable params: 276,323			
Non-trainable params: 0			

6.1.4. Red 4

En esta cuarta red 6.4, se agregan los bloques residuales, que permiten conexiones con residuos para obtener un mejor flujo del gradiente. De esta manera se busca no solucionar, sino reducir o mitigar la degradación de las redes muy profundas en el entrenamiento posterior.

Tabla 6.4

Red 4

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	2,368	2,368
ReLU-2	[1, 16, 720, 1280]	0	0
Conv2d-3	[1, 16, 720, 1280]	25,120	25,120
ResidualBlock-4	[1, 32, 720, 1280]	18,496	18,496
Conv2d-5	[1, 32, 720, 1280]	100,416	100,416
ResidualBlock-6	[1, 64, 720, 1280]	73,856	73,856
Conv2d-7	[1, 64, 720, 1280]	100,384	100,384
Conv2d-8	[1, 32, 720, 1280]	25,104	25,104
Conv2d-9	[1, 16, 720, 1280]	2,355	2,355
Tanh-10	[1, 3, 720, 1280]	0	0
Total params: 348,099			
Trainable params: 348,099			
Non-trainable params: 0			

6.1.5. Red 5

Con esta red 6.7, se aplica un nuevo enfoque, separando la red en dos partes, el Generador y el Discriminador, que en conjunto forman la denominada red GAN. El generador 6.5 trata de crear imágenes realistas a partir de ruido aleatorio. Tras ello, el discriminador 6.6, distingue entre imágenes reales y generadas por el Generador. Estas redes competirán entre sí durante el entrenamiento hasta que el Discriminador no sea capaz de diferenciar si la imagen es real o no.

Tabla 6.5

Generador

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 480, 640]	2,368	2,368
ReLU-2	[1, 16, 480, 640]	0	0
Conv2d-3	[1, 16, 480, 640]	25,120	25,120
Conv2d-4	[1, 32, 480, 640]	100,416	100,416
Conv2d-5	[1, 64, 480, 640]	200,768	200,768
Conv2d-6	[1, 64, 480, 640]	100,384	100,384
Conv2d-7	[1, 32, 480, 640]	25,104	25,104
Conv2d-8	[1, 16, 480, 640]	2,355	2,355
Sigmoid-9	[1, 3, 480, 640]	0	0
Total params: 456,515			
Trainable params: 456,515			
Non-trainable params: 0			

Tabla 6.6

Discriminador

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 480, 640]	1,792	1,792
MaxPool2d-2	[1, 64, 480, 640]	0	0
Conv2d-3	[1, 64, 240, 320]	73,856	73,856
MaxPool2d-4	[1, 128, 240, 320]	0	0
Conv2d-5	[1, 128, 120, 160]	147,584	147,584
MaxPool2d-6	[1, 128, 120, 160]	0	0
Conv2d-7	[1, 128, 60, 80]	73,792	73,792
MaxPool2d-8	[1, 64, 60, 80]	0	0
Conv2d-9	[1, 64, 30, 40]	18,464	18,464
MaxPool2d-10	[1, 32, 30, 40]	0	0
Linear-11	[1, 9600]	921,696	921,696
LeakyReLU-12	[1, 96]	0	0
Linear-13	[1, 96]	97	97
Sigmoid-14	[1, 1]	0	0
Total params: 1,237,281			
Trainable params: 1,237,281			
Non-trainable params: 0			

Tabla 6.7

Red 5 - GAN

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Generator-1	[1, 3, 480, 640]	456,515	456,515
Discriminator-2	[1, 3, 480, 640]	1,237,281	1,237,281
Total params: 1,693,796			
Trainable params: 1,693,796			
Non-trainable params: 0			

6.1.6. Red 6

De nuevo, volviendo a las estructuras anteriores, la red 6.8, extiende la red 6.4. Para ello, agrega más bloques residuales y capas convolucionales. Con ello se busca una mayor capacidad de extracción de características con detalle gracias a tener una mayor profundidad.

Tabla 6.8

Red 6

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	2,368	2,368
ReLU-2	[1, 16, 720, 1280]	0	0
Conv2d-3	[1, 16, 720, 1280]	25,120	25,120
ResidualBlock-4	[1, 32, 720, 1280]	18,496	18,496
Conv2d-5	[1, 32, 720, 1280]	100,416	100,416
ResidualBlock-6	[1, 64, 720, 1280]	73,856	73,856
Conv2d-7	[1, 64, 720, 1280]	401,536	401,536
ResidualBlock-8	[1, 128, 720, 1280]	295,168	295,168
Conv2d-9	[1, 128, 720, 1280]	401,472	401,472
Conv2d-10	[1, 64, 720, 1280]	100,384	100,384
Conv2d-11	[1, 32, 720, 1280]	25,104	25,104
Conv2d-12	[1, 16, 720, 1280]	2,355	2,355
Tanh-13	[1, 3, 720, 1280]	0	0
Total params: 1,446,275			
Trainable params: 1,446,275			
Non-trainable params: 0			

6.1.7. Red 7

En esta red 6.9, añadimos más capas de *pooling* y de convolución transpuesta para lograr tener un *upsampling* más eficiente. Sigue teniendo una estructura Encoder-Decoder.

Tabla 6.9

Red 7

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	1,792	1,792
ReLU-2	[1, 64, 720, 1280]	0	0
MaxPool2d-3	[1, 64, 720, 1280]	0	0
Conv2d-4	[1, 64, 360, 640]	73,856	73,856
ReLU-5	[1, 128, 360, 640]	0	0
MaxPool2d-6	[1, 128, 360, 640]	0	0
Conv2d-7	[1, 128, 180, 320]	295,168	295,168
ReLU-8	[1, 256, 180, 320]	0	0
MaxPool2d-9	[1, 256, 180, 320]	0	0
Conv2d-10	[1, 256, 90, 160]	1,180,160	1,180,160
ReLU-11	[1, 512, 90, 160]	0	0
MaxPool2d-12	[1, 512, 90, 160]	0	0
ConvTranspose2d-13	[1, 512, 45, 80]	524,544	524,544
ReLU-14	[1, 256, 90, 160]	0	0
ConvTranspose2d-15	[1, 256, 90, 160]	131,200	131,200
ReLU-16	[1, 128, 180, 320]	0	0
ConvTranspose2d-17	[1, 128, 180, 320]	32,832	32,832
ReLU-18	[1, 64, 360, 640]	0	0
ConvTranspose2d-19	[1, 64, 360, 640]	771	771
Sigmoid-20	[1, 3, 720, 1280]	0	0
Total params: 2,240,323			
Trainable params: 2,240,323			
Non-trainable params: 0			

6.1.8. Red 8

En esta red 6.10, implementamos por primera vez las redes UNet. Tenemos los bloques `EncoderBlock` y `DecoderBlock`, y conexiones residuales entre el encoder y el decoder. Se probó esta red por su efectividad conocida tanto para segmentación como para inpainting.

Tabla 6.10

Red 8

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
EncoderBlock-1	[1, 3, 480, 640]	2,832	2,832
EncoderBlock-2	[1, 16, 480, 640]	14,016	14,016
EncoderBlock-3	[1, 32, 480, 640]	55,680	55,680
EncoderBlock-4	[1, 64, 240, 320]	221,952	221,952
EncoderBlock-5	[1, 128, 120, 160]	886,272	886,272
EncoderBlock-6	[1, 256, 60, 80]	3,542,016	3,542,016
EncoderBlock-7	[1, 512, 30, 40]	4,721,664	4,721,664
Conv2d-8	[1, 512, 15, 20]	2,359,808	2,359,808
BatchNorm2d-9	[1, 512, 15, 20]	1,024	1,024
ReLU-10	[1, 512, 15, 20]	0	0
DecoderBlock-11	[1, 512, 15, 20], [1, 512, 15, 20]	9,441,792	9,441,792
DecoderBlock-12	[1, 512, 15, 20], [1, 512, 30, 40]	8,131,072	8,131,072
DecoderBlock-13	[1, 512, 30, 40], [1, 256, 60, 80]	2,296,064	2,296,064
DecoderBlock-14	[1, 256, 60, 80], [1, 128, 120, 160]	574,592	574,592
DecoderBlock-15	[1, 128, 120, 160], [1, 64, 240, 320]	143,936	143,936
DecoderBlock-16	[1, 64, 240, 320], [1, 32, 480, 640]	36,128	36,128
DecoderBlock-17	[1, 32, 480, 640], [1, 16, 480, 640]	11,664	11,664
DecoderBlock-18	[1, 16, 480, 640], [1, 3, 480, 640]	702	702
Conv2d-19	[1, 3, 480, 640]	12	12
Total params: 32,441,226			
Trainable params: 32,441,226			
Non-trainable params: 0			

6.1.9. Red 9

Esta red 6.11, es una versión simplificada y más sencilla de la anterior, la diferencia está en que se eliminan los bloques, pero se sigue utilizando la estructura básica de una UNet. De esta manera se logra tener una red mucho más ligera.

Tabla 6.11

Red 9

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	1,792	1,792
ReLU-2	[1, 64, 360, 640]	0	0
Conv2d-3	[1, 64, 360, 640]	73,856	73,856
ReLU-4	[1, 128, 180, 320]	0	0
Conv2d-5	[1, 128, 180, 320]	295,168	295,168
ReLU-6	[1, 256, 90, 160]	0	0
Conv2d-7	[1, 256, 90, 160]	1,180,160	1,180,160
ReLU-8	[1, 512, 45, 80]	0	0
ConvTranspose2d-9	[1, 512, 45, 80]	524,544	524,544
ReLU-10	[1, 256, 90, 160]	0	0
ConvTranspose2d-11	[1, 256, 90, 160]	131,200	131,200
ReLU-12	[1, 128, 180, 320]	0	0
ConvTranspose2d-13	[1, 128, 180, 320]	32,832	32,832
ReLU-14	[1, 64, 360, 640]	0	0
ConvTranspose2d-15	[1, 64, 360, 640]	771	771
Sigmoid-16	[1, 3, 720, 1280]	0	0
Total params: 2,240,323			
Trainable params: 2,240,323			
Non-trainable params: 0			

6.1.10. Red 10 - Final

En esta última red 6.12, se busca y se consigue encontrar la sencillez de la red anterior, pero con nuevas características que tratan de potenciar el desempeño en inpainting. Podremos ver en la siguiente sección una explicación más detallada de esta última red, ya que se ha tomado como red final.

Tabla 6.12

Red Final

Layer (type)	**Input Shape**	**Param #**	**Tr. Param #**
Conv2d-1	[1, 3, 720, 1280]	896	896
ReLU-2	[1, 32, 720, 1280]	0	0
MaxPool2d-3	[1, 32, 720, 1280]	0	0
Conv2d-4	[1, 32, 360, 640]	18,496	18,496
ReLU-5	[1, 64, 360, 640]	0	0
MaxPool2d-6	[1, 64, 360, 640]	0	0
Conv2d-7	[1, 64, 180, 320]	73,856	73,856
ReLU-8	[1, 128, 180, 320]	0	0
MaxPool2d-9	[1, 128, 180, 320]	0	0
Conv2d-10	[1, 128, 90, 160]	295,168	295,168
ReLU-11	[1, 256, 90, 160]	0	0
MaxPool2d-12	[1, 256, 90, 160]	0	0
Conv2d-13	[1, 256, 45, 80]	1,180,160	1,180,160
ReLU-14	[1, 512, 45, 80]	0	0
LSTM-15	[1, 3600, 512]	2,101,248	2,101,248
AttentionBlock-16	[1, 512, 45, 80]	5,771,264	5,771,264
ConvTranspose2d-17	[1, 1024, 45, 80]	1,048,832	1,048,832
ReLU-18	[1, 256, 90, 160]	0	0
ConvTranspose2d-19	[1, 512, 90, 160]	262,272	262,272
ReLU-20	[1, 128, 180, 320]	0	0
ConvTranspose2d-21	[1, 256, 180, 320]	65,600	65,600
ReLU-22	[1, 64, 360, 640]	0	0
ConvTranspose2d-23	[1, 128, 360, 640]	16,416	16,416
ReLU-24	[1, 32, 720, 1280]	0	0
ConvTranspose2d-25	[1, 64, 720, 1280]	195	195
Sigmoid-26	[1, 3, 720, 1280]	0	0
Total params: 10,834,403			
Trainable params: 10,834,403			
Non-trainable params: 0			

Cabe mencionar, que él se entenderá más adelante, en el capítulo de métricas y resultados (cap.7) por qué se ha seguido este recorrido, siempre tratando de obtener mejores resultados tras un entrenamiento similar para todas las redes.

6.2. Modelo final

El objetivo principal de este Trabajo de Fin de Grado es generar escenarios sintéticos a partir de imágenes ya existentes usando técnicas de inpainting mediante *Deep Learning*. Esto incluía los escenarios específicos de preparación y adaptación de los conjuntos de imágenes para el entrenamiento posterior de la red y también el desarrollo y entrenamiento de varios modelos hasta lograr encontrar uno que realmente aporte buenos resultados.

En esta sección se aborda esto último, ya que en la anterior se ha explicado la estructura y evolución que se ha logrado entre los distintos modelos.

A continuación, se presenta más en detalle el modelo final y su entrenamiento. Toda esta sección es el antecedente del objetivo específico de evaluación y optimización del rendimiento del modelo.

Vamos a adentrarnos en la arquitectura de la red final ya presentada en la tabla 6.12 de la sección anterior.

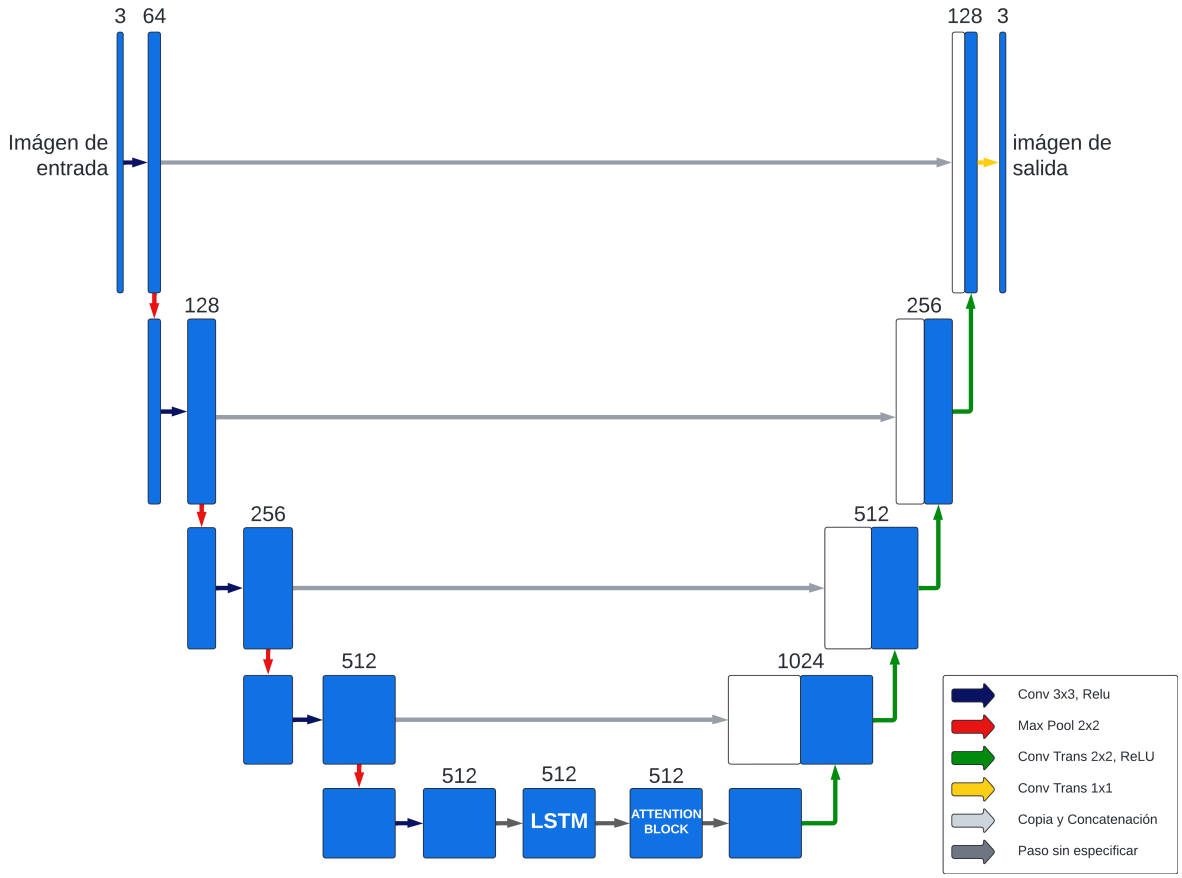


Figura 6.1: Estructura de mi red UNet

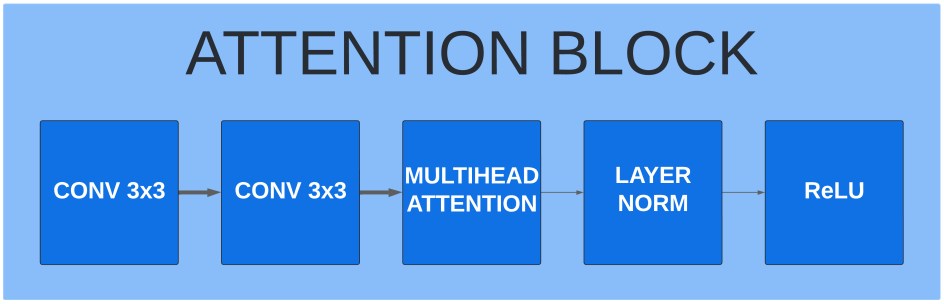


Figura 6.2: Bloque de Atención Multicabeza

6.2.1. Estructura de la UNet

La figura 6.1 muestra de forma clara la arquitectura en forma de **U** que tiene nuestra UNet. En ella distinguimos tres partes: la primera a la izquierda, es la fase de *downsampling* (encoder). La parte inferior situada en la zona intermedia se conoce como *bottelneck*. Finalmente, en el lado derecho tenemos la fase de *upsampling* (decoder).

Encoder - *Downsampling*

En esta primera fase, tenemos una secuencia de capas convolucionales (`nn.Conv2d`) que continúan con operaciones de submuestreo (`nn.MaxPool2d`). Como ya mencionamos en la sección correspondiente a inpainting del Marco Teórico 3.4, la función de esta parte de la red es reducir la resolución espacial. Gracias al aumento de la profundidad de los canales (y número de características). Tras repetir este proceso varias veces, se capturan las distintas características de alto nivel y semánticas más abstractas.

Fase central - *Bottleneck*

Esta fase central está compuesta por una capa LSTM y un bloque de atención multicabeza. En toda esta fase, las características de la fase del encoder se procesan secuencialmente y se les aplica la atención multicabeza. Para ello, primero se toman las características y se procesan a través de la capa LSTM (`nn.LSTM`), su aplicación está pensada a los datasets que son una secuencia de imágenes divididas en varios frames, como en nuestro caso con *BDD100K*. Con esto, la red aprende a capturar secuencias temporales y contextualidad. Posteriormente, la salida se reorganiza para adaptarla y ser procesada por el bloque de atención (ver figura 6.2). En este bloque, se aplica la atención multicabeza, lo que aporta grandes beneficios como pueden ser:

- **Captura de relaciones a larga distancia**. Este tipo de atención logra que la red establezca conexiones entre regiones muy distantes de la imagen. De esta forma, logra comprender el contexto global de la imagen, y puede predecir de forma precisa el espacio a rellenar.
- **Enfoque adaptativo**. En lugar de procesar la imagen de una manera uniforme, se enfoca en distintas regiones o características que considera más relevantes.

- **Procesamiento en paralelo**. Al dividir el espacio de características en varios subespacios, se pueden encontrar y capturar distintas características de manera paralela.

Todo esto en conjunto logra que la red pueda relacionar características a largas distancias entre si dentro de la imagen, con el objetivo de recrear más fácilmente las zonas enmascaradas.

Decoder - *Upsampling*

Esta es la fase final y es simétrica a la fase del *downsampling*. La diferencia es que realiza un sobre-muestreo aplicando la convolución transpuesta (`nn.ConvTranpose2d`). En cada paso, se concatenan las características con las correspondientes a su fase simétrica del *downsampling* (a esto se le conoce como conexiones residuales o *skip connections*). De esta manera, la red combina información de bajo nivel con la de alto nivel durante la reconstrucción. Todo esto se repite varias veces de forma gradual hasta que la salida tiene la misma resolución que la entrada original.

6.2.2. Implementación del código

Para poder aplicar y realizar todo lo mencionado anteriormente se puede consultar el código siguiente. Pero hemos de entender como funcionan primero. En PyTorch `nn.Module` es la clase principal que proporciona toda la funcionalidad básica que permite la creación de redes neuronales. Al definir una nueva clase de red neuronal, se ha de heredar de `nn.Module` y adaptar los métodos heredados como `__init__` y `forward`, que son el constructor donde se inicializan todas las capas de la red junto con los parámetros y el método de propagación hacia delante, en el que se reciben los datos de entrada durante el entrenamiento y que devuelve la salida de la red.

En la porción de código 6.1 vemos la implementación del bloque de Atención Multicabeza, en su método `forward` podemos ver que después de aplicar las convoluciones hay que adaptar los datos para poder realizar la atención correctamente y después devolverla a su estado original.

```
1  class AttentionBlock(nn.Module):
2      def __init__(self, in_channels, out_channels, kernel_size=3, padding
       =1):
3          super(AttentionBlock, self).__init__()
4          self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=
       kernel_size, padding=padding)
5          self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=
       kernel_size, padding=padding)
6          self.attn = nn.MultiheadAttention(out_channels, num_heads=8,
       batch_first=True)
7          self.norm = nn.LayerNorm(out_channels)
8          self.activation = nn.ReLU()
9
10     def forward(self, x):
11         x = self.conv1(x)
12         x = self.activation(x)
```

```
        x = self.conv2(x)
        b, c, h, w = x.size()
        x = x.view(b, c, h * w).permute(2, 0, 1)  # Reshape and permute
        attn_output, _ = self.attn(x, x, x)
        x = attn_output.permute(1, 2, 0).view(b, c, h, w)  # Revert the
    permute and reshape
        x = x.view(b, c, -1)  # Flatten the last two dimensions
        x = self.norm(x.reshape(b, -1, c))  # Reshape for LayerNorm and
    apply normalization
        x = x.view(b, c, h, w)  # Reshape back to original
        return x
```

Extracto de código 6.1: Clase AttentionBlock

En la porción siguiente (ver código 6.2) vemos la implementación de la clase principal de la red. Si nos fijamos, el encoder y el decoder se han implementado en un bloque de código secuencial en lugar de dividirlos en varios bloques. Se ha optado por hacerlo de esta forma para simplificar la definición y el flujo de capas del codificador y decodificador, de manera que el código sea mucho más fácil y claro de comprender. Realmente, si el código a implementar hubiera sido mucho más complejo, se habría debido definir los bloques del encoder y decoder por separado para tener un mejor control de los datos.

```
class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(),
        )

        self.lstm = nn.LSTM(512, 512, batch_first=True)
        self.attn_block = AttentionBlock(512, 512)

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(1024, 256, kernel_size=2, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(512, 128, kernel_size=2, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(256, 64, kernel_size=2, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 32, kernel_size=2, stride=2),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, kernel_size=1),
            nn.Sigmoid(),
```

```
        )

    def forward(self, x):
        skip_connections = []

        for layer in self.encoder:
            x = layer(x)
            skip_connections.append(x)
            if isinstance(layer, nn.MaxPool2d):
                skip_connections.pop()

        batch_size, channels, height, width = x.size()
        x = x.view(batch_size, -1, channels)
        x, _ = self.lstm(x)
        x = x.unsqueeze(1)
        x = x.permute(0, 2, 3, 1)
        x = x.reshape(batch_size, channels, height, width)

        x = self.attn_block(x)

        skip_connections = skip_connections[::-1]

        for i, layer in enumerate(self.decoder):
            if isinstance(layer, nn.ConvTranspose2d):
                x = layer(torch.cat((x, skip_connections[i]), dim=1))
            else:
                x = layer(x)

        return x
```

Extracto de código 6.2: Clase UNet

Para poder cargar correctamente simplemente será necesario llamar a la clase del modelo y pasarlo a la GPU, en este caso concreto con `model = UNet().to(device)`. Aparte de la carga del modelo, antes del entrenamiento se ha de definir un optimizador al que se le aplicarán regularizaciones.

6.2.3. Optimizador y regularizaciones

El optimizador sirve para poder actualizar los pesos y los sesgos (*bias*) del modelo de manera que sea lo más eficiente posible, su objetivo tiene minimizar la función de pérdida todo lo posible para mejorar el rendimiento del modelo. En nuestra red hemos utilizado el optimizador Adam.

Adam es uno de los optimizadores más utilizados en el *Deep Learning*. Fue introducido en 2014 por Diederik P. Kingma y Jimmy Ba[97]. Este optimizador combina dos propiedades del descenso del gradiente: el momento y el promedio del gradiente cuadrático (*momentum* y *RMSProp*). Su funcionamiento es el siguiente:

1. **Momento**: Adam calcula el promedio exponencial de los gradientes pasados. Con este método se acelera el proceso de convergencia y se suavizan los pesos en cada actualización de los mismos.

2. **Promedio del gradiente cuadrático**: se calcula un promedio exponencial de los cuadrados de los gradientes pasados. Con esto se logra adaptar la curvatura de la superficie del error y se ajusta la tasa de aprendizaje para cada parámetro.

3. **Corrección de sesgos**: se corrige el sesgo inicial en tanto en la estimación del momento como la del promedio del gradiente cuadrático, de manera que sea más estable.

Junto con Adam, se utilizan ciertas regularizaciones que sirven para prevenir el sobreajuste todo lo posible. Las regularizaciones utilizadas a lo largo de este Trabajo de Fin de Grado (pese a que finalmente haya alguna que no se haya aplicado) se explican a continuación:

1. **Regularizaciones L1 y L2**: son técnicas que agregan una penalización a la pérdida para que los pesos del modelo sean lo más pequeño posible y más simples, logrando prevenir el sobreajuste. Para más información sobre su funcionamiento se pueden consultar los siguientes papers [98] y [99]

2. **Dropout**. Esta técnica [100] [101], desactiva neuronas de manera aleatoria durante el entrenamiento, de manera que el modelo tenga que aprender a representar de manera más robusta y distribuida.

En pocas palabras, el optimizador Adam se ha utilizado gracias a su rápida convergencia, requerimiento de bajo ajuste de hiperparámetros y el manejo de gradientes dispersos que, junto a las regularizaciones L1 y L2, puede lograr que el modelo obtenido sea lo más preciso y generalizable posible. Para ello, en una primera versión, lo hemos cargado solo con regularización L1, poniendo la tasa de aprendizaje en 0,0001 con la siguiente línea: `optimizer = Adam(model.parameters(),lr=0.0001)`. Posteriormente, en la siguiente sección entraremos más en detalle de como mejorar este entrenamiento gracias al ajuste del optimizador.

6.3. Entrenamiento de la red

Este código de entrenamiento (ver código 6.3), se ha utilizado para absolutamente todos y cada uno de los modelos especificados anteriormente para el inpainting, en él utilizamos la biblioteca ***Weights and Biases*** (wandb) para el seguimiento y la visualización del proceso de entrenamiento.

```
1
2  # PARTE 1
3  wandb.init(
4      # Inicializar el proyecto de wandb en el que va a estar este
       entrenamiento
5      project="EL NOMBRE DE TU PROYECTO",
6      name="EL NOMBRE DE TU EJECUCION",
7  )
8
9  metrics= defaultdict(list)
10 EPOCHS = 1100
11 init_epoch=0#+last_epoch+1
12 lossfn = torch.nn.MSELoss()
```

```
13
14
15 # Definir la perdida adversarial
16 criterion_adv = nn.L1Loss()
17
18 # PARTE 2
19 for epoch in range(init_epoch,EPOCHS):
20     start_time = time.time()
21     train_loss = [] ; train_psnr = []
22     train_og_loss, train_og_psnr = [],[]
23     model.train()
24
25 # PARTE 3
26     for mask_batch, binary_mask, inpainted_batch in train_loader:
27         # Mover los datos a la GPU
28         mask_batch = mask_batch.to(device)
29         inpainted_batch = inpainted_batch.to(device)
30         binary_mask = binary_mask.to(device)
31
32 # PARTE 4
33
34         optimizer.zero_grad()
35
36
37         # Alimentar los datos al modelo
38         painted_batch = model(mask_batch)
39
40
41
42         # La funcion de perdida es la MSE
43         error = (inpainted_batch - painted_batch)**2
44         masked_error = binary_mask * error  # Aplicar la mascara binaria
    al error
45         unmasked_error = (1 - binary_mask) * error  # Aplicar la inversa
    de la mascara binaria al error
46         loss_mse = 10*torch.mean(masked_error) + torch.mean(
    unmasked_error)  # Calcular la perdida usando ambos errores
47
48         # Calcular la perdida adversarial
49         loss_adv = criterion_adv(painted_batch, inpainted_batch)
50         # La perdida total es una combinacion de la perdida MSE y la
    perdida adversarial
51         loss = loss_mse + loss_adv
52
53         # PSNR (dB) metric
54         psnr = 20 * torch.log10(1. / torch.sqrt(loss))
55 # PARTE 5
56         loss.backward()
57         optimizer.step()
58
59 # PARTE 6
60         train_loss.append(loss.item())
61         train_psnr.append(psnr.item())
62
63
64
65
```

```
    # Media de las metricas tras cada epoca (media de todos los batches)
    metrics['train_loss'].append(np.mean(train_loss))
    metrics['train_psnr'].append(np.mean(train_psnr))
    metrics['train_loss_og'].append(np.mean(train_og_loss))
    metrics['train_psnr_og'].append(np.mean(train_og_psnr))

# PARTE 7

    #############
    ### Plots ###

    print(f"Epoch {epoch + 1} of {EPOCHS} took {time.time() - start_time
    :.3f}s\t Loss:{np.mean(train_loss)}\t PSNR:{np.mean(train_psnr)}\n")

    wandb.log({"Epoch": epoch + 1, "Loss": np.mean(train_loss), "PSNR":
    np.mean(train_psnr)})

    wandb_images = [mask_batch[0], painted_batch[0], inpainted_batch[0]]
    log_images(wandb_images, epoch)

# PARTE 8
    ### Guardado del diccionario con el modelo epoca a epoca.
    torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            }, "EL NOMBRE QUE QUIERAS DE TU MODELO.pt")

# PARTE 9
wandb.finish()
```

Extracto de código 6.3: Entrenamiento del modelo

Veamos cada parte del código por porciones.

1. Inicialización de wandb y configuración de épocas.
   - Se inicia wandb con los parámetros del proyecto y el nombre de la ejecución.
   - Definimos un diccionario `metrics` en el que se van a almacenar las métricas de entrenamiento.
   - Se establece un número total de épocas.
   - Definimos la función de pérdida adversarial L1 (`criterion_adv`).
2. Ciclo de entrenamiento.
   - Se itera sobre un rango de épocas.
   - Se inicializan varias listas para almacenar las pérdidas y PSNR en entrenamiento.
   - Se configura el modelo en modo entrenamiento (`model.train()`).
3. Iteración sobre el lote de datos de entrenamiento.
   - Se itera sobre los lotes de datos de la imagen enmascarada, la máscara binaria y la imagen objetivo, la "inpaiteada". (`mask_batch, binary_mask,`

`inpainted_batch`) cargadas mediante el `train_loader` definido anteriormente en la carga de datos.

- Movemos los datos a la GPU utilizando `to(device)`.

4. Cálculo de métricas y propagación hacia adelante.
   - Con `optimizer.zero_grad()` se restablecen los gradientes del optimizador.
   - Se calcula el error entre la imagen inpaiteada y la imagen pintada (que es la generada por el modelo).
   - Se aplica la máscara binaria al error anterior para poder separar en dos partes ese error, en la parte de la máscara y en la no enmascarada.
   - Se calcula la perdida del error cuadrático medio (MSE) pero modificada a cada uno de los dos errores anteriores, es decir, al enmascarado y no enmascarado. La modificación realizada ha sido el multiplicar por diez a la zona enmascarada. Esto ha sido para tratar de darle mayor peso a la zona a recrear con nuestro modelo. (`loss_mse`)
   - También se calcula la pérdida adversarial (`loss_adv`) con la función de perdida L1.
   - Calculamos la perdida total combinando la pérdida MSE modificada y adversarial (`loss`).
   - Se calcula el PSNR (`psnr`) sobre la pérdida total anterior [1].
5. Retropropagación y optimización:
   - Con `loss.blackward()` se realiza la retropropagación de la pérdida.
   - Se actualizan los pesos usando el optimizador (`optimizer.step()`).
6. Registro de métricas.
   - Se añaden las métricas anteriores al final de sus respectivas listas.
   - Se calculan la media de las métricas en la época y se agregan al diccionario `metrics`.
7. Carga y visualización de datos (en local y wandb).
   - Se muestra un mensaje en cada época como salida con las métricas obtenidas en la época actual.
   - Mediante `wandb.log()` se cargan las métricas en wandb.
   - Se registran imágenes de ejemplo en wandb gracias a la función `log_images()` 6.4.

[1] Este PSNR no es real, ya que esta métrica se ha de aplicar directamente sobre el error MSE original. Simplemente, se va a utilizar este PSNR modificado como métrica de referencia en el entrenamiento, el PSNR real, será más alto, pero hasta la validación y toma de resultados, nos guiaremos por esta métrica modificada tomada en el entrenamiento.

8. Guardado del modelo: se guarda en un diccionario el estado del modelo y el optimizador al final de cada época.

9. Finalización de wandb. Se finaliza la ejecución de wandb mediante `wandb.finish()`.

```
from torchvision.utils import make_grid
def log_images(images, epoch):
    '''
    Funcion que envia el log con las imagenes a wandb
    '''
    length = len(images)
    images = make_grid(images, length)

    images = wandb.Image(images, caption = "1º: Merged Mask, 2º: Output, 3º: Painted Image")
    wandb.log({"examples": images, 'epoch':epoch})

    return print('Logged correctly')
```

Extracto de código 6.4: Función para hacer log de las imágenes en wandb

El optimizador utilizado durante la retropropagación del paso 5, tiene un papel muy importante, y el cambio en sus regularizaciones puede hacer que los resultados mejoren bastante. En la sección 6.3.2 se explica en detalle.

6.3.1. Visualización de resultados en *wandb*

Wandb tiene una interfaz gráfica en la que gracias a un *dashboard* podemos ver y comparar todas las ejecuciones que hayamos hecho y así compararlas. Aparte, tiene un *dashboard* separado para cada ejecución por separado, donde se pueden ver más en detalle todos los datos.

Como se comentó anteriormente, a wandb subimos una ejecución mediante el uso de *logs*. Lo que se pasa en cada *log*, es lo que sale en wandb. Durante el entrenamiento, generalmente, hemos pasado la época actual, nuestra pérdida y nuestro PSNR.

Como esquema general del *dashboard* con todas las ejecuciones podemos ver el *dashboard* de la figura 6.3.

En el *dashboard*, podemos ver a la izquierda, las distintas ejecuciones de entrenamiento que se han realizado, llegando a tener más de 200 pruebas de entrenamiento y muestra de resultados. En la zona superior podemos ver y seleccionar una a una las ejecuciones con las imágenes de entrenamiento enviadas a wandb mediante la función `log_images()` y en la zona inferior los gráficos que *plotean* el PSNR, la pérdida y el número de épocas. Gracias a estos gráficos podemos ver cuál ha sido el mejor entrenamiento.

Asimismo, podemos entrar a cada uno de los gráficos y mostrar la comparación con el número de ejecuciones que se desee. Si ampliamos la gráfica del PSNR, podemos verlo claramente (fig. 6.4).

Figura 6.3: Dashboard página general Wandb

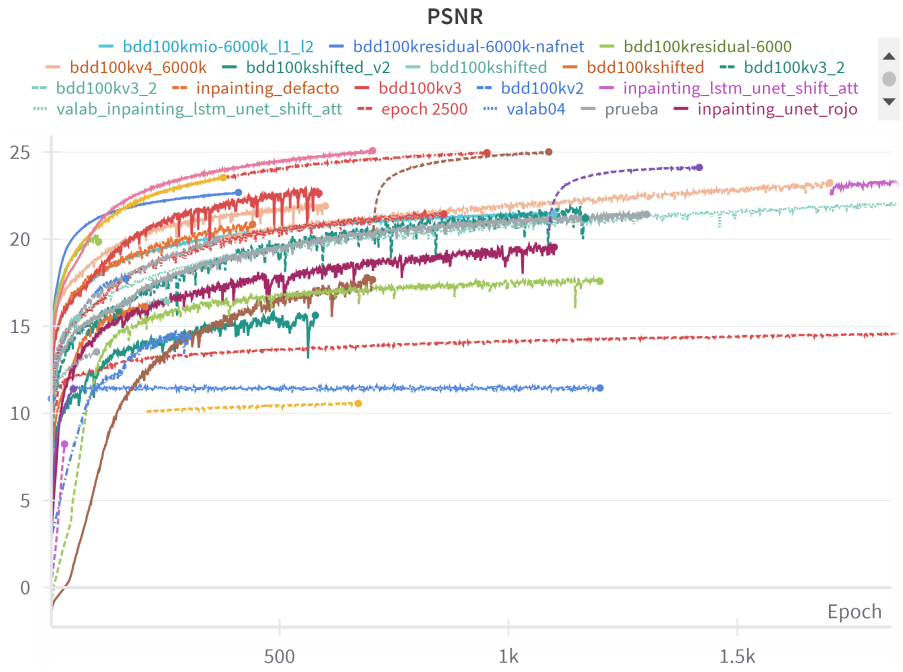


Figura 6.4: Comparativa de PSNR para distintas ejecuciones.

Si elegimos el mejor entrenamiento con mejores resultados, vemos que en general, de todas las ejecuciones totales realizadas, el dataset *BDD100K* es el que da mejores resultados. Podemos ver esa ejecución en concreto en el *dashboard* mostrado en la figura 6.5.

Si nos fijamos en la figura 6.6, podemos ver a la izquierda la imagen con la máscara, en el centro la recreada por nuestro modelo y a la derecha el *Ground Truth* al que tiene que llegar el modelo. Gracias a esto, junto con las métricas podemos ver que esta ejecución ha sido el mejor entrenamiento de toda la red para todos los conjuntos de datos utilizados y en específico, ***BDD100K***.

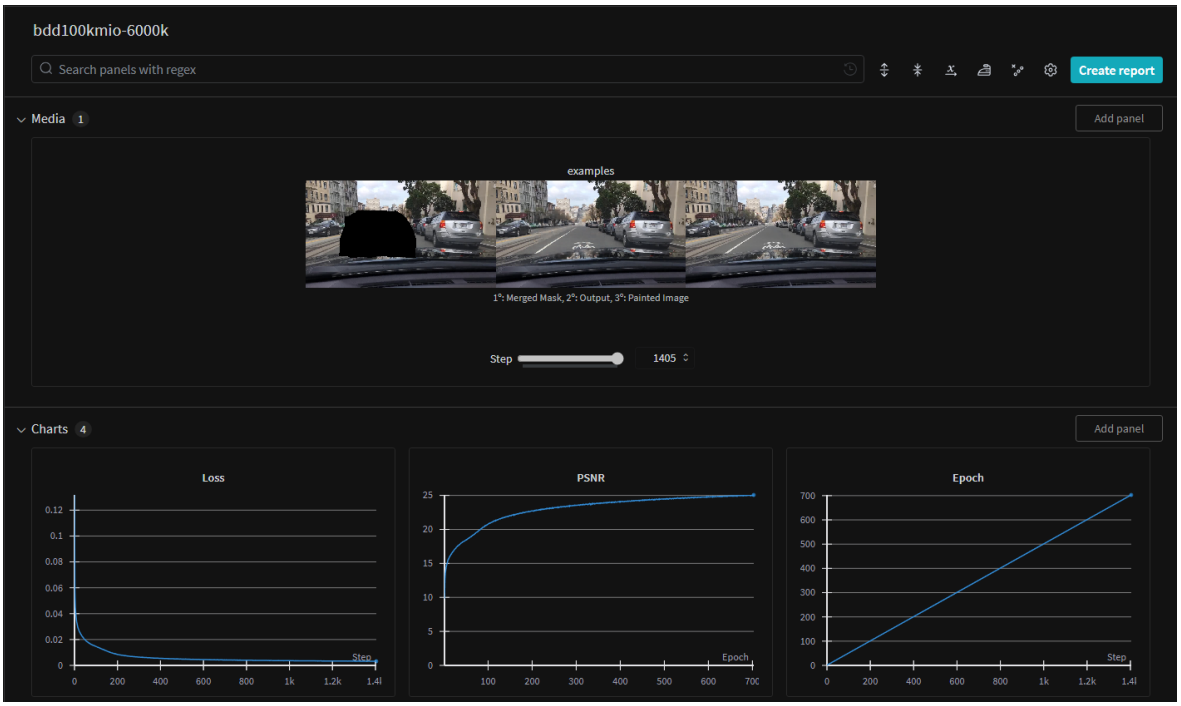


Figura 6.5: Dashboard de la mejor ejecución.

Figura 6.6: Visualización de la última época de entrenamiento de BDD100K.

Pese a ello, si en wandb filtramos para que nos muestre los resultados para los datasets *DEFACTO* (ver figuras 6.7 y 6.8) y *VKITTI2* (ver figuras 6.9 y 6.10), podemos ver que de nuevo, esta red, da muy buenos resultados en torno a 25 de nuestro PSNR, que traducido a PSNR Real, es aproximadamente 30. Se entenderá la diferencia entre el PSNR Real y el nuestro en el capítulo siguiente.

Figura 6.7: Visualización de la última época de entrenamiento en DEFACTO

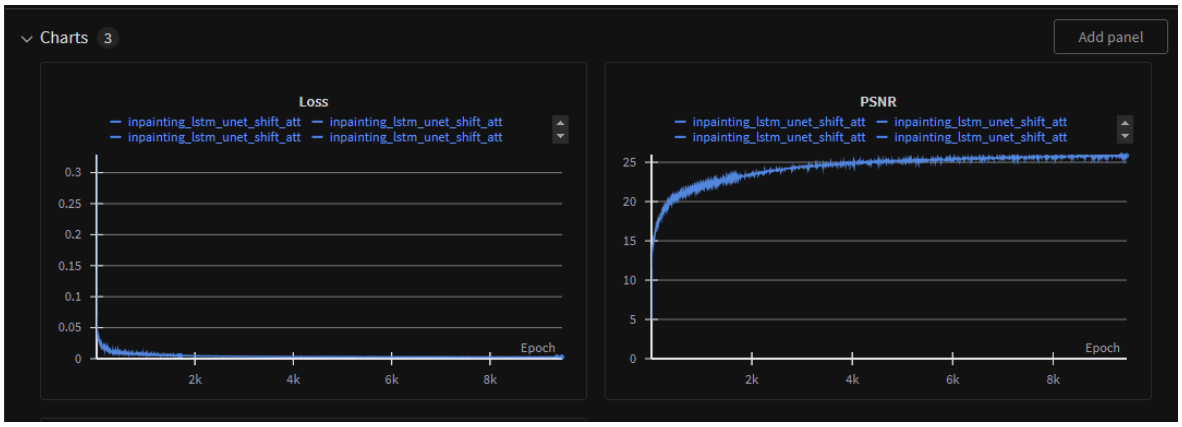


Figura 6.8: Métricas de DEFACTO durante el entrenamiento.

Figura 6.9: Visualización de la última época de entrenamiento en Virtual Kitti V2.

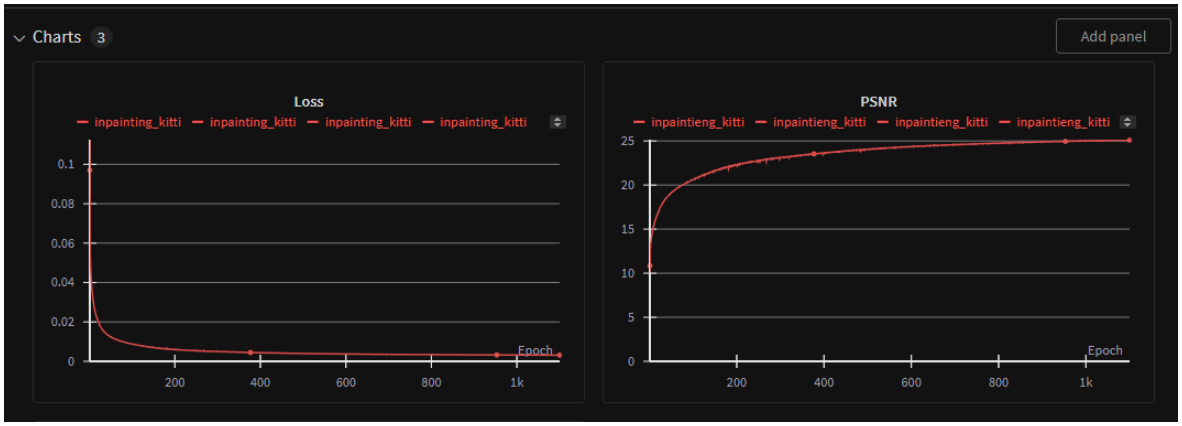


Figura 6.10: Métricas de Virtual Kitti V2 durante el entrenamiento.

6.3.2. Optimización del entrenamiento

A lo largo de este Trabajo de Fin de Grado, se ha optado por seguir varias técnicas de optimización del entrenamiento.

Se han buscado distintos enfoques como el uso de distintos valores en los hiperparámetros de *regularizaciones L1/L2*, el uso de *dropout* y uso de ciertos *schedulers* que sirven para modificar la tasa de aprendizaje durante el entrenamiento:

- **Consinne Annealing** [102] sirve para disminuir la tasa de aprendizaje en función de la curva del coseno.
- **ReduceLROnPlateau** [103] reduce la tasa de aprendizaje cuando la pérdida deja de mejorar tras ciertas épocas de aprendizaje.
- **CyclicLR** [104] ajusta la tasa de aprendizaje cíclicamente entre dos límites.

Pese a todos los intentos de encontrar redes con mejores resultados mediante el uso de estas técnicas de ajuste de la tasa de aprendizaje, para nuestro modelo concreto durante su entrenamiento, no se obtuvieron mejores resultados que con el ajuste inicial de tasa de aprendizaje en 0,001. Por ello, se buscó otro enfoque de búsqueda de hiperparámetros, para ello, hemos utilizado una técnica de búsqueda de hiperparámetros mediante la biblioteca ***Optuna*** [105], que es una biblioteca de optimización de hiperparámetros que utiliza algoritmos de optimización bayesiana que ayuda a encontrar los mejores valores para un modelo concreto.

Para poder utilizarlo, hemos creado una función en la que hemos pasado el bucle de entrenamiento con pocas épocas, se han elegido los hiperparámetros del optimizador (tasa de aprendizaje y peso del decaimiento) de manera automática, de la manera que *Optuna* cree conveniente.

Para esta modificación automática de hiperparámetros, le hemos tenido que decir que el objetivo es minimizar la pérdida todo lo posible en un número de intentos `n_trials` determinadas. A lo largo de esas pruebas, *Optuna* entrenará el modelo con esas distintas combinaciones de hiperparámetros que irán variando en función de los resultados de pruebas anteriores y algoritmos de optimización bayesiana. Una vez completadas todas las pruebas, *Optuna* nos devuelve los hiperparámetros con los que se han obtenido mejores resultados y los muestra.

Con este último método se han logrado obtener mejores resultados que con el uso de otras técnicas mencionadas anteriormente, por lo que se ha optado por mantener el entrenamiento de la red con estos parámetros (consultar código 6.5 para ver su implementación[2]).

```
# Funcion de entrenamiento para distintas pruebas distintas
def train_model(trial):
    # Iniciar hiperparametros
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-3)
    weight_decay = trial.suggest_loguniform("weight_decay", 1e-5, 1e-3)

    # Iniciar el modelo con los hiperparametros     model = UNet().to(
    device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=
    weight_decay)
    EPOCHS=100

    # Bucle de entrenamiento
    for epoch in range(100):

        ################################
        RESTO DE CODIGO DE ENTRENAMIENTO
        ################################

        print(f"Epoch {epoch} of {EPOCHS} took {time.time() - start_time
    :.3f}s\t Loss:{np.mean(train_loss)}")

    # Retorno de la perdida en esta ejecucion
    return np.mean(train_loss)

```

[2]Documentación de Optuna.

```
# Funcion para definir el objetivo y buscar la perdida
def objective(trial):
    # Inicializar Wandb
    wandb.init(entity="cidautai", project="inpainting", name="
    hyperparameter_search")

    # Entrenamiento del modlo y obtencion de la perdida
    loss = train_model(trial)

    # Finalizar la ejecucion de wandb
    wandb.finish()

    # Return the loss for optimization
    return loss

# Definir el espacio de busqueda y el inicio de la optimizacion de
    parametros
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=15)  # Adjust n_trials as needed

# Quedarnos con los mejores hiperparametros
best_params = study.best_params
print("Best hyperparameters:", best_params)
```

Extracto de código 6.5: Funciones de mejora de optimizador

Gracias al código anterior, hemos podido ver que para el dataset *BDD100K* se han obtenido los siguientes resultados de pérdida.

`Best hyperparameters: 'lr': 22.69011603e-05, 'weight_decay' : 2.007011005e-05)` (ver gráfico 6.11).

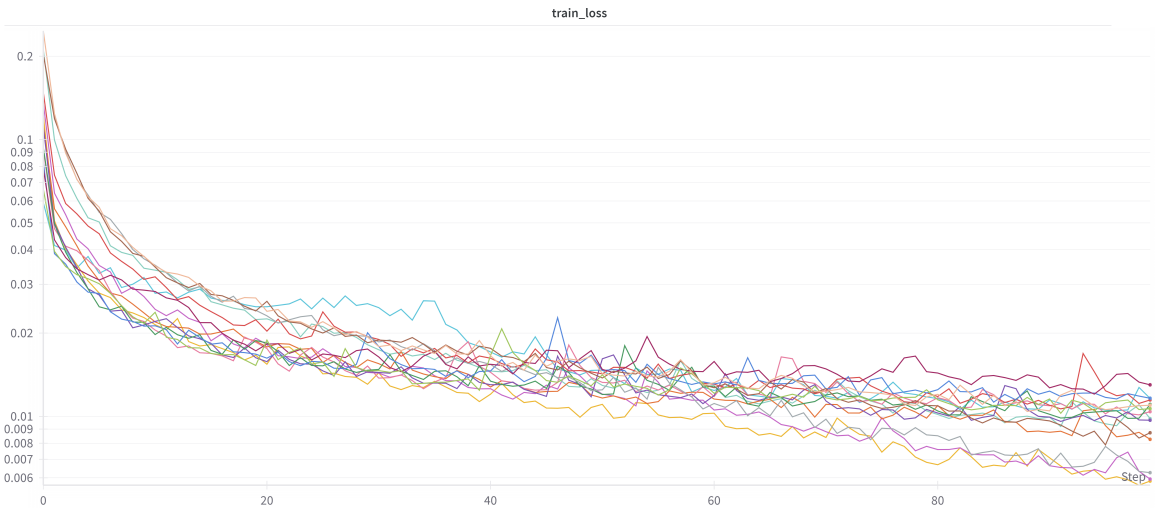


Figura 6.11: Gráficas con evolución de la pérdida con distintas tasas de aprendizaje y pesos del decaimiento. (La escala del eje Y es logarítmica para ver mejor los resultados).

Con WandB, hemos podido ver como es de cierto que, realmente, mejora con unos métodos u otros. En esta prueba, con un lote de 250 imágenes en la que se ha cambiado

el *scheduler* del optimizador, y el tamaño del *batch* de entrenamiento, hemos visto que el que mejor funciona y entrega mejores resultados es Optuna (ver figura 6.12)

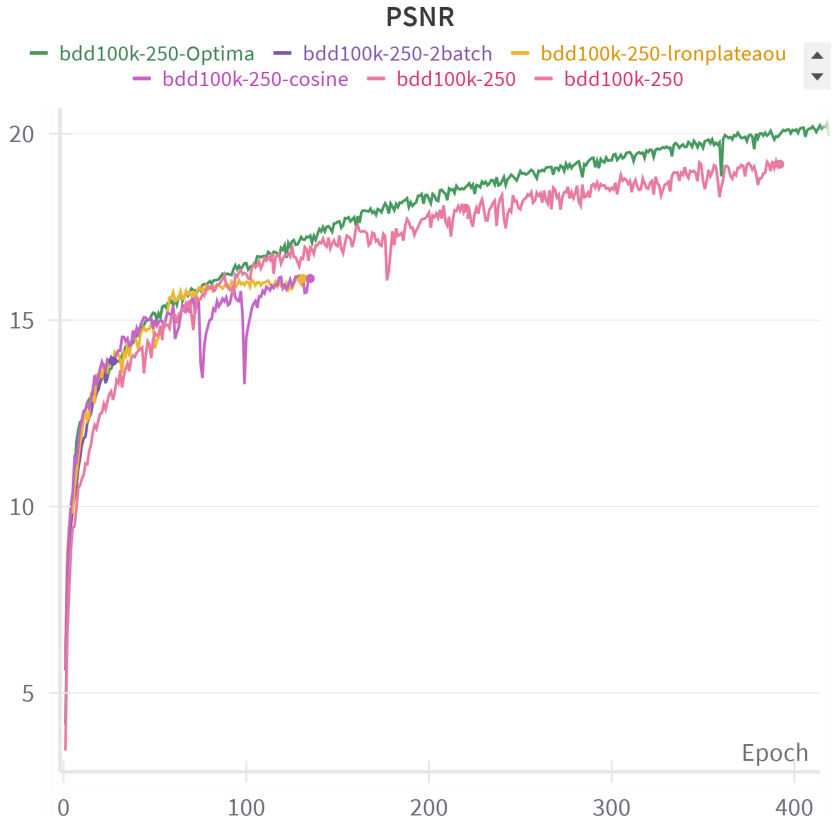


Figura 6.12: Comparativa de PSNR para la optimización del entrenamiento

7. Resultados

En este capítulo, se mostrará como de buenos han sido nuestros resultados basados en métricas, y gracias a ellas, se verá la evolución que se ha ido logrando a través de los distintos modelos presentados en el capítulo anterior. De la misma forma, veremos los resultados del modelo final para los distintos datasets utilizados, también se mostrarán los resultados en un mismo modelo pero con distinto número de imágenes durante el entrenamiento. Pudiendo así completar lo que queda de nuestros objetivos, que es corroborar y demostrar que ha habido un progreso, y que los resultados son buenos para imágenes de conducción autónoma. Para poder probarl todo lo anterior, se ha optado por utilizar métricas como el MSE, PSNR, SSIM MSSIM y LPIPS que son las medidas base para este tipo de proyectos (se recomienda leer el Anexo A).

7.1. Búsqueda de la mejor red

En esta sección, se muestran los resultados obtenidos entre las distintas redes introducidas en el capítulo 6. Estas comparativas han servido para elegir la mejor red entre todas las creadas.

Para hacerlo de una manera lo más representativa posible (pese a en su momento haber tomado estos resultados red a red), se ha creado una función muy similar a la del entrenamiento, en ella se ha pasado una lista con todos los modelos preentrenados ya cargados. Posteriormente, toma las imágenes del conjunto de validación cargado en el *train_loader* como se explicó en el capítulo 5. Para evaluar todos los modelos y encontrar el mejor para el inpainting, se utiliza como base el dataset *DEFACTO*.

El código sería el siguiente.

```
from torchvision.utils import make_grid
import cv2
import numpy as np
import skimage.metrics
import lpips
import torch
import pytorch_msssim
import matplotlib.pyplot as plt

# Inicializar el modelo LPIPS
loss_fn = lpips.LPIPS(net='alex')

# Funcion para evaluar todos los modelos en todas las imagenes
def evaluar_todos(modelos):
    for i, modelo in enumerate(modelos, start=1):
        print(f'Evaluando el modelo {i}')
        modelo.eval()
        losses = []
        psnrs = []
        ssims = []
```

```
        ms_ssims = []
        lpips_values = []
        imagenes_impresas = 0
        for mask_batch, _, inpainted_batch in test_loader:

            mask_batch = mask_batch.to(device)
            inpainted_batch = inpainted_batch.to(device)


            # Alimentar los datos al modelo
            with torch.no_grad():
                painted_batch = modelo(mask_batch)


            loss = torch.mean((inpainted_batch - painted_batch)**2)
            psnr = 20 * torch.log10(1. / torch.sqrt(loss))


            # Calcular el SSIM
            ssim = pytorch_msssim.ssim(inpainted_batch, painted_batch,
    data_range=1.0)

            # Calcular el MS-SSIM
            ms_ssim = pytorch_msssim.ms_ssim(inpainted_batch,
    painted_batch, data_range=1.0)

            # Calcular el LPIPS
            lpips = loss_fn.forward(inpainted_batch.cpu(), painted_batch.
    cpu())

            losses.append(loss.item())
            psnrs.append(psnr.item())
            ssims.append(ssim)
            ms_ssims.append(ms_ssim)
            lpips_values.append(lpips.mean().item())
        # Calcular las medias de las metricas
        loss_promedio = sum(losses) / len(losses)
        psnr_promedio = np.mean(psnrs)
        ssim_promedio = sum(ssims) / len(ssims)
        ms_ssim_promedio = sum(ms_ssims) / len(ms_ssims)
        lpips_promedio = sum(lpips_values) / len(lpips_values)
        # Imprimir las medias de las metricas
        print(f'Modelo {i}: Loss promedio = {loss_promedio}, PSNR
    promedio = {psnr_promedio}, SSIM promedio = {ssim_promedio}, MS-SSIM
    promedio = {ms_ssim_promedio}, LPIPS promedio = {lpips_promedio}'
```

Extracto de código 7.1: Función para probar los modelos

Para esta prueba, se han usado 200 imágenes totalmente desconocidas para los modelos y que tras aplicar el código anterior, finalmente se nos muestran los siguientes resultados (ver tabla 7.1).

Sobre la red GAN (correspondiente a la número 5) no se pueden obtener las métricas de la tabla, ya que su funcionamiento es distinto. Concretamente, necesita otras métricas como el FID, P-IDS y U-IDS, pero gracias a la visualización de imágenes durante el entrenamiento, se comprobó que no era una buena red y se dejó de lado.

Tabla 7.1

Comparación de resultados en los distintos modelos creados.

Redes	Loss (MSE)	PSNR	SSIM	MS-SSIM	LPIPS
Red 1	0.00045	34.6685	0.9860	0.9886	0.0129
Red 2	0.00026	36.4926	0.9792	0.9911	0.0098
Red 3	0.00283	25.6012	0.7549	0.9431	0.3387
Red 4	0.00041	34.0755	0.9799	0.9915	0.0147
Red 5	-	-	-	-	-
Red 6	0.00078	31.1806	0.9436	0.9842	0.0162
Red 7	0.003002	25.3656	0.7356	0.9281	0.2462
Red 8	0.00087	31.3151	0.9637	0.9817	0.0286
Red 9	0.00049	33.3508	0.9523	0.9874	0.0126
Red Final	**0.0000265**	**45.9637**	**0.9942**	**0.9991**	**0.0020**

Según Patil, S. *et al.*[106], ciertos algoritmos de inpainting como *Navier-Stokes*[107] y *Telea*[108], logran un PSNR entre 32 y 34 y un SSIM por encima de 0.96 para pequeñas tareas de inpainting como son en nuestro dataset *DEFACTO*, cuyas máscaras a inpaintear, suelen ser de objetos pequeños.

En las pruebas realizadas, podemos comprobar que en la gran mayoría de redes utilizadas, se han logrado resultados similares, por otro lado, la red final, ha logrado superar estos resultados con creces obteniendo un PSNR de 45.96

Para ver mejor las métricas anteriores, se presentan en forma de tabla de rankings y así visualizar la posición final correspondiente a cada red.

Tabla 7.2

Ranking de resultados para los distintos modelos creados

Redes	Loss (MSE)	PSNR	SSIM	MS-SSIM	LPIPS
Red 1	3	3	2	4	4
Red 2	2	2	4	3	2
Red 3	8	8	8	8	9
Red 4	4	4	3	2	5
Red 5	-	-	-	-	-
Red 6	6	7	7	6	6
Red 7	9	9	9	9	8
Red 8	7	6	5	7	7
Red 9	5	5	6	5	3
Red Final	**1**	**1**	**1**	**1**	**1**

Una vez se ha comprobado que red es la mejor para el inpainting, se ha terminado con el primer caso de uso. Ahora solo queda validar la red con el conjunto de datos ***BDD100K*** que está preparado específicamente para la conducción autónoma. En este caso, el conjunto de prueba consta de 1000 imágenes no vistas nunca por el modelo. Lo validaremos gracias a la comparación de los resultados obtenidos en función del número de imágenes utilizadas durante el entrenamiento. Concretamente, se entrenó con 6000, 12000, 24000 y 48000 imágenes respectivamente. Para discutir y valorar

hasta qué punto merece la pena gastar tantos recursos computacionales durante el entrenamiento, se presenta la siguiente tabla, en la que veremos la comparativa y el tiempo para cada número de imágenes por época en segundos.

Tabla 7.3

Comparación de resultados según el número de imágenes utilizadas para el entrenamiento de la red final para el dataset BDD100K

Imágenes	**Loss (MSE)**	**PSNR**	**SSIM**	**MS-SSIM**	**LPIPS**	**Tiempo x época (s)**
6000	0.00102	30.51908	0.97407	0.9639	0.0321	236
12000	0.001009	30.6063	0.9758	0.9654	0.0303	478
24000	0.00093	31.0047	0.9779	0.9686	0.0286	960
48000	0.00084	31.4826	0.9796	0.9715	0.0279	1920
60000	Descartado por tiempo computacional muy elevado					

Vemos, que al tener cada vez más datos, generaliza cada vez mejor, obteniendo resultados métricamente mejores. Se da por hecho que siguiendo los resultados anteriores, el posible entrenamiento con 60000 imágenes, nos haría obtener aún mejores resultados. Pese a ello, cada aumento de duplicidad en el número de imágenes, implica ese mismo aumento en el tiempo por época, de manera que al aumentar al doble el número de imágenes, aumentará al doble tiempo. Con ello, computacionalmente no merece la pena entrenar un modelo que consiga métricas "cercanas". Ante todo, la decisión final variará de los objetivos y gastos temporales que tenga el encargado de entrenarla.

En este caso, por las fechas y número de horas establecidas en 300 para Trabajo de Fin de Grado, se descartó completamente el entrenamiento de la red con 60000 imágenes.

8. Conclusiones

Una vez realizada la revisión de la literatura, se han comprendido diversos problemas existentes en el campo del *Machine Learning*, concretamente, los relacionados con escenarios de conducción autónoma.

Se han alcanzado todos los objetivos propuestos. Desde crear un modelo que funcione correctamente para aportar una solución al problema del inpainting, hasta adaptar y preparar un dataset específico de conducción autónoma para poder aplicarle el modelo desarrollado.
En concreto, logrando crear un modelo de *Deep Learning*, que resuelve el problema del inpainting en imágenes de conducción autónoma, para que así, un sistema pueda interpretar y completar información visual faltante de manera autónoma.

8.1. Lecciones aprendidas

A lo largo del desarrollo del Trabajo de Fin de Grado, se han adquirido distintas lecciones con una gran utilidad de cara a futuros proyectos.

- Se ha logrado aplicar y seguir correctamente una metodología que han permitido tener una planificación efectiva y organizada.
- Gracias a esta metodología, y su planificación por pequeños *sprints* semanales, se ha logrado resolver problemas complejos de una manera sencilla, gracias a descomponer el TFG en distintas fases.
- Se ha aprendido a buscar distintas soluciones a problemas, gracias a la recopilación e investigación de distintos trabajos en los que se abordaron problemas similares.
- Se ha encontrado la forma de optimizar la preparación de los recursos necesarios y gestionarlos de una forma óptima. Establecer los objetivos alcanzables y buscar mejoras sobre los ya planteados.
- La posibilidad de desarrollar modelos neuronales basados en el estado del arte con resultados aceptables ha sido demostrada. De forma que se abre la puerta a posibles líneas de investigación futuras.

8.2. Trabajo futuro

Debido a la limitación de tiempo para este Trabajo de Fin de Grado, se han identificado varias áreas de expansión y mejora de cara a trabajo futuro.

Se buscará desarrollar una heurística avanzada que permita la colocación de vehículos en imágenes ya existentes de la forma más realista posible, de forma que se generen datasets sintéticos indistinguibles respecto a los reales.

Se tratará de simplificar el modelo final todo lo posible, para así reducir su número de hiperparámetros, de manera que se facilite su aplicación en tiempo real para imágenes capturadas por cámaras reales en vehículos autónomos.

Y por último, se explorará, en la medida de lo posible, aplicar distintos métodos para poder utilizar todas las GPUs que tengamos disponibles de forma simultánea, para así acelerar el entrenamiento, mejorando el aprendizaje y los resultados.

Bibliografía

[1] *Introducing ChatGPT*, https://openai.com/blog/chatgpt, (Accessed on 30-04-2024).

[2] IBM, *¿Qué es un chatbot? — IBM*, https://www.ibm.com/es-es/topics/chatbots, (Accessed on 30-04-2024).

[3] R. Leal, *ChatGPT: Así ve la Inteligencia Artificial el futuro del automóvil — SoyMotor.com*, https://soymotor.com/coches/articulos/chatgpt-asi-ve-la-inteligencia-artificial-el-futuro-del-automovil, (Accessed on 30-04-2024).

[4] E. Sandu, *Inteligencia artificial en la conducción autónoma - metaverso.pro*, https://metaverso.pro/blog/inteligencia-artificial-en-la-conduccion-autonoma/, (Accessed on 30-04-2024).

[5] *La Inteligencia Artificial en el sector del automóvil*, https://www.quadis.es/articulos/la-inteligencia-artificial-en-el-sector-del-automovil/70662614, (Accessed on 30-04-2024), ene. de 2024.

[6] REPSOL, *La Inteligencia Artificial en vehículos autónomos*, https://openroom.fundacionrepsol.com/es/contenidos/la-inteligencia-artificial-en-vehiculos-autonomos/, (Accessed on 30-04-2024).

[7] C. G. Valenzuela, *Inteligencia Artificial revoluciona el sector del automóvil*, https://computerhoy.com/motor/inteligencia-artificial-mundo-sector-automovil-1266250, (Accessed on 30-04-2024).

[8] J. Van Brummelen, M. O'Brien, D. Gruyer y H. Najjaran, «Autonomous vehicle perception: The technology of today and tomorrow,» *Transportation Research Part C: Emerging Technologies*, vol. 89, págs. 384-406, 2018, ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2018.02.012. dirección: https://www.sciencedirect.com/science/article/pii/S0968090X18302134.

[9] A. Geiger, P. Lenz y R. Urtasun, «Are we ready for autonomous driving? The KITTI vision benchmark suite,» en *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, págs. 3354-3361. DOI: 10.1109/CVPR.2012.6248074.

[10] J. P. Colomé, *Raquel Urtasun, the engineer who believes she has found the key to self-driving cars: 'I have no doubts' — Technology — EL PAÍS English*, https://english.elpais.com/technology/2024-01-01/raquel-urtasun-the-engineer-who-believes-she-has-found-the-key-to-self-driving-cars-i-have-no-doubts.html#, (Accessed on 30-04-2024), ene. de 2024.

[11] I. Guerrero, *La navarra Raquel Urtasun, entre los cien principales líderes del mundo en Inteligencia Artificial*, https://navarracapital.es/la-navarra-raquel-urtasun-entre-los-cien-principales-lideres-del-mundo-en-inteligencia-artificial/, (Accessed on 30-04-2024), dic. de 2023.

[12] *Fundacion Cidaut*, https://www.cidaut.es/es/sobre-nosotros/cidaut, (Accessed on 30-04-2024).

[13] L. Ma, T. Ma, R. Liu, X. Fan y Z. Luo, *Toward Fast, Flexible, and Robust Low-Light Image Enhancement*, 2022. arXiv: 2204.10137 [cs.CV].

[14] K. Zhang, W. Ren, W. Luo et al., «Deep image deblurring: A survey,» *International Journal of Computer Vision*, vol. 130, n.º 9, págs. 2103-2130, 2022.

[15] J.-E. Campagne, *Denoising: from classical methods to deep CNNs*, 2024. arXiv: 2404.16617 [cs.CV].

[16] W. Quan, J. Chen, Y. Liu, D.-M. Yan y P. Wonka, *Deep Learning-based Image and Video Inpainting: A Survey*, 2024. arXiv: 2401.03395 [cs.CV].

[17] C. Drumond, *What is Scrum? [+ How to Start] — Atlassian*, https://www.atlassian.com/agile/scrum, (Accessed on 30-04-2024).

[18] Google, *¿Qué es la inteligencia artificial o IA? — Google Cloud — Google Cloud — cloud.google.com*, [Accessed 14-04-2024]. dirección: https://cloud.google.com/learn/what-is-artificial-intelligence?hl=es-419#section-2.

[19] J. Mei, Y. Ma, X. Yang et al., *Continuously Learning, Adapting, and Improving: A Dual-Process Approach to Autonomous Driving*, 2024. arXiv: 2405.15324 [cs.RO].

[20] F. Liu, Z. Lu y X. Lin, *Vision-Based Environmental Perception for Autonomous Driving*, 2022. arXiv: 2212.11453 [cs.CV].

[21] *Everything You Ever Wanted To Know About Computer Vision. — by Ilija Mihajlovic — Towards Data Science*, https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e, (Accessed on 28/05/2024).

[22] *Computer Vision & OpenCV — AI Planet (formerly DPhi)*, https://aiplanet.com/learn/getting-started-with-deep-learning-es/convolutional-neural-networks/1607/computer-vision-opencv, (Accessed on 28/05/2024).

[23] *An Overview of Computer Vision. Computer vision is a field of... — by Michelle Venables — Towards Data Science*, https://towardsdatascience.com/an-overview-of-computer-vision-1f75c2ab1b66, (Accessed on 28/05/2024).

[24] *Computer Vision – 22 Technologies*, https://22-tech.com/computer-vision/, (Accessed on 28/05/2024).

[25] E. Alpaydin, *Introduction to Machine Learning*, 4.ª ed. The MIT Press, 2020.

[26] S. R. Dubey, S. K. Singh y B. B. Chaudhuri, *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*, 2022. arXiv: 2109.14545 [cs.LG].

[27] M. Franke y J. Degen, «The softmax function: Properties, motivation, and interpretation,» 2023.

[28] A. Entezami, H. Sarmadi y B. Saeedi Razavi, «An innovative hybrid strategy for structural health monitoring by modal flexibility and clustering methods,» *Journal of Civil Structural Health Monitoring*, vol. 10, n.º 4, págs. 845-859, 2020.

[29] J. Liang, J. Cui, J. Wang y W. Wei, «Graph-based semi-supervised learning via improving the quality of the graph dynamically,» *Machine Learning*, vol. 110, págs. 1345-1388, 2021. DOI: 10.1007/s10994-021-05975-y.

[30] A. Sherstinsky, «Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network,» *Physica D: Nonlinear Phenomena*, vol. 404, pág. 132 306, 2020, ISSN: 0167-2789. DOI: https://doi.org/10.1016/j.physd.2019.132306. dirección: https://www.sciencedirect.com/science/article/pii/S0167278919305974.

[31] *Introduction to RNN — cs231n.github.io*, CS231n: Deep Learning for Computer Vision, [Accessed 19-04-2024]. dirección: https://cs231n.github.io/rnn/.

[32] J. Elman, «Finding structure in time,» *Cognitive Science*, vol. 14, págs. 179-211, 1990.

[33] M. Jordan, «Attractor dynamics and parallelism in a connectionist sequential machine,» *Proc. of the Eighth Annual Conference of the Cognitive Science Society*, págs. 531-546, 1986.

[34] M. Jordan, «Serial Order: a parallel distributed processing approach,» Institute for Cognitive Science. University of California, inf. téc., 1986.

[35] P. J. Werbos, «Backpropagation through time: what it does and how to do it,» *Proceedings of the IEEE*, vol. 78, n.º 10, págs. 1550-1560, 1990.

[36] S. Hochreiter y J. Schmidhuber, «Long short-term memory,» *Neural computation*, vol. 9, n.º 8, págs. 1735-1780, 1997.

[37] Wikipedia, the free encyclopedia, *Long short-term memory*, [Accessed 24-04-2024]. dirección: https://commons.wikipedia.org/wiki/File:Peephole%20Long%20Short-Term%20Memory.svg.

[38] *CS231n Convolutional Neural Networks for Visual Recognition — cs231n.github.io*, CS231n: Deep Learning for Computer Vision, [Accessed 18-04-2024]. dirección: https://cs231n.github.io/convolutional-networks/.

[39] *Introduction to Convolution Neural Network - GeeksforGeeks*, https://www.geeksforgeeks.org/introduction-convolution-neural-network/, (Accessed on 27/05/2024).

[40] V. N. Balasubramanian, *NOC:Deep Learning for Computer Vision, IIT Hyderabad*, https://nptel.ac.in/courses/106106224, (Accessed on 25-04-2024).

[41] B. Xu, N. Wang, T. Chen y M. Li, «Empirical Evaluation of Rectified Activations in Convolutional Network,» *arXiv preprint arXiv:1505.00853*, 2015.

[42] F. Bieder, R. Sandkühler y P. C. Cattin, *Comparison of Methods Generalizing Max- and Average-Pooling*, 2021. arXiv: 2103.01746 [cs.CV].

[43] I. Thottam, *The Cost of Conservation and Restoration - Art Business News*, https://artbusinessnews.com/2015/12/the-cost-of-conservation-and-restoration/, (Accessed on 26-04-2024).

[44] A. A. Efros y T. K. Leung, «Texture synthesis by non-parametric sampling,» Cited by: 2585, vol. 2, 1999, págs. 1033-1038. dirección: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0033285309&partnerID=40&md5=b329de25ee598a8d0370aac9cbfac7ea.

[45] M. Bertalmio, G. Sapiro, V. Caselles y C. Ballester, «Image inpainting,» en *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, ép. SIGGRAPH '00, USA: ACM Press/Addison-Wesley Publishing Co., 2000, págs. 417-424, ISBN: 1581132085. DOI: 10.1145/344779.344972. dirección: https://doi.org/10.1145/344779.344972.

[46] C. Barnes, E. Shechtman, A. Finkelstein y D. B. Goldman, «PatchMatch: A randomized correspondence algorithm for structural image editing,» *ACM Trans. Graph.*, vol. 28, n.º 3, pág. 24, 2009.

[47] T. Ružić y A. Pižurica, «Context-aware patch-based image inpainting using Markov random field modeling,» *IEEE transactions on image processing*, vol. 24, n.º 1, págs. 444-456, 2014.

[48] H. Liu, X. Bi, G. Lu y W. Wang, «Exemplar-based image inpainting with multi-resolution information and the graph cut technique,» *IEEE Access*, vol. 7, págs. 101 641-101 657, 2019.

[49] M. Bertalmio, A. L. Bertozzi y G. Sapiro, «Navier-stokes, fluid dynamics, and image and video inpainting,» en *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, IEEE, vol. 1, 2001, págs. I-I.

[50] J. Shen y T. F. Chan, «Mathematical models for local nontexture inpaintings,» *SIAM Journal on Applied Mathematics*, vol. 62, n.º 3, págs. 1019-1043, 2002.

[51] J. Shen, S. H. Kang y T. F. Chan, «Euler's elastica and curvature-based inpainting,» *SIAM journal on Applied Mathematics*, vol. 63, n.º 2, págs. 564-592, 2003.

[52] S. Liu, Y. Wang, J. Wang, H. Wang, J. Zhang y C. Pan, «Kinect depth restoration via energy minimization with tv 21 regularization,» en *2013 IEEE International Conference on Image Processing*, IEEE, 2013, págs. 724-724.

[53] C. Guillemot y O. Le Meur, «Image inpainting: Overview and recent advances,» *IEEE signal processing magazine*, vol. 31, n.º 1, págs. 127-144, 2013.

[54] H. Li, W. Luo y J. Huang, «Localization of diffusion-based inpainting in digital images,» *IEEE transactions on information forensics and security*, vol. 12, n.º 12, págs. 3050-3064, 2017.

[55] T. K. Shih, L.-C. Lu, Y.-H. Wang y R.-C. Chang, «Multi-resolution image inpainting,» en *2003 International Conference on Multimedia and Expo. ICME'03. Proceedings (Cat. No. 03TH8698)*, IEEE, vol. 1, 2003, págs. I-485.

[56] J. Mairal, M. Elad y G. Sapiro, «Sparse representation for color image restoration,» *IEEE Transactions on image processing*, vol. 17, n.º 1, págs. 53-69, 2007.

[57] R.-C. Chang y T. K. Shih, «Multilayer lnpainting on Digitalized Artworks.,» *Journal of Information Science & Engineering*, vol. 24, n.º 4, 2008.

[58] N. Kawai, T. Sato y N. Yokoya, «Image inpainting considering brightness change and spatial locality of textures and its evaluation,» en *Advances in Image and Video Technology: Third Pacific Rim Symposium, PSIVT 2009, Tokyo, Japan, January 13-16, 2009. Proceedings 3*, Springer, 2009, págs. 271-282.

[59] B. Shen, W. Hu, Y. Zhang e Y.-J. Zhang, «Image inpainting via sparse representation,» en *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 2009, págs. 697-700.

[60] L. He, Y. Xing, K. Xia, J. Tan et al., «An adaptive image inpainting method based on continued fractions interpolation,» *Discrete Dynamics in Nature and Society*, vol. 2018, 2018.

[61] J. Jam, C. Kendrick, K. Walker, V. Drouard, J. G.-S. Hsu y M. H. Yap, «A comprehensive review of past and present image inpainting methods,» *Computer Vision and Image Understanding*, vol. 203, pág. 103 147, 2021, ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2020.103147. dirección: https://www.sciencedirect.com/science/article/pii/S1077314220301661.

[62] G. Brauwers y F. Frasincar, «A General Survey on Attention Mechanisms in Deep Learning,» *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, n.º 4, págs. 3279-3298, abr. de 2023, ISSN: 2326-3865. DOI: 10.1109/tkde.2021.3126456. dirección: http://dx.doi.org/10.1109/TKDE.2021.3126456.

[63] D. Soydaner, «Attention mechanism in neural networks: where it comes and where it goes,» *Neural Computing and Applications*, vol. 34, n.º 16, págs. 13 371-13 385,

mayo de 2022, ISSN: 1433-3058. DOI: 10.1007/s00521-022-07366-3. dirección: http://dx.doi.org/10.1007/s00521-022-07366-3.

[64] A. E. Orhan y X. Pitkow, *Skip Connections Eliminate Singularities*, 2018. arXiv: 1701.09175 [cs.NE].

[65] D. Wu, Y. Wang, S.-T. Xia, J. Bailey y X. Ma, *Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets*, 2020. arXiv: 2002.05990 [cs.LG].

[66] V. Badrinarayanan, A. Kendall y R. Cipolla, *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*, 2016. arXiv: 1511.00561 [cs.CV].

[67] J. C. Ye y W. K. Sung, *Understanding Geometry of Encoder-Decoder CNNs*, 2019. arXiv: 1901.07647 [cs.LG].

[68] K. Aitken, V. V. Ramasesh, Y. Cao y N. Maheswaranathan, *Understanding How Encoder-Decoder Architectures Attend*, 2021. arXiv: 2110.15253 [cs.LG].

[69] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza et al., *Generative Adversarial Networks*, 2014. arXiv: 1406.2661 [stat.ML].

[70] T. S. Silva, «A Short Introduction to Generative Adversarial Networks,» 2017.

[71] O. Ronneberger, P. Fischer y T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015. arXiv: 1505.04597 [cs.CV].

[72] H. Liu, B. Jiang, Y. Xiao y C. Yang, *Coherent Semantic Attention for Image Inpainting*, 2019. arXiv: 1905.12384 [cs.CV].

[73] H. Zheng, Z. Lin, J. Lu et al., *CM-GAN: Image Inpainting with Cascaded Modulation GAN and Object-Aware Training*, 2022. arXiv: 2203.11947 [cs.CV].

[74] R. E. Turner, *An Introduction to Transformers*, 2024. arXiv: 2304.10557 [cs.LG].

[75] W. Li, Z. Lin, K. Zhou, L. Qi, Y. Wang y J. Jia, *MAT: Mask-Aware Transformer for Large Hole Image Inpainting*, 2022. arXiv: 2203.15270 [cs.CV].

[76] P. Jeevan, D. S. Kumar y A. Sethi, *WavePaint: Resource-efficient Token-mixer for Self-supervised Inpainting*, 2023. arXiv: 2307.00407 [cs.CV].

[77] S. Zhao, J. Cui, Y. Sheng et al., *Large Scale Image Completion via Co-Modulated Generative Adversarial Networks*, 2021. arXiv: 2103.10428 [cs.CV].

[78] Wikipedia, *GNU/Linux*, Wikipedia, la enciclopedia libre, Accessed: 15-Apr-2024, 2024. dirección: https://es.wikipedia.org/wiki/GNU/Linux.

[79] Wikipedia, *Microsoft Windows*, Wikipedia, la enciclopedia libre, Accessed: 15-Apr-2024, 2024. dirección: https://es.wikipedia.org/wiki/Microsoft_Windows.

[80] Canonical Ltd., *Ubuntu 22.04 LTS (Jammy Jellyfish)*, https://releases.ubuntu.com/jammy/, Accessed: 15-Apr-2024, 2024.

[81] Python Software Foundation, *Python 3.10.14*, https://www.python.org/downloads/release/python-31014/, Accessed: 15-Apr-2024, 2024.

[82] Universidad de Valladolid, *Minería de Datos - Guía Docente de la Universidad de Valladolid*, https://apps.stic.uva.es/guias_docentes/uploads/2023/551/46970/1/Documento.pdf, Accessed: 15-Apr-2024, 2023.

[83] P. S. Foundation, *venv — Creación de entornos virtuales*, Python 3.8 documentation, Accessed: 15-Apr-2024, 2024. dirección: https://docs.python.org/es/3.8/library/venv.html.

[84] *OpenCV: Introduction to OpenCV-Python Tutorials*, OpenCV documentation index, Accessed: 15-Apr-2024, 2024. dirección: https://docs.opencv.org/4.x/d0/de3/tutorial_py_intro.html.

[85] *pillow*, PyPI, Accessed: 15-Apr-2024, 2024. dirección: https://pypi.org/project/pillow/.

[86] F. en ciencia de datos — DataScientest.com, *Matplotlib: todo lo que tienes que saber sobre la librería Python de Dataviz*, Accessed: 15-Apr-2024, 2024. dirección: https://datascientest.com/es/todo-sobre-matplotlib.

[87] PyTorch, *PyTorch*, PyTorch official website, Accessed: 15-Apr-2024, 2024. dirección: https://pytorch.org/.

[88] NVIDIA Corporation, *Latest Official NVIDIA Driver*, https://www.nvidia.com/Download/index.aspx, Accessed: 15-Apr-2024, 2024.

[89] NVIDIA Corporation, *CUDA Toolkit 12.4 Update 1*, https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html, Accessed: 15-Apr-2024, 2024.

[90] Matplotlib Development Team, *Matplotlib 3.8.4*, https://pypi.org/project/matplotlib/, Accessed: 15-Apr-2024, 2024.

[91] *Weights & Biases*, Weights & Biases, Accessed: 15-Apr-2024, 2024. dirección: https://wandb.ai/.

[92] G. MAHFOUDI, B. TAJINI, F. RETRAINT, F. MORAIN-NICOLIER, J. L. DUGELAY y M. PIC, «DEFACTO: Image and Face Manipulation Dataset,» en *2019 27th European Signal Processing Conference (EUSIPCO)*, 2019, págs. 1-5. DOI: 10.23919/EUSIPCO.2019.8903181.

[93] T. Lin, M. Maire, S. J. Belongie et al., «Microsoft COCO: Common Objects in Context,» *CoRR*, vol. abs/1405.0312, 2014. arXiv: 1405.0312. dirección: http://arxiv.org/abs/1405.0312.

[94] N. Samet, S. Hicsonmez y E. Akbas, «HoughNet: Integrating near and long-range evidence for bottom-up object detection,» en *European Conference on Computer Vision (ECCV)*, 2020.

[95] Y. Cabon, N. Murray y M. Humenberger, *Virtual KITTI 2*, 2020. arXiv: 2001.10773 [cs.CV].

[96] F. Yu, H. Chen, X. Wang et al., *BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning*, 2020. arXiv: 1805.04687 [cs.CV].

[97] D. P. Kingma y J. Ba, *Adam: A Method for Stochastic Optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[98] A. Y. Ng, «Feature selection, L 1 vs. L 2 regularization, and rotational invariance,» en *Proceedings of the twenty-first international conference on Machine learning*, 2004, pág. 78.

[99] C. Cortes, M. Mohri y A. Rostamizadeh, *L2 Regularization for Learning Kernels*, 2012. arXiv: 1205.2653 [cs.LG].

[100] Z. Liu, Z. Xu, J. Jin, Z. Shen y T. Darrell, *Dropout Reduces Underfitting*, 2023. arXiv: 2303.01500 [cs.LG].

[101] Y. Li, W. Ma, C. Chen et al., *A Survey on Dropout Methods and Experimental Verification in Recommendation*, 2022. arXiv: 2204.02027 [cs.LG].

[102] I. Loshchilov y F. Hutter, *SGDR: Stochastic Gradient Descent with Warm Restarts*, 2017. arXiv: 1608.03983 [cs.LG].

[103] A. Al-Kababji, F. Bensaali y S. P. Dakua, *Scheduling Techniques for Liver Segmentation: ReduceLROnPlateau Vs OneCycleLR*, 2022. arXiv: 2202.06373 [cs.CV].

[104] L. N. Smith, *Cyclical Learning Rates for Training Neural Networks*, 2017. arXiv: 1506.01186 [cs.CV].

[105] T. Akiba, S. Sano, T. Yanase, T. Ohta y M. Koyama, *Optuna: A Next-generation Hyperparameter Optimization Framework*, 2019. arXiv: 1907.10902 [cs.LG].

[106] S. Patil, A. Joshi y S. Sawant, «Recovering Images Using Image Inpainting Techniques,» en *Robotics, Control and Computer Vision*, H. Muthusamy, J. Botzheim y R. Nayak, eds., Singapore: Springer Nature Singapore, 2023, págs. 27-38, ISBN: 978-981-99-0236-1.

[107] M. A. Ebrahimi, M. Holst y E. Lunasin, *The Navier-Stokes-Voight Model for Image Inpainting*, 2009. arXiv: 0901.4548 [math.NA].

[108] A. Telea, «An image inpainting technique based on the fast marching method,» *Journal of graphics tools*, vol. 9, n.º 1, págs. 23-34, 2004.

[109] S. N. Gowda, Y. Thakre, S. N. Gowda y X. Jin, *Reimagining Reality: A Comprehensive Survey of Video Inpainting Techniques*, 2024. arXiv: 2401.17883 [cs.CV].

[110] Z. Wang, A. C. Bovik, H. R. Sheikh y E. P. Simoncelli, «Image quality assessment: from error visibility to structural similarity,» *IEEE transactions on image processing*, vol. 13, n.º 4, págs. 600-612, 2004.

[111] R. Zhang, P. Isola, A. A. Efros, E. Shechtman y O. Wang, *The Unreasonable Effectiveness of Deep Features as a Perceptual Metric*, 2018. arXiv: 1801.03924 [cs.CV].

[112] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler y S. Hochreiter, *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, 2018. arXiv: 1706.08500 [cs.LG].

[113] *Conv2d — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html, (Accessed on 15-05-2024).

[114] *MaxPool2d — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html, (Accessed on 15-05-2024).

[115] *ConvTranspose2d — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html, (Accessed on 15-05-2024).

[116] *BatchNorm2d — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html, (Accessed on 15-05-2024).

[117] *Linear — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.Linear.html, (Accessed on 15-05-2024).

[118] *Sigmoid — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html, (Accessed on 15-05-2024).

[119] *ReLU — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html, (Accessed on 15-05-2024).

[120] *Tanh — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html, (Accessed on 15-05-2024).

[121] *LSTM — PyTorch 2.3 documentation*, https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html, (Accessed on 15-05-2024).

[122] G. Jocher, A. Chaurasia y J. Qiu, *Ultralytics YOLO*, ver. 8.0.0, ene. de 2023. dirección: https://github.com/ultralytics/ultralytics.

[123] J. Redmon, S. Divvala, R. Girshick y A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, 2016. arXiv: 1506.02640 [cs.CV].

[124] J. Redmon y A. Farhadi, *YOLO9000: Better, Faster, Stronger*, 2016. arXiv: 1612.08242 [cs.CV].

[125] J. Redmon y A. Farhadi, *YOLOv3: An Incremental Improvement*, 2018. arXiv: 1804.02767 [cs.CV].

[126] A. Bochkovskiy, C.-Y. Wang y H.-Y. M. Liao, *YOLOv4: Optimal Speed and Accuracy of Object Detection*, 2020. arXiv: 2004.10934 [cs.CV].

[127] C. Li, L. Li, H. Jiang et al., *YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications*, 2022. arXiv: 2209.02976 [cs.CV].

[128] C.-Y. Wang, A. Bochkovskiy y H.-Y. M. Liao, *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*, 2022. arXiv: 2207.02696 [cs.CV].

[129] C.-Y. Wang, I.-H. Yeh y H.-Y. M. Liao, *YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information*, 2024. arXiv: 2402.13616 [cs.CV].

[130] J. Terven, D.-M. Córdova-Esparza y J.-A. Romero-González, «A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS,» *Machine Learning and Knowledge Extraction*, vol. 5, n.º 4, págs. 1680-1716, nov. de 2023, ISSN: 2504-4990. DOI: 10.3390/make5040083. dirección: http://dx.doi.org/10.3390/make5040083.

[131] *YOLOv9 - Ultralytics YOLO Documentos*, https://docs.ultralytics.com/es/models/yolov9/, (Accessed on 06-06-2024).

[132] *Detect - Ultralytics YOLO Docs*, https://docs.ultralytics.com/tasks/detect/, (Accessed on 06-06-2024).

[133] *Segment - Ultralytics YOLO Docs*, https://docs.ultralytics.com/tasks/segment/, (Accessed on 06-06-2024).

[134] Y. LeCun, L. Bottou, Y. Bengio y P. Haffner, «Gradient-Based Learning Applied to Document Recognition,» *Proceedings of the IEEE*, vol. 86, n.º 11, págs. 2278-2324, nov. de 1998.

[135] Y. LeCun, *LeNet-5: Unusual Patterns*, http://yann.lecun.com/exdb/lenet/weirdos.html, (Accessed on 29-04-2024).

[136] A. Krizhevsky, I. Sutskever y G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks,» en *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou y K. Weinberger, eds., vol. 25, Curran Associates, Inc., 2012. dirección: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[137] M. D. Zeiler y R. Fergus, *Visualizing and Understanding Convolutional Networks*, 2013. arXiv: 1311.2901 [cs.CV].

[138] C. Szegedy, W. Liu, Y. Jia et al., *Going Deeper with Convolutions*, 2014. arXiv: 1409.4842 [cs.CV].

[139] K. Simonyan y A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[140] S. Banga, *VGG-Net Architecture Explained. The company Visual Geometry Group... — by Siddhesh Bangar — Medium*, https://medium.com/@siddheshb008/vgg-net-architecture-explained-71179310050f, (Accessed on 29-04-2024).

[141] K. He, X. Zhang, S. Ren y J. Sun, *Deep Residual Learning for Image Recognition*, 2015. arXiv: 1512.03385 [cs.CV].

[142] S. Ioffe y C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015. arXiv: 1502.03167 [cs.LG].

[143] C.-T. Li, *10 Papers You Must Read for Deep Image Inpainting — Towards Data Science*, https://towardsdatascience.com/10-papers-you-must-read-for-deep-image-inpainting-2e41c589ced0, (Accessed on 02-05-2024).

[144] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell y A. A. Efros, *Context Encoders: Feature Learning by Inpainting*, 2016. arXiv: 1604.07379 [cs.CV].

[145] C. Yang, X. Lu, Z. Lin, E. Shechtman, O. Wang y H. Li, «High-resolution image inpainting using multi-scale neural patch synthesis,» en *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, págs. 6721-6729.

[146] S. Iizuka, E. Simo-Serra y H. Ishikawa, «Globally and locally consistent image completion,» *ACM Trans. Graph.*, vol. 36, n.º 4, jul. de 2017, ISSN: 0730-0301. DOI: 10.1145/3072959.3073659. dirección: https://doi.org/10.1145/3072959.3073659.

[147] U. Demir y G. Unal, *Patch-Based Image Inpainting with Generative Adversarial Networks*, 2018. arXiv: 1803.07422 [cs.CV].

[148] P. Isola, J.-Y. Zhu, T. Zhou y A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, 2018. arXiv: 1611.07004 [cs.CV].

[149] Z. Yan, X. Li, M. Li, W. Zuo y S. Shan, *Shift-Net: Image Inpainting via Deep Feature Rearrangement*, 2018. arXiv: 1801.09392 [cs.CV].

[150] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu y T. S. Huang, *Generative Image Inpainting with Contextual Attention*, 2018. arXiv: 1801.07892 [cs.CV].

[151] Y. Wang, X. Tao, X. Qi, X. Shen y J. Jia, *Image Inpainting via Generative Multi-column Convolutional Neural Networks*, 2018. arXiv: 1810.08771 [cs.CV].

[152] G. Liu, F. A. Reda, K. J. Shih, T.-C. Wang, A. Tao y B. Catanzaro, *Image Inpainting for Irregular Holes Using Partial Convolutions*, 2018. arXiv: 1804.07723 [cs.CV].

[153] K. Nazeri, E. Ng, T. Joseph, F. Z. Qureshi y M. Ebrahimi, *EdgeConnect: Generative Image Inpainting with Adversarial Edge Learning*, 2019. arXiv: 1901.00212 [cs.CV].

[154] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu y T. Huang, *Free-Form Image Inpainting with Gated Convolution*, 2019. arXiv: 1806.03589 [cs.CV].

Anexos

A. Métricas para inpainting.

En tareas de *Computer Vision*, para garantizar que las soluciones propuestas sobre distintos problemas son adecuadas, hemos de tener algo que nos pueda guiar. En este caso, gracias a distintas métricas existentes, podemos garantizar que las soluciones propuestas a las tareas de *Computer Vision*, sean lo más perfectas posible. De forma que si una de esas soluciones es mejor que el resto de las propuestas, será considerada como el SOTA en ese problema concreto.

En este anexo, se explican una a una las distintas métricas que se han utilizado en este TFG, ya que se han usado para valorar y validar como de buenos han sido los resultados obtenidos por los distintos modelos creados a la hora de solucionar el problema del inpainting.

Si se quieren ver las ventajas y limitaciones de las distintas métricas presentadas, recientemente se ha discutido por Shereyank N Gowda, *et al.* en el paper *Reimagining Reality: A Comprehensive Survey of Video Inpainting Techniques* [109].

A.1. MSE (Error de mínimos cuadrados)

El error MSE, mide la diferencia promedio al cuadrado entre los valores de los píxeles de la imagen *ground truth* y la imagen generada por nuestro modelo. Esta métrica es común para evaluar de forma matemática la calidad de las imágenes construidas artificialmente. Su cálculo es el siguiente:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(I_{\text{original}}(i) - I_{\text{generada}}(i))^2$$

Donde:

- N es el número total de píxeles.
- $I_{\text{original}}(i)$ es el valor del píxel en la imagen original.
- $I_{\text{generada}}(i)$ es el valor del píxel en la imagen generada.

Su valor óptimo, que indica la perfección, es 0.

A.2. PSNR (*Peak Signal-to-Noise Ratio*)

El PSNR, se podría considerar el inverso al MSE. Pese a ello no es así del todo, pero a menor MSE, mayor PSNR. Mide la relación entra la energía de la señal, que es la imagen original, y el ruido, que es la diferencia entre la imagen original y la imagen

generada por nuestro modelo. Esta diferencia se expresa en decibelios (dB) y su fórmula es la siguiente:

$$PSNR = 20 \cdot \log_{10} \frac{1}{\sqrt{MSE}}$$

En este caso, no hay un valor determinado que indique la perfección, cuanto mayor sean los resultados mejores serán los modelos y las imágenes generadas. En el caso del inpainting, hemos visto en el capítulo 7 que para pequeñas tareas de inpainting, es habitual lograr un PSNR entre 32 y 34.

A.3. SSIM (*Structural Similarity Index*)

El SSIM es una métrica que trata de representar en la medida de lo posible la percepción de la vista humana. Fue propuesta por Wang *et al.* en el paper *Image Quality Assessment: Similarity* [110] ella se comparan varios puntos, como la estructura, la luminancia y el contraste entre dos imágenes. Cuanto más cercano sea a 1, habrá mayor similitud estructural, por lo que a nuestro ojo, la imagen generada será lo más parecida posible a la original. Su fórmula es la siguiente:

$$SSIM(x, y) = [l(x, y)]^{\alpha} \cdot [c(x, y)]^{\beta} \cdot [s(x, y)]^{\gamma}$$

Donde:

- $l(x, y)$ es la función de comparación de luminancia.
- $c(x, y)$ es la función de comparación de contraste.
- $s(x, y)$ es la función de comparación de estructura.
- α, β, γ son los parámetros que ajustan la importancia relativa de cada componente

A.4. MSSIM (SSIM medio)

Según los trabajadores de NVIDIA, Nilson y Akenine-Möller, el SSIM, tiene sus limitaciones, pudiendo provocar resultados inesperados, llevando a conclusiones incorrectas sobre la calidad de la imagen generada. Y si encima, se utiliza como función de pérdida en el entrenamiento de un modelo de *Deep Learning*, podría guiarlo mal. Por lo mencionado anteriormente se descubrió el MS-SIM, que es el promedio del SSIM sobre toda la imagen, dando una medida de similitud para toda la imagen completa en lugar de pixel a pixel.

Su fórmula es la siguiente:

$$MSSIM = \frac{1}{M} \sum_{i=1}^{M} SSIM(x_i, y_i)$$

Donde:

- M es el número total de píxeles en la imagen.
- x_i, y_i son las coordenadas de los píxeles en las imágenes que se están comparando.

De nuevo, cuanto más cerca esté de 1, más cerca estará la imagen generada de la perfección. Además, si es superior a 0.99, se le considera prácticamente perfecta, ya que la diferencia respecto a la original es imperceptible al ojo humano.

A.5. LPIPS (*Learned Perceptual Image Patch Similarity*)

Propuesto por Zhang *et al.* en su paper *The Unreasonable Effectiveness of Deep Features as a Perceptual Metric* [111]. En él nos explican que se utiliza para evaluar la similitud perceptual entre imágenes, que en nuestro caso son la original y la generada. Nos comentan que esta métrica funciona bien en métodos tanto supervisados como no supervisados y que supera a todas las que hemos discutido anteriormente como media de percepción humana.

A.6. Conclusiones

Hemos visto las métricas utilizadas durante nuestro entrenamiento y evaluación de los distintos modelos generados, viendo para qué sirve cada una y que es lo que representa.

Gracias a esta explicación, ya podemos entender los resultados obtenidos en detalle, y gracias a ello podemos comparar como de mejores o peores son unos respecto a otros en la sección de resultados 7.

Prácticamente, todas nuestras redes utilizan algoritmos supervisados, para las que utilizan algoritmos no supervisados como lo son las redes GAN, se utilizan otras métricas como el **FID** (*Frechet Inception Distance*) [112], el **P-IDS** (*Paired Inception Discriminative Score*) y **U-IDS** (*Unpaired Inception Discriminative Score*) [77]

B. Glosario y ejemplos de redes CNN en PyTorch.

En este anexo, se presentan cada uno de los métodos utilizados de PyTorch en las distintas redes del capítulo 6.

- **nn.Conv2d**[113]: Operación que aplica la convolución sobre una entrada.
- **nn.MaxPool2d**[114]: Aplicación del *MaxPooling* en PyTorch. Donde, tomando el valor máximo, reduce la dimensionalidad de los mapas de características manteniendo la información más importante.
- **nn.ConvTranspose2d**[115]: Aplica una convolución transpuesta o deconvolución sobre la entrada. Se suele utilizar para aumentar la dimensionalidad de los mapas de características junto con un *stride* = 2.
- **nn.BatchNorm2d**[116]: Conocida como *Batch Normalization*, aplica una normalización de las entradas a cada capa de la red neuronal.
- **nn.Linear**[117]: Operación que aplica una transformación lineal a la entrada. Se suele usar en las capas totalmente conectadas de una red.
- **nn.Sigmoid**[118]: Función de activación que coloca los valores de la entrada en un rango de 0 a 1.
- **nn.ReLU**[119]: Es la función de activación que pone los valores negativos a 0. Tiene variaciones como GeLU y LeakyReLU.
- **nn.Tanh**[120]: Función de activación que pone los valores entre el rango -1 y 1. Sirve para introducir no linearidad.
- **nn.LSTM**[121]: Es una variante de las RNN, con mecanismos para solucionar la evanescencia del gradiente.

No se profundiza en explicar que es una convolución o una deconvolución, como otros términos que ya se han explicado en detalle que son en el Marco Teórico, correspondiente al capítulo 3. Y si se quiere tomar más información sobre su funcionamiento, recomiendo consultar la documentación de PyTorch referenciada para cada uno de los términos.

B.1. Ejemplos de aplicación de redes convolucionales

A continuación mediante figuras representamos como cambia la convolución y la deconvolución en función de los parámetros de entrada que tengan sus respectivas funciones **Conv2d** y **ConvTranspose2d** como el relleno(*padding*), entrada (*input*) y *stride* (tamaño de paso al recorrer la matriz), mencionando también sus equivalencias.

B.1.1. Convolución

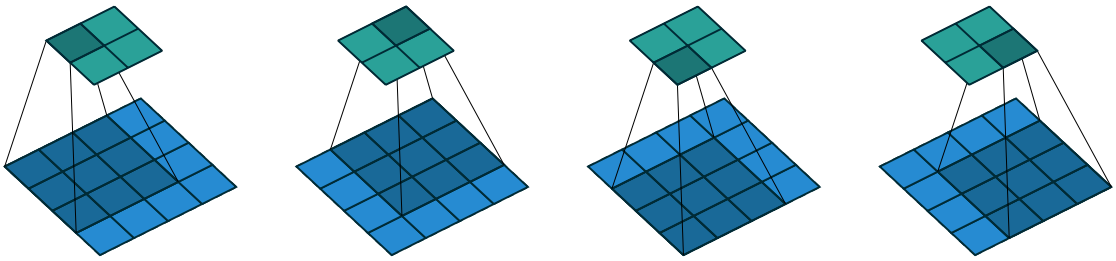


Figura B.1: Convolución de un kernel 3×3 sobre una entrada de 4×4 utilizando stride 1 y sin padding (es decir, $i = 4$, $k = 3$, $s = 1$ y $p = 0$).

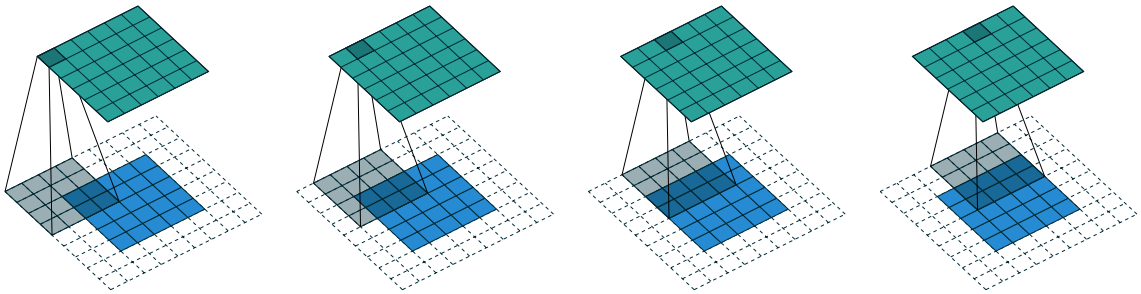


Figura B.2: Convolución de un kernel 4×4 sobre una entrada de 5×5 rellenada con un borde de ceros de 2×2 utilizando stride de 1 y padding $= 2$ (es decir, $i = 5$, $k = 4$, $s = 1$ y $p = 2$).

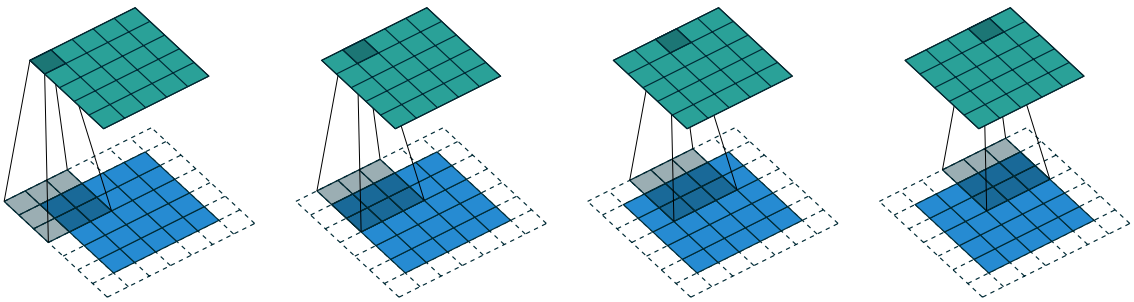


Figura B.3: Convolución de un kernel 3×3 sobre una entrada de 5×5 utilizando relleno 1 y stride 1 (es decir, $i = 5$, $k = 3$, $s = 1$ y $p = 1$).

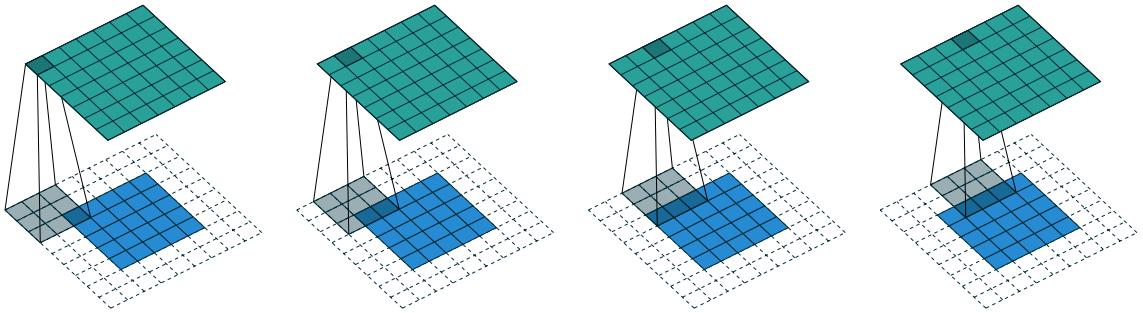


Figura B.4: Convolución de un kernel 3×3 sobre una entrada de 5×5 utilizando relleno y stride arbitrarios (es decir, $i = 5$, $k = 3$, $s = 1$ y $p = 2$).

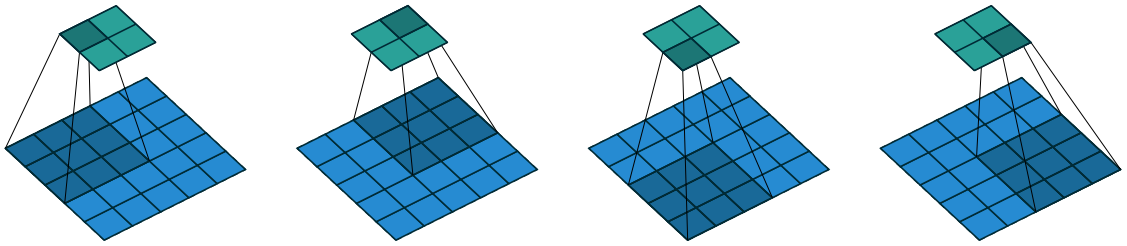


Figura B.5: Convolución de un kernel 3×3 sobre una entrada de 5×5 utilizando un stride de 2×2 y sin relleno (es decir, $i = 5$, $k = 3$, $s = 2$ y $p = 0$).

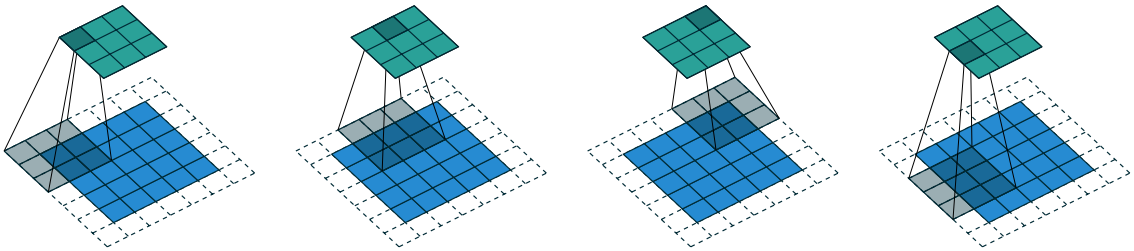


Figura B.6: Convolución de un kernel 3×3 sobre una entrada de 5×5 rellenada con un borde de ceros de 1×1 utilizando un stride de 2×2 (es decir, $i = 5$, $k = 3$, $s = 2$ y $p = 1$).

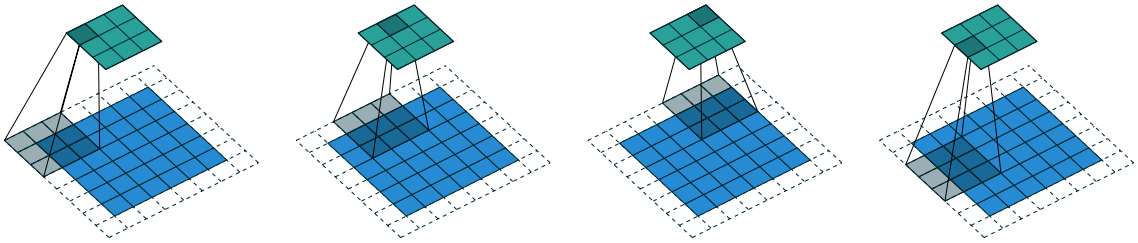


Figura B.7: Convolución de un kernel 3×3 sobre una entrada de 6×6 rellenada con un borde de ceros de 1×1 utilizando un stride de 2×2 (es decir, $i = 6$, $k = 3$, $s = 2$ y $p = 1$). En este caso, la fila inferior y la columna derecha de la entrada rellenada con ceros no están cubiertas por el kernel.

B.1.2. Deconvolución

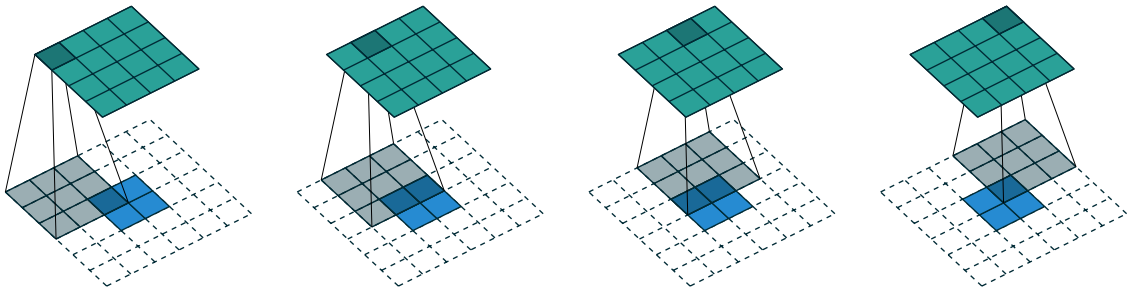


Figura B.8: Convolución transpuesta de un kernel 3×3 sobre una entrada de 4×4 utilizando un stride $= 1$ (es decir, $i = 4$, $k = 3$, $s = 1$ y $p = 0$). Es equivalente a la convolución de un kernel 3×3 sobre una entrada de 2×2 rellenada con un borde de ceros de 2×2 utilizando un stride $= 1$ (es decir, $i' = 2$, $k' = k$, $s' = 1$ y $p' = 2$).

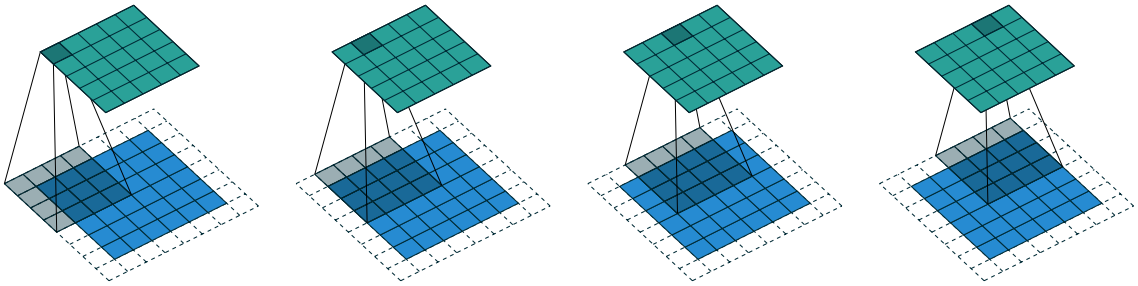


Figura B.9: Convolución transpuesta de un kernel 4×4 sobre una entrada de 5×5 rellenada con un borde de ceros de 2×2 utilizando un stride $= 1$ (es decir, $i = 5$, $k = 4$, $s = 1$ y $p = 2$). Es equivalente a la convolución de un kernel 4×4 sobre una entrada de 6×6 rellenada con un borde de ceros de 1×1 utilizando un stride $= 1$ (es decir, $i' = 6$, $k' = k$, $s' = 1$ y $p' = 1$).

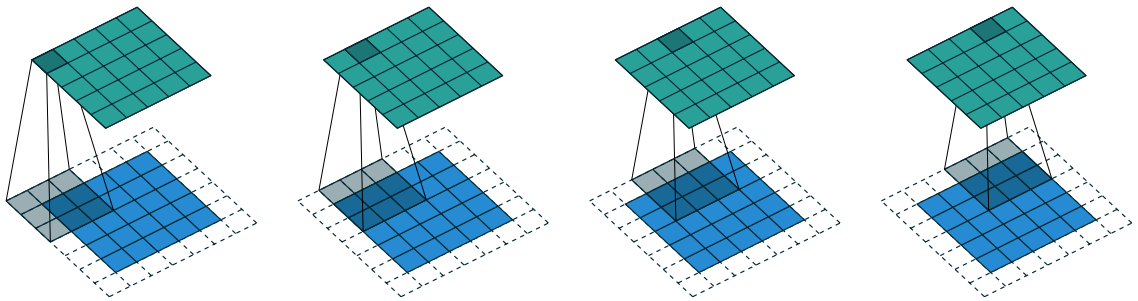


Figura B.10: Convolución transpuesta de un kernel 3×3 sobre una entrada de 5×5 utilizando relleno $= 1$ y un stride $= 1$ (es decir, $i = 5$, $k = 3$, $s = 1$ y $p = 1$). Es equivalente a la convolución de un kernel 3×3 sobre una entrada de 5×5 utilizando relleno $= 1$ y un stride $= 1$ (es decir, $i' = 5$, $k' = k$, $s' = 1$ y $p' = 1$).

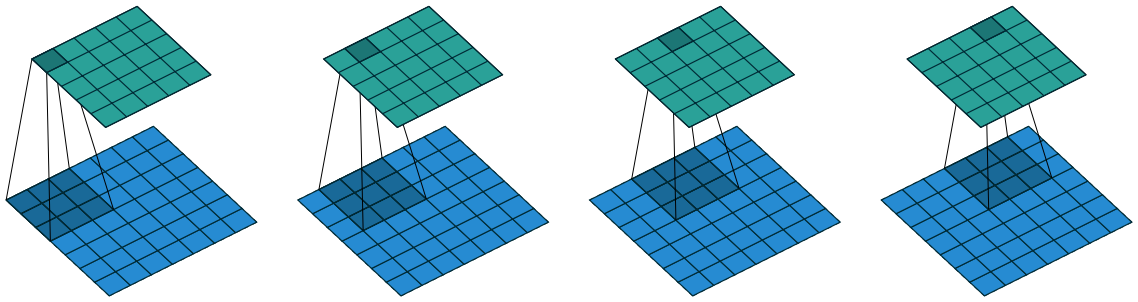


Figura B.11: Convolución transpuesta de un kernel 3×3 sobre una entrada de 5×5 utilizando relleno completo y stride $= 1$ (es decir, $i = 5$, $k = 3$, $s = 1$ y $p = 2$). Es equivalente a la convolución de un kernel 3×3 sobre una entrada de 7×7 utilizando un stride $= 1$ (es decir, $i' = 7$, $k' = k$, $s' = 1$ y $p' = 0$).

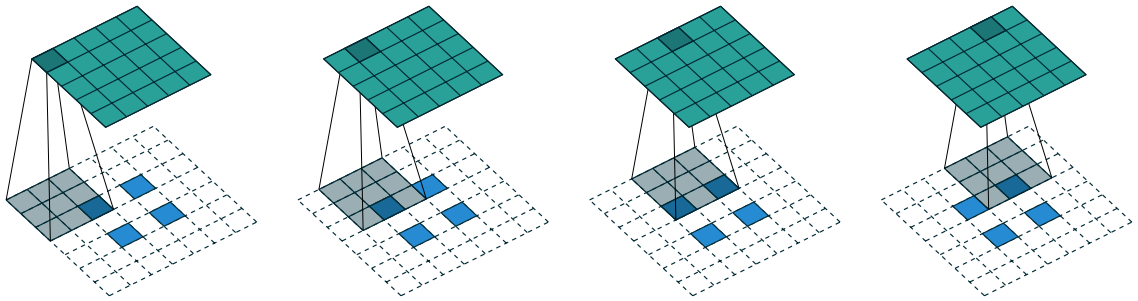


Figura B.12: Convolución transpuesta de un kernel 3×3 sobre una entrada de 5×5 utilizando stride de 2×2 (es decir, $i = 5$, $k = 3$, $s = 2$ y $p = 0$). Es equivalente a la convolución de un kernel 3×3 sobre una entrada de 2×2 (con 1 cero insertado entre las entradas) rellenada con un borde de ceros de 2×2 utilizando stride $= 1$ (es decir, $i' = 2$, $\tilde{i}' = 3$, $k' = k$, $s' = 1$ y $p' = 2$).

C. YOLO

YOLO, son las siglas de *You Only Look Once*, que el sistema SOTA de detección de objetos en tiempo real, utiliza redes CNN para su propósito. Actualmente, las últimas versiones, las implementa Ultralytics [122].

Su arquitectura inicial consta de 24 capas convolucionales, seguidas de dos totalmente conectadas.

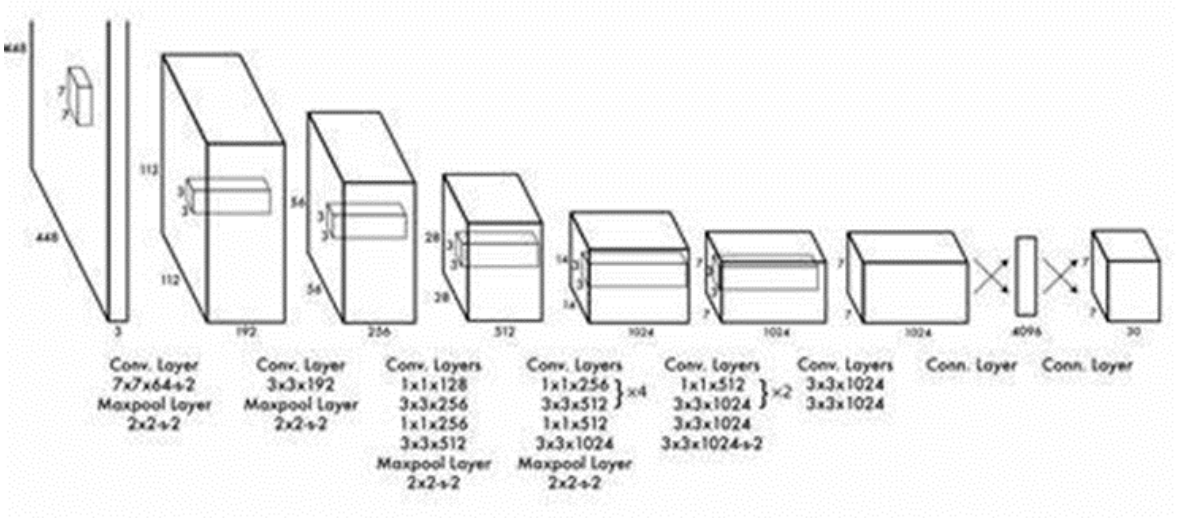


Figura C.1: Arquitectura de YOLOv1, extraída de [123]

C.1. Versiones

Originalmente, el algoritmo de YOLO era capaz de detectar 20 clases diferentes, repartidas, en animales, vehículos y otros objetos cotidianos. A lo largo de los años, se han ido desarrollando nuevas versiones de este algoritmo, con cada vez mejores resultados.

Para más información, consultar el paper *"A COMPREHENSIVE REVIEW OF YOLO: FROM YOLOV1 TO YOLOV8 AND YOLO-NAS"* [130]

De forma adicional, a principios de 2024, salió una nueva versión, YOLOv9, cuyas mejoras se obtienen al aplicar técnicas innovadoras como la Información de Gradiente Programable (PGI) y la Red de Agregación de Capas Eficiente Generalizada (GELAN). Según Ultralytics, "YOLOv9 representa un avance fundamental en la detección de objetos en tiempo real, ya que ofrece mejoras significativas en términos de eficacia, precisión y adaptabilidad."[131]

Tabla C.1

Comparativa de las versiones de YOLO.

Versión	**Año**	**Descripción**	**Por qué usar esta versión**
YOLOv1 [123]	2015	Versión inicial. Detección de objetos más veloz que predecesores. Aplicación red neuronal sobre imagen completa.	Mejora en velocidad de detección respecto a métodos anteriores.
YOLOv2 [124]	2017	Detecta más de 9000 clases de objetos.	Ampliación significativa en la cantidad de clases detectables.
YOLOv3 [125]	2018	Detección de objetos a distintas escalas (Tamaño: pequeño, mediano y grande).	Mejora en la detección de objetos de diferentes tamaños.
YOLOv4 [126]	2020	Gran salto de importancia. Aplicación de redes CNN. Extrae más características. Aumento de precisión de la red.	Avances significativos en precisión y extracción de características.
YOLOv5	2020	Mejoras de rendimiento de arquitectura de la red.	Optimizaciones en la arquitectura para un mejor rendimiento.
YOLOv6 [127]	2022	No se conocen en detalle las mejoras de esta red.	-
YOLOv7 [128]	2022	No se conocen en detalle las mejoras de esta red.	Superó a todos los detectores de objetos conocidos anteriormente en velocidad y precisión, pasando de 5 a 160 FPS
YOLOv8	2023	Vuelve a utilizar el dataset MS-COCO	Versión más estable y utilizada hasta el momento
YOLOv9 [129]	2024	Aplicación de técnicas innovadoras como PGI y GELAN.	Incorporación de técnicas de vanguardia para mejorar la detección.
YOLO-NAS	2023	Uso de arquitecturas de red neuronal generadas automáticamente mediante algoritmos de búsqueda de arquitectura neural (NAS). Optimización para mejorar la eficiencia y precisión. Capacidad de adaptarse mejor a diferentes tipos de datos y requisitos de detección.	Utilizar cuando se busca una arquitectura optimizada automáticamente para eficiencia y precisión.

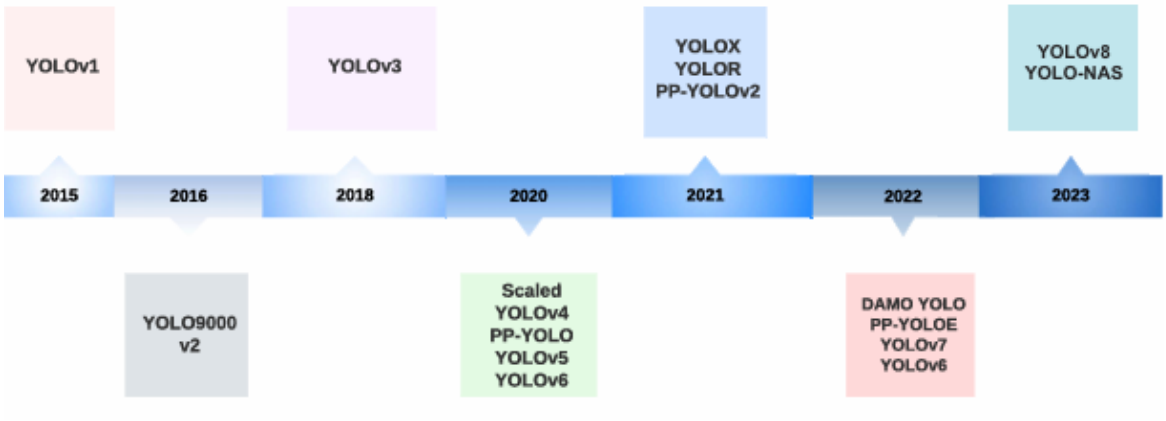


Figura C.2: Línea temporal de las distintas versiones de YOLO.

C.2. Utilidad en nuestro TFG

En este Trabajo de Fin de Grado, se ha empleado YOLO en su versión 8, (yolov8) para la detección [132] y segmentación (con yolo-seg [133]) de vehículos, para poder obtener de una imagen cualquiera un vehículo, seleccionarlo y segmentarlo. Pudiendo de esta manera obtener su máscara y pegarla encima de su imagen original, obteniendo así la imagen de entrada para nuestro modelo.

Figura C.3: YOLO Object Detection. Obtenido de [132]

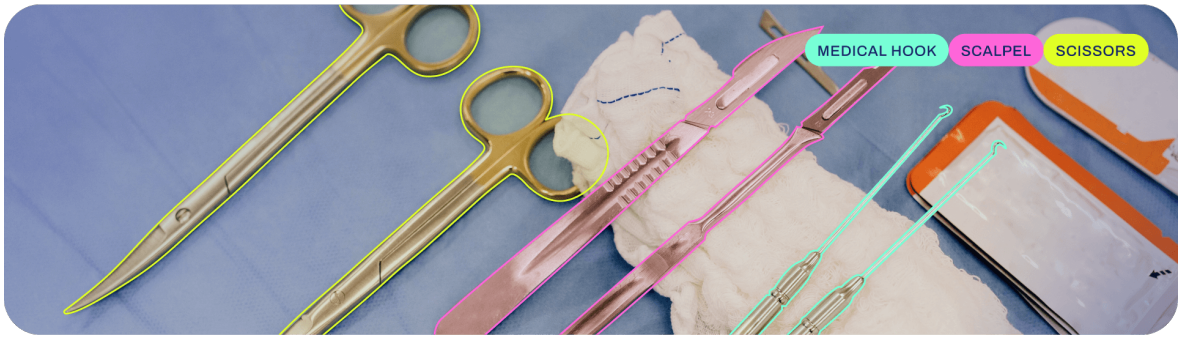


Figura C.4: YOLO Instance Segmentation. Obtenido de [133]

D. Arquitecturas de redes CNN más comunes.

A continuación se van a introducir a las redes CNN más conocidas y se va a mostrar sus estructuras. Los detalles de su arquitectura están expuestos más en detalle y se pueden consultar en CS231[38].

D.1. LeNet

La primera aplicación funcional de redes CNN fue desarrollada por Yann LeCun en 1993 [134], se puede ver en la figura D.2. Esta red utiliza la conocida arquitectura *LeNet*, que se usaba para leer códigos postales, dígitos, etc. Su arquitectura se muestra en la figura D.1

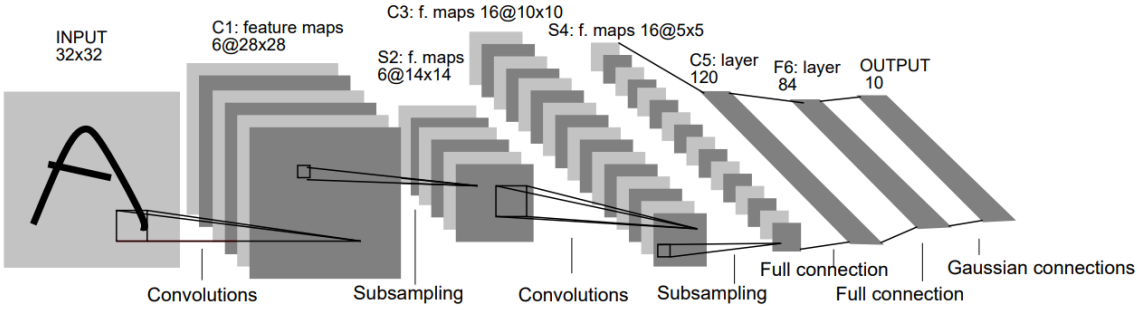


Figura D.1: Arquitectura LeNet

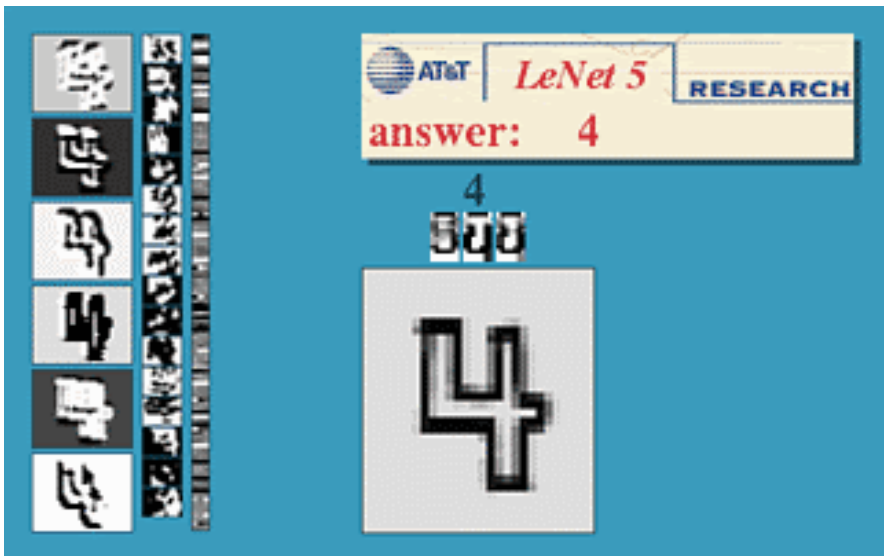


Figura D.2: Ejemplo real aplicación LeNet[135]

D.2. AlexNet

Desarrollado por Alex Krizhevsky, Ilya Sutskever y Geoff Hinton [136]. Fue el ganador del challenge *ILSVRC* en 2012, ganando ampliamente al segundo finalista. Su arquitectura es similar a *LeNet*, pero más grande y profunda, con varias convoluciones apiladas sin poolings entre medio.

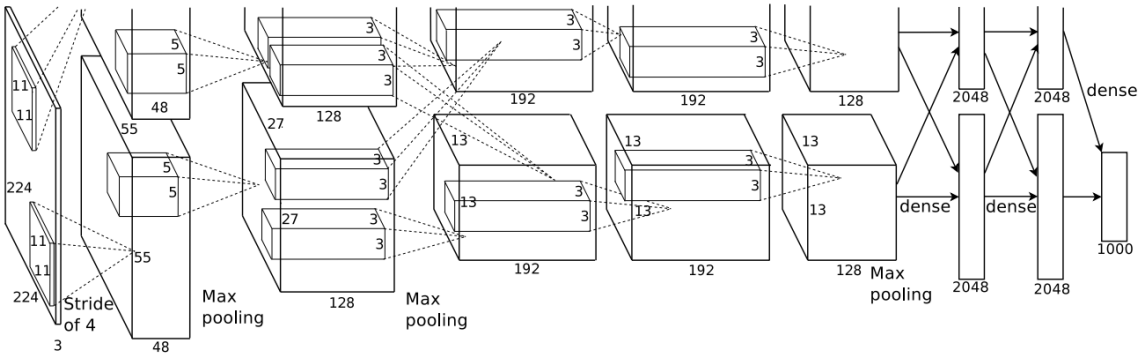


Figura D.3: Arquitectura AlexNet [136]

D.3. ZF Net

Ganador del *ILSVRC* de 2013, es una CNN de Matthew Zeiler y Rob Fergus[137]. Es una mejora de *AlexNet* lograda al reducir hiperparámetros gracias expandir las convoluciones intermedias y reducir el stride y tamaño del kernel en la primera capa.

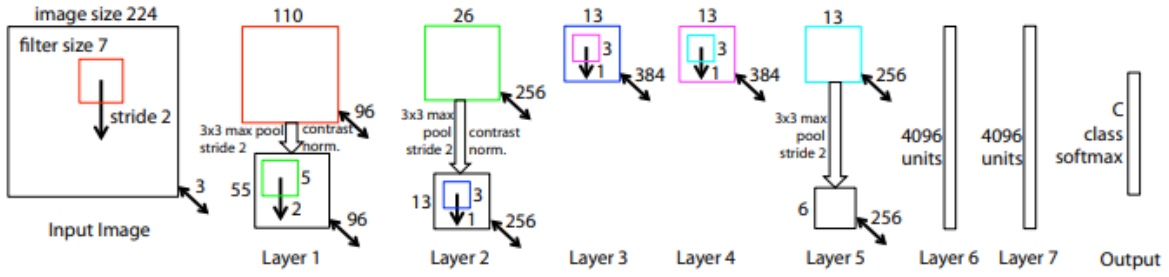


Figura D.4: Arquitectura de ZF Net [137]

D.4. GoogLeNet

Esta red de Szegedy *et al.*[138] de Google, fue la ganadora del challenge en 2014, redujo los 60 millones de parámetros de *AlexNet* a solo 4 millones gracias a un bloque conocido como *Inception Module*. Además, se usó el *Average Pooling*[42] (ver fig.D.6) en lugar de capas totalmente conectadas, de manera que se eliminó un gran número de parámetros también.

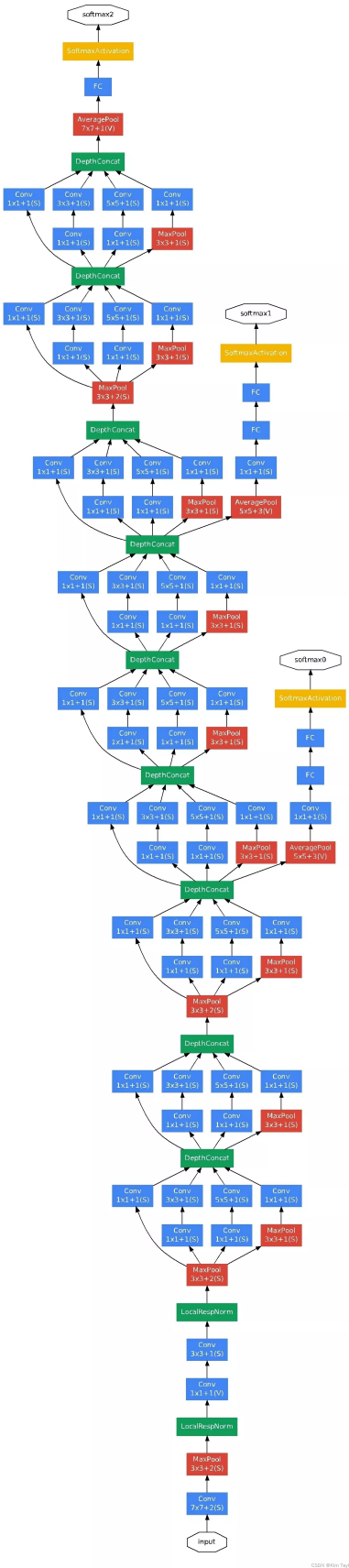
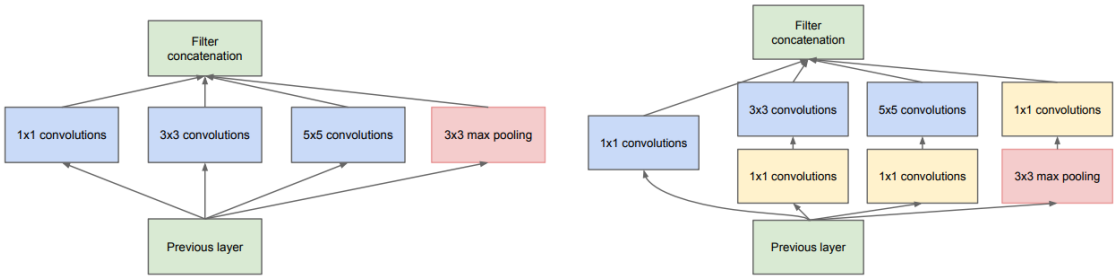


Figura D.5: Arquitectura GoogLeNet [138]



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figura D.6: Inception Module [138]

D.5. VGGNet

En el segundo puesto del challenge *ILSVRC* de 2014 quedo la red de Karen Simonyan y Andrew Zisserman, que se conoce como *VGGNet*[139]. Su mayor contribución fue la demostración de que la profundidad de la red, es un componente critico para obtener un buen rendimiento. Su red final, contenía 16 capas convolucionales/totalmente conectadas (CONV/FC) y utiliza una arquitectura extremadamente homogénea que solo utiliza convoluciones 3x3 y un pooling 2x2 de principio a fin.

Una desventaja de la red, es que es muy costosa de evaluar, ya que requiere de un montón de memoria y parámetros (140 millones) debido al número de capas totalmente conectadas.

A partir de entonces se descubrió que esas capas totalmente conectadas se pueden eliminar manteniendo el rendimiento, y de esta manera reduciendo de una forma considerable el número de parámetros totales.

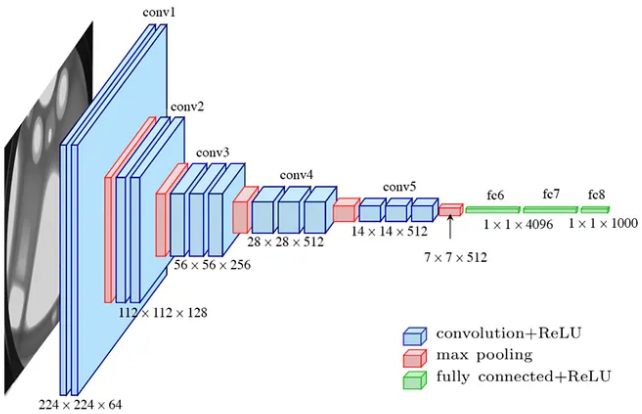


Figura D.7: Arquitectura VGGNet. Tomado de [140]

D.6. ResNet

Es la primera red residual, desarrollada por Kaiming He *et al.* [141], fue la ganadora del challenge *ILSVRC* de 2015. Su novedad fue la aplicación de skip-connections junto con un uso muy elevado de batch normalizations[142]. La arquitectura no tiene capas totalmente conectadas al final de la red, por lo que se reduce la complejidad del modelo y su número de parámetros.

El bloque residual propuesto evita en gran medida el problema de la evanescencia del gradiente, con lo que se logró mejorar el rendimiento de las CNNs. De hecho, esta red, ha sido hasta hace no mucho tiempo el SOTA en clasificación de imágenes con CNNs.

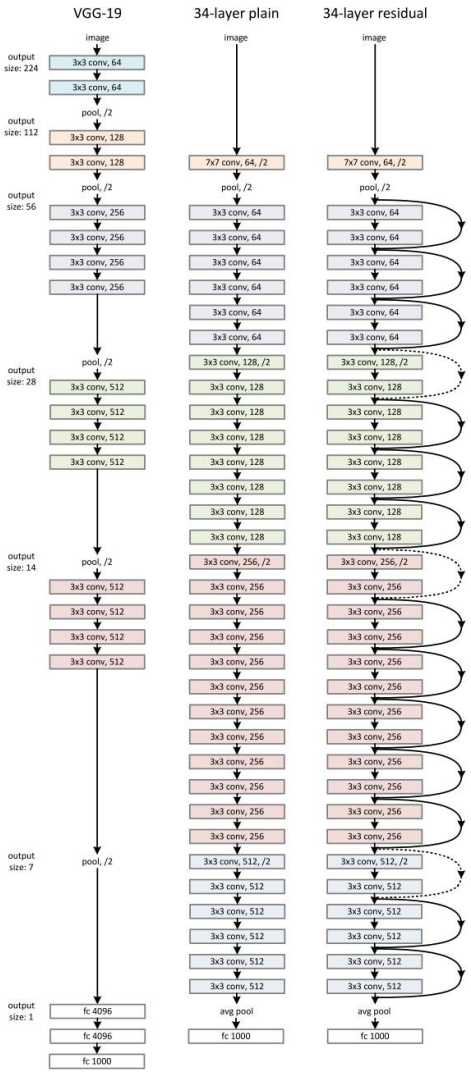


Figura D.8: Arquitectura ResNet[141]

E. Arquitecturas de redes de inpainting más conocidas.

En esta sección, nos basamos en el artículo *"10 Papers You Must Read for Deep Image Inpainting"*[143] para poder ver las redes más conocidas y utilizadas para inpainting a lo largo de los últimos años junto con su estructura.

E.1. Context Encoder

Red creada en el año 2016 por D.Pathak *et al.*[144], fue la primera red basada en GAN[69] para la restauración de imágenes mediante técnicas de inpainting. En ella se introducen conceptos básicos y una capa totalmente conectada por canales, para así poder entender el contexto de la imagen.

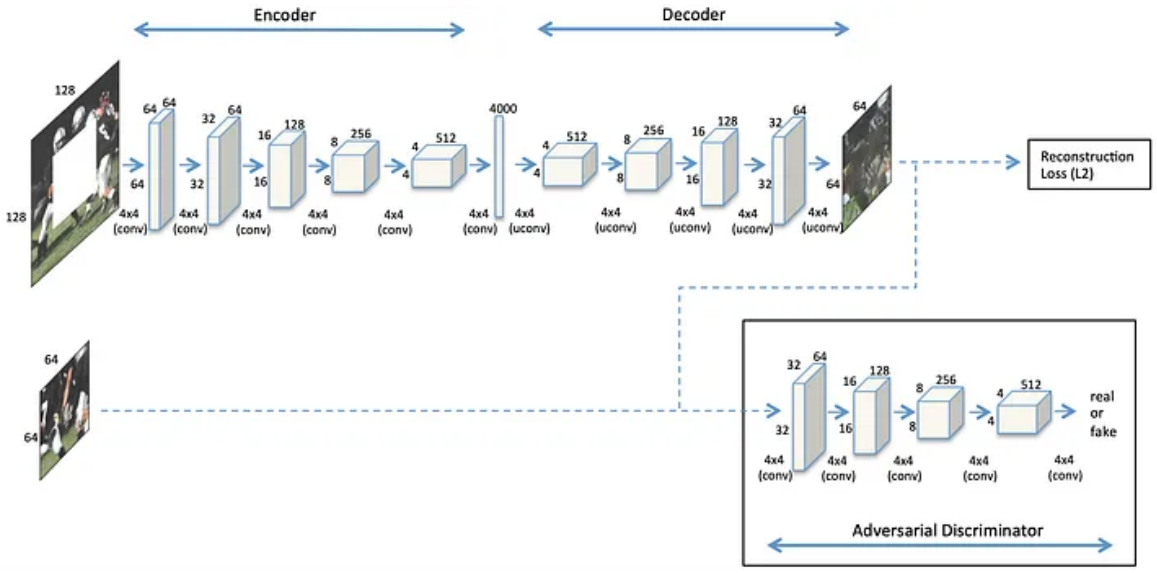


Figura E.1: Arquitectura de la red Context Encoder

E.2. MSNPS

Multi-Scale Neural Patch Synthesis [145] es una red creada en 2016 y se puede considerar como una versión mejorada de la *Context Encoder*[144]. Utiliza una red de textura, que logra mejorar los detalles visuales y también utiliza una perdida propuesta por ellos, la pérdida de textura, que se relaciona con la perdida perceptual y de estilo.

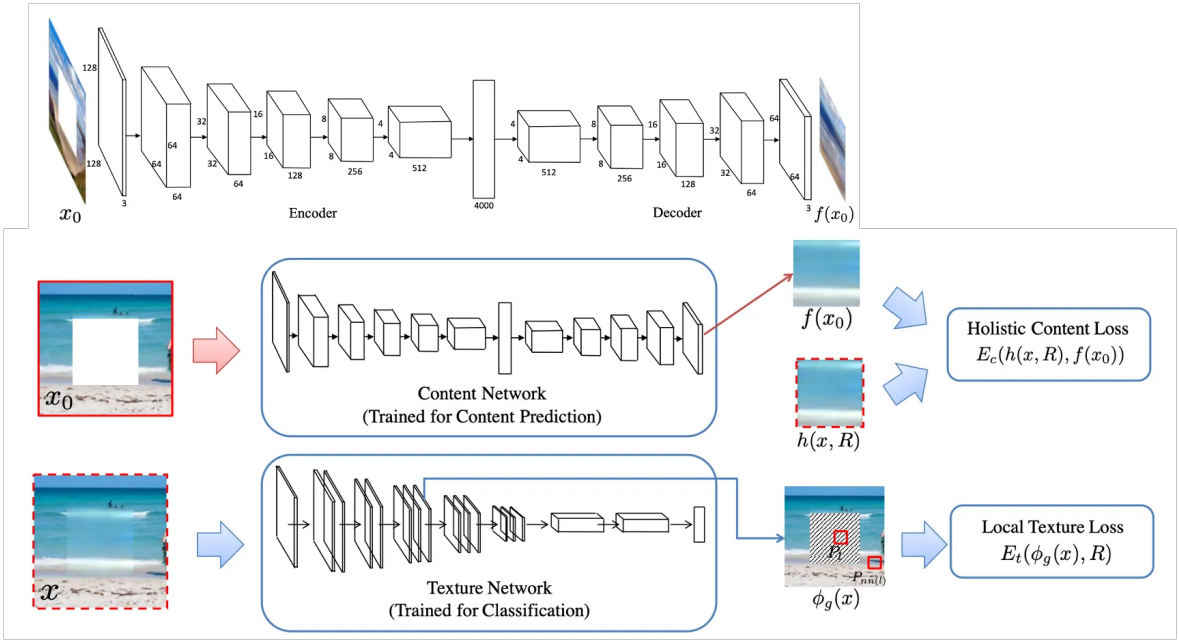


Figura E.2: Arquitectura de la red MSNPS

E.3. GLCIC

Globally and Locally Consistent Image Completion [146] es una red creada en 2017. Esta red es completamente convolucional, con dilataciones que sirven para entender el contexto evitando el uso de capas costosas y con la posibilidad de utilizar tamaños diferentes de imagen, también utiliza discriminadores tanto globales como locales entrenados a la par que el generador.

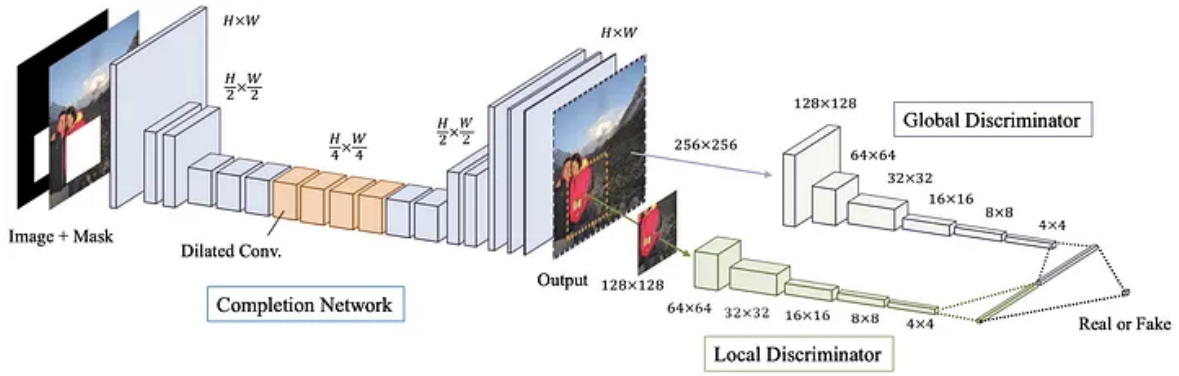


Figura E.3: Arquitectura de la red GLCIC

E.4. Patch-based Image Inpainting with GANs

La red propuesta en este paper [147] se creó en 2018, se puede considerar una variante de la red anterior (*GLCIC* [146]), su novedad es la incorporación de aprendizaje residual [141] y una red GAN conocida como *PatchGAN*[148], que sustituye la

de original de *GLCIC*, con la combinación de estas dos novedades logran mejorar la consistencia de la estructura global y los detalles en la textura de la imagen generada.

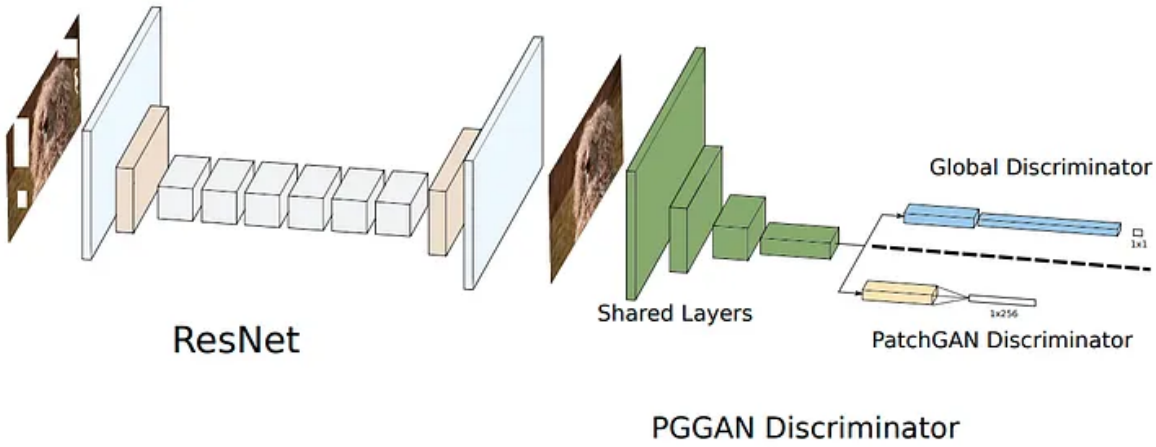


Figura E.4: Arquitectura de la red propuesta en el paper [146]

E.5. Shift-Net

Shift-Net[149], creada en 2018, combina redes CNN modernos y el método tradicional de "copy-paste", introduce una pérdida de guía que busca que las características decodificadas faltantes sean cercanas a las codificadas en buen estado. También añaden una capa de desplazamiento que permite tomar información de los vecinos más cercanos a la parte faltante de la imagen, para refinar los detalles en la zona generada.

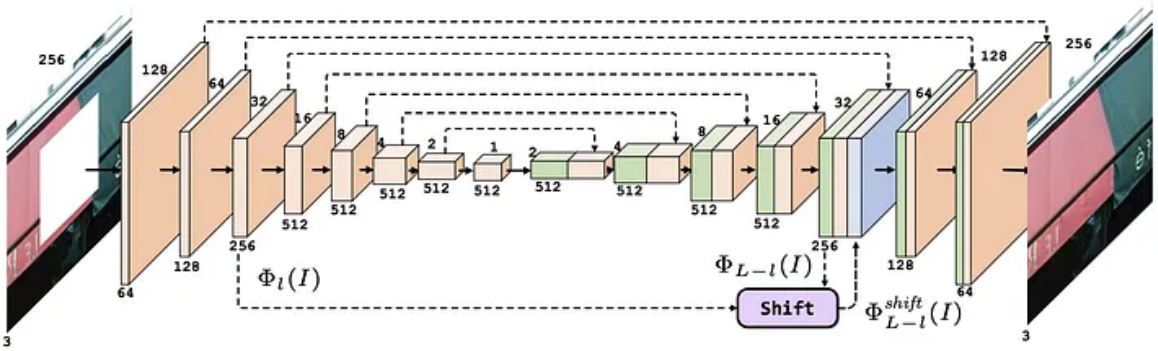


Figura E.5: Arquitectura de Shift-Net

E.6. DeepFill v1

DeepFill v1 (2018) [150], está considerada como una versión mejorada de *Shift-Net*[149]. En ella se introduce una capa de atención contextual que refina las características aún más en las áreas faltantes.

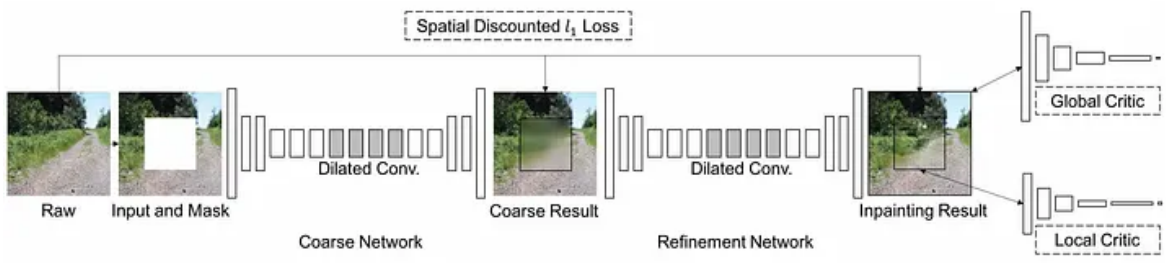


Figura E.6: Arquitectura de red DeepFill v1

E.7. GMCNN

Generative Multi-column Convolutional Neural Networks (2018) [151] tiene como característica especial el uso de varias ramas y tamaños de filtro distintos para mejorar las texturas. Asimismo, propone nuevas funciones de pérdida para esta tarea, a esta función de pérdida se la conoce como *"Implicit Diversified Marckov Random Field (ID-MRF)"*

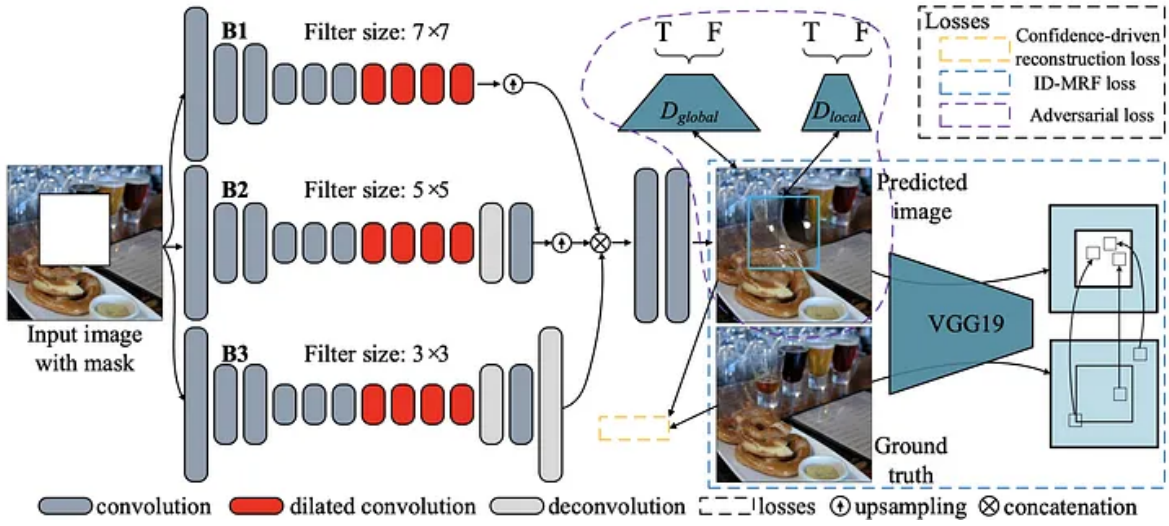


Figura E.7: Arquitectura red CMCNN

E.8. PartialConv

En el paper *"Image Inpainting for Irregular Holes using Partial Convolutions"*[152] de 2018 nos permite restaurar imágenes con máscaras irregulares utilizando convoluciones parciales.

Module Name	Filter Size	# Filters/Channels	Stride/Up Factor	BatchNorm	Nonlinearity
PConv1	7×7	64	2	-	ReLU
PConv2	5×5	128	2	Y	ReLU
PConv3	5×5	256	2	Y	ReLU
PConv4	3×3	512	2	Y	ReLU
PConv5	3×3	512	2	Y	ReLU
PConv6	3×3	512	2	Y	ReLU
PConv7	3×3	512	2	Y	ReLU
PConv8	3×3	512	2	Y	ReLU
NearestUpSample1		512	2	-	-
Concat1(w/ PConv7)		512+512		-	-
PConv9	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample2		512	2	-	-
Concat2(w/ PConv6)		512+512		-	-
PConv10	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample3		512	2	-	-
Concat3(w/ PConv5)		512+512		-	-
PConv11	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample4		512	2	-	-
Concat4(w/ PConv4)		512+512		-	-
PConv12	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample5		512	2	-	-
Concat5(w/ PConv3)		512+256		-	-
PConv13	3×3	256	1	Y	LeakyReLU(0.2)
NearestUpSample6		256	2	-	-
Concat6(w/ PConv2)		256+128		-	-
PConv14	3×3	128	1	Y	LeakyReLU(0.2)
NearestUpSample7		128	2	-	-
Concat7(w/ PConv1)		128+64		-	-
PConv15	3×3	64	1	Y	LeakyReLU(0.2)
NearestUpSample8		64	2	-	-
Concat8(w/ Input)		64+3		-	-
PConv16	3×3	3	1	-	-

Figura E.8: Arquitectura de PartialConv

E.9. EdgeConnect

EdgeConnect: Generative Image Inpainting with Adversarial Edge Learning (2019) [153], se centra en separar el inpainting en dos pasos. Primero, trata de predecir los bordes de las regiones faltantes, y segundo, tratar de completar la imagen de acuerdo a los bordes predichos anteriormente.

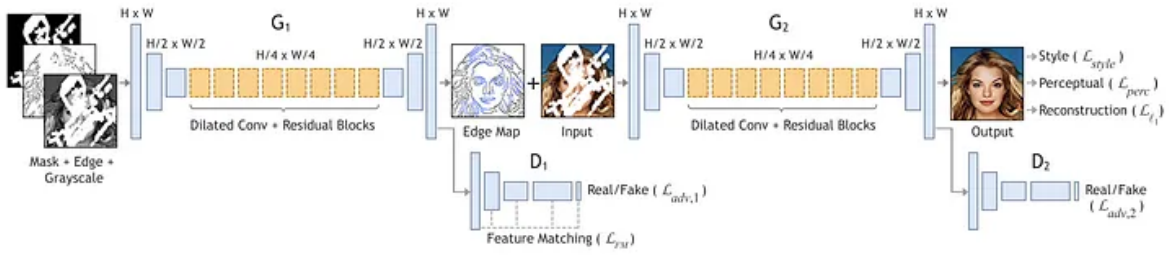


Figura E.9: Arquitectura de EdgeConnect

E.10. DeepFill v2

Free-Form Image Inpainting with Gated Convolution (2019) [154] es uno de los mejores algoritmos que se pueden usar actualmente. Se le puede considerar como una

versión mejorada de *DeepFill v1*[150], *PartialConv*[152] y *EdgeConnect*[153]. En esta estructura, se aplica una combinación de Convoluciones Parciales y una versión aprendible de la parcial, conocida como "*Gated Convolution*". Por último, utiliza *SN-PatchGan* [154] para estabilizar el entrenamiento de la red GAN.

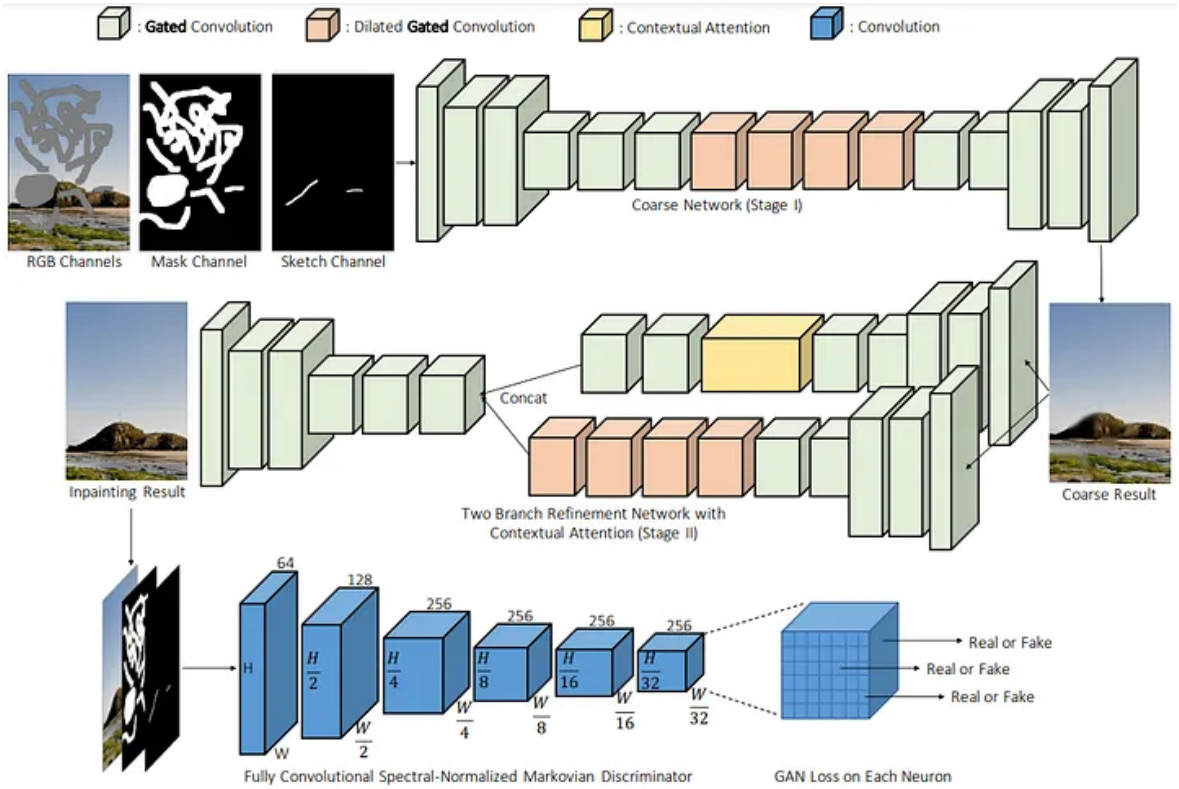


Figura E.10: Arquitectura DeepFill V2

Esta recopilación habla de redes desde 2016 hasta 2020, pese a ello se ha tratado sobre el SOTA de inpainting para distintos datasets en la sección 3.4