Escuela Técnica Superior de
Ingeniería Informática

TRABAJO FIN DE GRADO

Detección Visual de Anomalías para Robots Móviles

Realizado por
Adrián Candil Salas

Para la obtención del título de
Graduado en Ingeniería Informática - Tecnologías Informáticas

Dirigido por
Rocío González Díaz
Javier Perera Lago

En el Departamento de
Matemática Aplicada I

Convocatoria de julio, curso 2024

En primer lugar me gustaría no solo dedicar este trabajo, sino todo mi recorrido académico a mis padres Jaime y Luz María. Siempre han sido y serán un pilar fundamental en mi vida y sin ellos no habría llegado hasta aquí.

Tanbién me gustaría dedicarselo a mi hermmana Lidia , para la cual me esfuerzo por ser un ejemplo a seguir.

Así mismo, quisiera también que esta dedicatoria sea dirigida hacia el resto de mi familia y a mis amigos que son para mí como hermanos. Especialmente a Nacho, Enrique y Juan Pablo los cuales han sido parte de este camino a diario durante mi carrera universitaria y han vivido tanto los buenos como los malos momentos conmigo y a Cristina, que a pesar de no ser nuestra compañera de piso ha vivido mucho con nosotros. Además deseo mencionar a mi compañero Jose con en cual he luchado muchas batallas en la facultad.

Finalmente agradecer a mi pareja Aida, la cual me ha dado el apoyo que necesitaba para no tropezar justo al final del camino.

Espero que ser parte de esta dedicatoria refleje el cariño y el agradecimiento que siento por todos los que me habéis acompañado. Sin vosotros no habría llegado aquí.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas aquellas personas que contribuyeron de manera significativa al desarrollo de este trabajo.

En primer lugar, deseo agradecer a mis tutores Rocío González Díaz y Javier Perera Lago por su orientación y dedicación a lo largo del proyecto. Sus conocimientos y consejos fueron fundamentales para el desarrollo del mismo.

Una vez más quiero agradecer a mi familia y amigos, los cuales me han brindado siempre su cariño y apoyo. Para mí han sido y serán siempre cruciales.

Por último quiero reconocer a mis compañeros y profesores, quienes compartiendo conmigo sus conocimientos han contribuido a mi desarrollo académico.

A todos a los que habéis estado ahí siempre, de corazón os estoy eternamente agradecido.

Resumen

En este trabajo se realiza un estudio sobre una arquitectura diseñada para el reconocimiento de anomalías mediante inteligencia artificial en las imágenes que toman las cámaras de robots móviles presentada en el artículo *An Outlier Exposure Approach to Improve Visual Anomaly Detection Performance for Mobile Robots* [1].

Este proyecto presenta el desarrollo del estudio de esta arquitectura introduciendo previamente los conceptos necesarios para su entendimiento. He realizado el diseño y la implementación de varias de las arquitecturas propuestas, realizando modificaciones a estas y probando su rendimiento. El código se encuentra en el siguiente GitHub.

Abstract

This work presents a study on an architecture designed for anomaly detection using artificial intelligence in images captured by mobile robot cameras, as presented in the article *An Outlier Exposure Approach to Improve Visual Anomaly Detection Performance for Mobile Robots* [1].

This project involves the development of the study of this architecture, introducing the necessary concepts for its understanding. I have designed and implemented several of the proposed architectures, making modifications to them and testing their performance. The code can be found at the following GitHub.

Índice general

Índice de figuras

Índice de tablas

Índice de extractos de código

1. Introducción

Día tras día, la inteligencia artificial (IA) está tomando un rol cada vez más importante dentro de la sociedad, extendiéndose en todos los sectores. Desde *chatboxes* como chat-GPT o IAs que generan imágenes hasta el desarrollo de coches o robots autónomos. Los robots autónomos son máquinas capaces de moverse por un entorno para realizar diversas tareas como exploración, manipulación o transporte de objetos. Durante su desplazamiento por un entorno en constante cambio pueden aparecer elementos en el camino del robot que supongan un peligro para él. Por tanto es necesario que este sea capaz de detectar estos elementos y evitarlos.

Para ello en este trabajo vamos a centrarnos en cubrir esta necesidad: ¿Cómo puede un robot equipado con una cámara detectar elementos anómalos?

Para responder a esta respuesta centraremos nuestra atención en la arquitectura propuesta por el grupo de investigadores formado por Dario Mantegazza , Alessandro Giusti , Luca Maria Gambardella, y Jérôme Guzzi [1]. Esta consiste en primero reducir la dimensión de la imagen tomada por el robot y posteriormente aplicar diferentes modelos de clasificación sobre esta reducción para poder decidir si la imagen contiene o no una anomalía.

Para el estudio de esta arquitectura, además de basarnos en el artículo que la desarrolla, se han revisado numerosas fuentes bibliográficas y sobre todo se ha tratado de simular los experimentos llevados a cabo por el equipo mencionado previamente. Este código queda disponible en mi repositorio de github.

Este trabajo se divide en varias partes. La sección 2, que contempla la metodología seguida durante el desarrollo del trabajo. A continuación encontramos la sección 3, la cual describe la planificación inicial del proyecto así como el desarrollo real de este. Después, en la sección 4, hablamos de los conceptos previos necesarios para el entendimiento de la arquitectura desarrollada. La sección 5 explica la arquitectura propuesta por el artículo de estudio [1] y explica la implementación que he realizado. La sección 6 es un breve resumen del contenido del github en el que se encuentra el código del proyecto. Finalmente la sección 7 resume los resultados obtenidos con este proyecto.

2. Metodología

La realización de este trabajo ha constado de un trabajo paralelo de desarrollo de código y de investigación.

Comencé estudiando el artículo *An Outlier Exposure Approach to Improve Visual Anomaly Detection Performance for Mobile Robots* [1] de los investigadores mencionados previamente en la introducción. Una vez tuve un nivel de comprensión suficiente, comencé a desarrollar el código. El desarrollo del código ha sido la parte a la que mas tiempo he dedicado como veremos mas adelante en la sección 3. Este desarrollo se basa en la traducción del código aportado en el artículo mencionado previamente, que está en PyTorch [9], a Tensorflow [10] en un notebook de GoogleColab [11]. Pytorch y Tensorflow son las librerías mas desarrolladas para la implementación de modelos de aprendizaje profundo. Decidí llevar a cabo esta traducción debido a que en las asignaturas de inteligencia artificial usamos Tensorflow, por tanto al ser una arquitectura que ya conocía, facilitaría el desarrollo del trabajo al mismo tiempo que añadiría algo más que simplemente reproducir el código en Pytorch. Por otra parte GoogleColab es una herramienta que ofrece Google para trabajar con notebooks de Python. Debido a las limitaciones de GoogleColab, he tenido que buscar una gran cantidad de soluciones, que veremos durante el documento, para poder llevar a cabo los experimentos.

Como gran parte de los experimentos han sido largos entrenamientos de modelos de IA, durante la supervisión de los entrenamientos realicé gran parte de la revisión bibliográfica del trabajo, necesaria para el desarrollo del código.

Este trabajo es el resultado de un proceso continuo de aprendizaje y supervisión, habiendo realizado reuniones cada dos semanas con Rocío González Díaz y Javier Perera Lago, tutores de este TFG.

3. Planificación temporal y coste

En esta sección vamos a ver la planificación del proyecto así como el coste estimado si fuese un proyecto real. En la planificación expongo las actividades realizadas, los tiempos estimados de estas y el tiempo real que hemos dedicado a cada actividad. La planificación ha ido variando durante el proyecto como veremos a continuación.

Por otra parte encontramos el análisis del coste que supondría este proyecto si fuese un proyecto financiado. Además en esta sección expondré el coste real del proyecto, ya que he tenido que adquirir ciertos servicios para su desarrollo.

3.1. Análisis temporal

La idea inicial era realizar la entrega del proyecto en la primera convocatoria. Desafortunadamente debido a la imposibilidad de hacerlo, finalmente la entrega se realizó en la segunda convocatoria un mes después. Este proyecto se divide en cuatro fases:

1. Iniciación
2. Planificación
3. Ejecución
4. Seguimiento y Control

La fase de iniciación es la fase inicial del proyecto donde se identifican las necesidades y se evalúa la viabilidad del este. Durante la fase de planificación se establece un plan concreto para llevar acabo el desarrollo del proyecto. La fase de ejecución consiste en llevar a cabo el plan establecido en la planificación. Finalmente, el seguimiento y control consiste en la monitorización de la ejecución, asegurando que se está llevando a cabo de forma correcta.

La siguiente tabla muestra el desarrollo de las siguientes fases.

	Iniciación	**Planificación**	**Ejecución**	**Seguimiento y Control**
Inicio	13/09/2023	03/10/2023	06/11/2023	03/10/2023
Fin estimado	03/10/2023	10/10/2023	23/05/2024	17/05/2024
Fin real	03/10/2023	17/10/2023	18/06/2024	18/06/2024

Tabla 3.1: Tiempos estimados y reales

Las actividades realizadas a lo largo del proyecto han sido las siguientes: planificación del trabajo, reuniones de seguimiento, desarrollo de código, estudio de la bibliografía, presentación, desarrollo de la memoria.

A lo largo del desarrollo del proyecto he tenido que dedicar una considerable cantidad de tiempo al estudio de la bibliografía y al desarrollo de código, aunque como ya mencioné en la metodología, parte de este tiempo de desarrollo fue supervisar que los entrenamientos fuesen como se esperaba y mientras tanto llevaba acabo el estudio bubliográfico para implementar diferentes modificaciones y técnicas que podía aplicar para mejorar la arquitectura. Las horas reales dedicadas al proyecto serían las siguientes:

Tarea	Horas estimadas	Horas reales
Planificación	6:00h	6:00h
Reuniones	7:00h	8:00h
Desarrollo del código	150:00h	203:00h
Estudio de la bibliografía	100:00h	55:00h
Desarrollo de la presentación	7:00h	6:30h
Desarrollo de la memoria	30:00h	40:00h
Total	300:00h	318:30h

Tabla 3.2: Tiempo estimado vs tiempo real

3.2. Coste

Para el análisis del coste que supondría el proyecto consideraremos tanto el pago al personal, como los gastos de servicios externos.

Comenzamos pues con el coste de personal. Vamos a asumir que el alumno ya es considerado un ingeniero informático. Consideremos también que estará cobrando el sueldo medio de un informático. Encontramos este dato en Talent.com.

Rol	Sueldo	Horas	Coste
Ingeniero informático	13.85€/h	318:30h	4411.25€

Tabla 3.3: Coste de personal

Vamos a considerar ahora los servicios utilizados que aparecen en la tabla 3.4. Consideramos, en primer lugar los servicios básicos que son la luz consumida y el acceso a internet. Por otra parte se ha tenido que contratar varios servicios de Google: Google One, Unidades de ejecución en GoogleColab y GoogleColabPro. Los servicios de GoogleOne nos proporcionan un mayor almacenamiento en GoogleDrive, lo cual nos dará la capacidad de almacenar los datasets ahí. Por su parte GoogleColabPro nos dará máquinas más potentes que habilitarán la ejecución del código. Para poder usar estas máquinas, GoogleColab consume unidades de ejecución. Dependiendo de cuantos recursos consumas se restará un número aproximado de unidades de ejecución. Aproximadamente unas 4 por hora. Cada adquisición me proporciona 100 unidades y otras 100 al mes con la suscripción a GoogleColabPro. Los gastos de internet y luz corresponden al pago mensual que realizo.

Servicio	Precio	Meses	Total
Luz	20€/mes	8	160€
Internet	11€/mes	8	88€
Google One	1.99€/mes	3	5.97€
Unidades de ejecución	11.99€/adquisición	3	35.97€
GoogleColabPro	11.99€/mes	2	23.98€
Total			313.92€

Tabla 3.4: Coste de servicios

El coste total del proyecto será por tanto la suma del coste de personal y de servicios.

-	Coste
Coste de personal	4411.25€
Coste de servicios	313.92€
Total	4713.18€

Tabla 3.5: Coste total

4. Conceptos previos

Para poder comprender en su totalidad la arquitectura trabajada a lo largo de este proyecto, primero debemos explicar una serie de conceptos. Concretamente, realizaremos una breve introducción a las redes neuronales, definiremos el perceptrón multicapa, describiremos la operación de convolución, estableceremos qué es una red neuronal convolucional y especificaremos concretamente cómo está estructurado un autoencoder y finalmente hablaremos del entrenamiento de modelos y de las diferentes técnicas utilizadas para evitar el sobreajuste.

4.1. Red neuronal

Una red neuronal es una estructura matemática inspirada en el sistema nervioso. Funciona utilizando un elevado número de unidades de procesamiento que tratan de simular a las neuronas. Estas neuronas se interconectan formando la estructura de la red. [6, página 58]

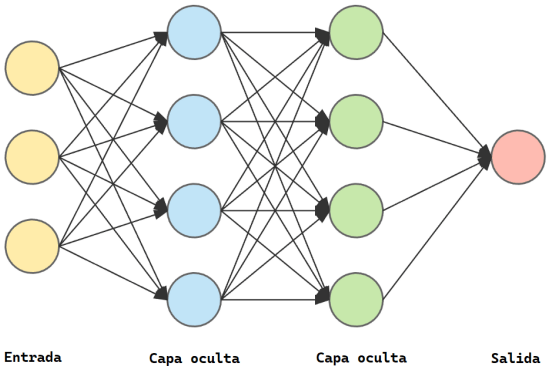

Figura 4.1: Red neuronal, [2]

La estructura de una red neuronal, como podemos observar en la Figura 4.1, consta de una capa de entrada, la cual recibe directamente la información que vamos a procesar; capas ocultas, las cuales contienen las neuronas cuya distribución e interconexión determina la topología de la red y finalmente, una capa de salida que, dependiendo de la función de nuestra red, proporcionará un tipo de información u otra (p.e: una clasificación, una imagen modificada, etc.)

En cada neurona se realiza la siguiente operación llamada sinapsis:

$$a_j = g\left(\sum_{i=0}^{n} w_{ij} a_i\right)$$

donde a_j es la activación de la neurona j, g es la función de activación común en toda la capa, w_{ij} es el peso asociado a la conexión entre la neurona i de la capa anterior y la neurona j, y a_i son las activaciones provenientes de las neuronas en la capa anterior.

La función de activación es una función que se aplica al resultado del cómputo de la sinapsis para introducir la no linealidad en el modelo, permitiendo que este aprenda y represente relaciones complejas de datos.

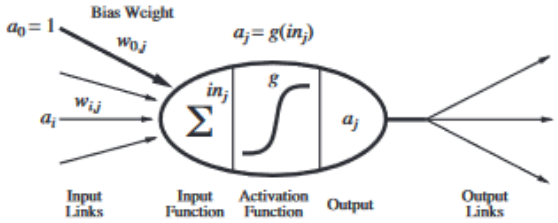

Figura 4.2: Neurona, [3]

4.1.1. Perceptrón multicapa

Un perceptrón multicapa es la forma más simple de red neuronal y la base de los modelos más avanzados desarrollados en el aprendizaje profundo. Normalmente, se utilizan en problemas de clasificación donde es necesario asignar etiquetas a observaciones (binarias o multinomiales) basadas en los datos de entrada.

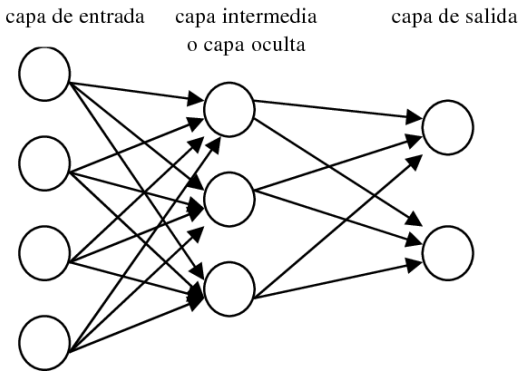

Figura 4.3: Perceptrón multicapa, [4]

4.2. Convolución

Una convolución es una operación matemática que combina dos funciones para producir una tercera función. Esta operación se utiliza para determinar cómo la forma de una función se modifica por otra. Concretamente, nos interesa enfocar el análisis de la operación desde un punto de vista matricial [6, página 122].

En el tratamiento de imágenes, estas son tratadas como matrices. Vamos a centrarnos por tanto en la convolución matricial, operación que, combinada con una red neuronal da lugar a una red neuronal convolucional [6, página 36].

La operación de convolución matricial se basa en hacer un barrido de la imagen con un kernel o filtro dando como reultado otra matriz del mismo tamaño o de tamaño reducido llamada mapa de características [6, página 122-127].

Veamos cómo se aplica la operación con un ejemplo. Tenemos una imagen de entrada 5×5 y un filtro 3×3. En este caso concreto el mapa de características tendrá un tamaño reducido tomando como referencia la imagen de entrada.

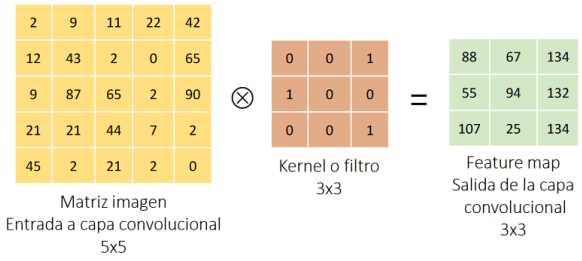

Figura 4.4: Operación de convolución, [5]

El mapa de características se obtiene paso a paso de la siguiente forma definida en la figura 4.5. El kernel se va desplazando por la imagen y va realizando una multiplicación elemento a elemento con la submatriz que superpone y suma los resultados para ir obteniendo cada uno de los valores del mapa de características.

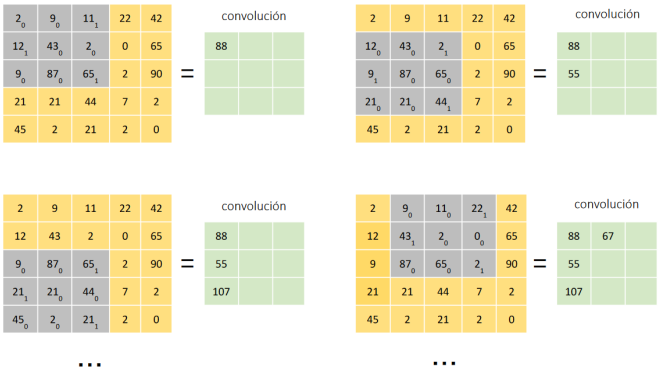

Figura 4.5: Convolución paso a paso, [5]

En la primera multiplicación de la figura 4.5, la operación sería la siguiente:

$$2 \cdot 0 + 9 \cdot 0 + 11 \cdot 1 + 12 \cdot 1 + 43 \cdot 0 + 2 \cdot 0 + 9 \cdot 0 + 87 \cdot 0 + 65 \cdot 1 = 88$$

El caso que acabamos de ver es válido para imágenes en blanco y negro que están formadas por un solo canal. Llamamos canal en este contexto a cada una de las dimensiones que contiene información relevante de un dato de entrada. Sin embargo esta operación puede realizarse también en imágenes a color que están formadas por 3 canales. Esto es debido a que las imágenes en color están formadas por los canales RGB. El modelo RGB es un sistema de representación basado en la combinación de los colores rojo, verde y azul. Al representar una imagen de forma matricial, podemos considerar la imagen como el tensor formado por las matrices correspondientes a cada uno de los canales del RGB. De esta forma una imagen será un tensor 3D y por tanto el kernel deberá tener también tres canales.

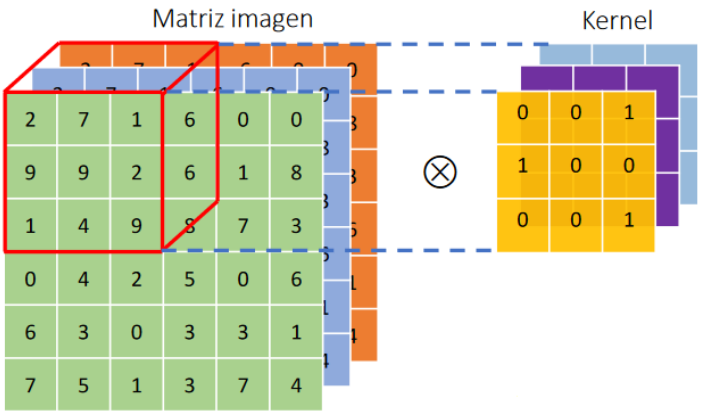

Figura 4.6: Convolución 3D, [5]

El mapa de características se obtendría en este caso de la siguiente manera definida en la figura4.7:

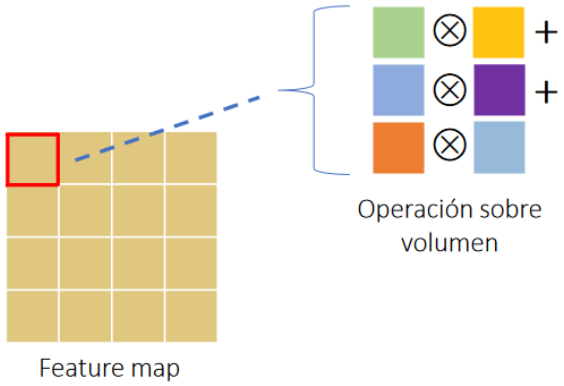

Figura 4.7: mapa de características 3D, [5]

De esta forma por cada kernel extrae un mapa de características unidimensional realizando la suma de la convolución de cada canal.

4.3. Red neuronal convolucional

Mezclando las redes neuronales y la convolución obtenemos uno de los instrumentos más populares de los últimos tiempos: la red neuronal convolucional [6, página 30].

Estas redes son redes neuronales que en vez de aprender los pesos que interconectan a las neuronas, la red aprenderá los pesos de los k filtros aplicados a la entrada de la capa, generando los k mapas de características correspondientes y aplicando la función de activación correspondiente a la capa para dar el output. Todo este proceso conforma una capa convolucional.[6, página 120-177]

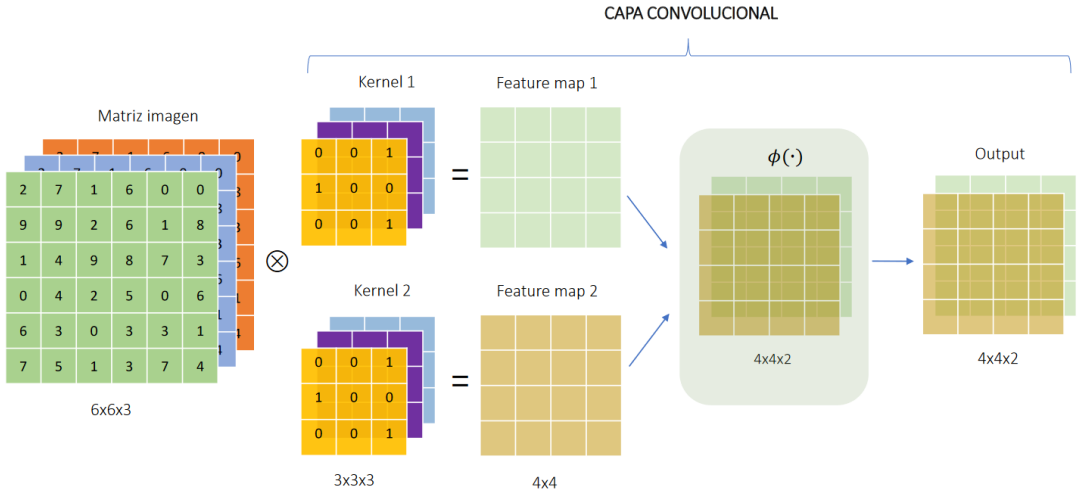

Figura 4.8: Capa convolucional, [5]

La sucesión de diferentes capas convolucionales, distribuidas de diferentes maneras dependiendo de la funcionalidad que queramos que tenga la red resultará en una red neuronal convolucional.

Normalmente se concatenan varias capas convolucionales para reducir la dimensión y aprender las características de una imagen y se concatena a un perceptrón multicapa que es el que realiza la clasificación. Un ejemplo de esto es el siguiente 4.9, el cual encontramos en [6, página 153]:

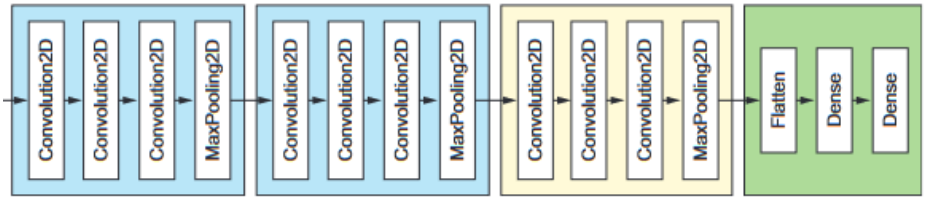

Figura 4.9: Red convolucional + red neuronal, [6]

Concretamente para construir nuestro modelo como veremos más adelante hemos utilizado capas Conv2D y Conv2DTranspose y en la versión inicial de nuestro modelo 5.4, MaxPooling2D y Upsample2D.

Conv2D realiza la operación de convolución descrita hasta el momento. Por otra parte las capas Conv2DTranspose, realiza la operación llamada convolución transpuesta o deconvolución. Esta operación consiste en aumentar la dimensión espacial y reconsrtuir la imagen. Teniendo en cuenta que Conv2D reduce la dimensión y extrae características y Conv2DTranspose aumenta la dimensión y reconstruye la imagen, podemos considerarlas operaciones opuestas.

Vamos a definir brevemente la operación de MaxPooling2D y Umsample2D que utilizamos en versiones iniciales del código, concretamente en la versión 1 del autoencoder5.4.

La operación de MaxPooling consiste en resumir las características barriendo con un kernel los mapas de características resultantes de una convolución aplicando la función máximo.

Por otro lado la operación de upsample es la opuesta a la de pooling, aumentando en el proceso la dimensión espacial de un mapa de características más pequeño mediante interpolación.

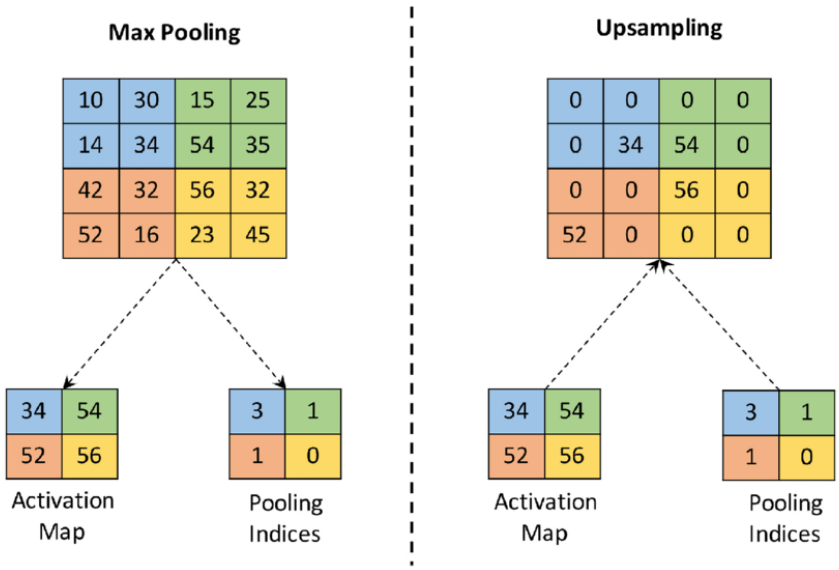

Figura 4.10: Pooling vs Upsample [7]

En nuestro estudio, vamos a profundizar concretamente en la estructura del autoencoder.

4.4. Autoencoder

Un autoencoder es una red neuronal convolucional formada por 3 bloques: el codificador, el cuello de botella y el decodificador. El objetivo del autoencoder es reducir la dimensión de la entrada y luego reconstruirla minimizando el error cometido durante el proceso.[12, página 1] Esta técnica se usa por ejemplo con el fin de reducir el peso de almacenamiento de los datos de entrada o como en nuestro caso, para preprocesar estos datos y dárselos como entrada a otros modelos de entrenamiento.

El codificador recibe un dato de entrada, para nuestro caso será una imagen, y reducirá su dimensión. Nuestro codificador recibirá imágenes $64 \times 64 \times 3$ y las reduce a embeddings unidimensionales de tamaño 128.

El decodificador se encarga de reconstruir la codificación producida por el encoder. En nuestro caso reconstruirá el embedding 128 a una imagen $64 \times 64 \times 3$ de nuevo.

El cuello de botella es la unión entre el codificador y el decodificador y donde encontramos los elementos codificados.

Para visualizar mejor esta red neuronal podemos imaginarnos una pajarita ya que su forma es idéntica a estas. En 4.11 podemos apreciar la forma mencionada.

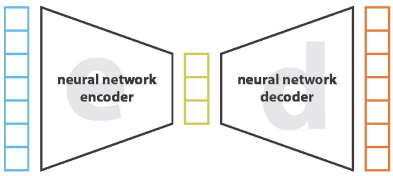

Figura 4.11: Autoencoder, [8]

4.5. Entrenamiento de una red

Como hemos mencionado previamente, las diferentes redes tienen que ser entrenadas sobre un conjunto de datos concreto para que se ajusten a una tarea. En nuestro caso, trabajamos con imágenes así que la arquitectura entrenará sobre un conjunto de imágenes.

La idea es que las imágenes se introduzcan en el autoencoder y este aprenda a reconstruirlas reduciendo el valor de la función de pérdida. Esta función trata de cuantificar la diferencia de los resultados que da la red y los valores reales de los datos de entrenamiento. Dependiendo del propósito de la red se usarán diferentes

funciones para esta finalidad. En nuestro caso usaremos el error cuadrático medio como función de pérdida durante el entrenamiento de un autoencoder y la entropía binaria cuadrada para el entrenamiento de un perceptrón multicapa.

$$\text{Error cuadrático medio} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\text{Entropía binaria cruzada} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

En estas funciones de pérdida, y_i son las etiquetas reales de los datos del conjunto de datos e $\hat{y}_i$ son las predicciones hechas por el modelo.

El aprendizaje consistirá pues en que los pesos de los filtros de las diferentes capas del los modelos se ajusten al conjunto de entrenamiento que le hemos proporcionado. Sin embargo debido al alto número de parámetros que se utilizan, enfrentamos el problema del sobreajuste. En cada época del entrenamiento los pesos se van actualizando a medida que el valor de la función de pérdida baja. La actualización de los pesos se hace mediante el algoritmo de descenso por el gradiente, en el que en cada época, cada peso se actualiza de la siguiente manera:

$$w_{ij} \leftarrow w_{ij} - \eta\frac{\partial L}{\partial w_{ij}}$$

donde w_{ij} es el valor del peso, η es el factor de aprendizaje, un valor preestablecido que actúa de regulador de la actualización de los pesos y finalmente $\frac{\partial L}{\partial w_{ij}}$ es el gradiente de la función L, que es la función de pérdida correspondiente, respecto a w_{ij}

El sobreajuste es a efectos prácticos que la red se ajuste demasiado a los datos de entrada y no generalice bien sobre otros datos. Para evitar esto existen múltiples técnicas que hemos aplicado durante los experimentos.

4.5.1. Dropout

El dropout consiste en, durante el entrenamiento hacer que un porcentaje de los pesos no entrenen. Los pesos que no entrenan se deciden aleatoriamente para que así no aprendan características específicas de conjunto de entrenamiento. En cada época se apagan diferentes pesos [13, página 2].

4.5.2. Aumento de datos (data augmentation)

La técnica del aumento de datos se basa en realizar diferentes modificaciones a las imágenes de entrada para que, manteniendo una base similar, tengamos una cantidad de datos más amplia con diferentes características. De esta forma la red es

capaz de aprender las características deseadas, permitiéndole así generalizar mejor sobre otros datos.[14, página 7]

Las modificaciones más comunes son rotaciones sobre el eje de simetría horizontal o vertical, rotaciones de diferentes grados, zoom-in o zoom-out, modificación en el brillo de la imagen o la introducción de ruido de sal y pimienta. [14, páginas 7-10] Pongamos de ejemplo algunas imágenes de nuestro conjunto de datos a las que se le han aplicado estas modificaciones:

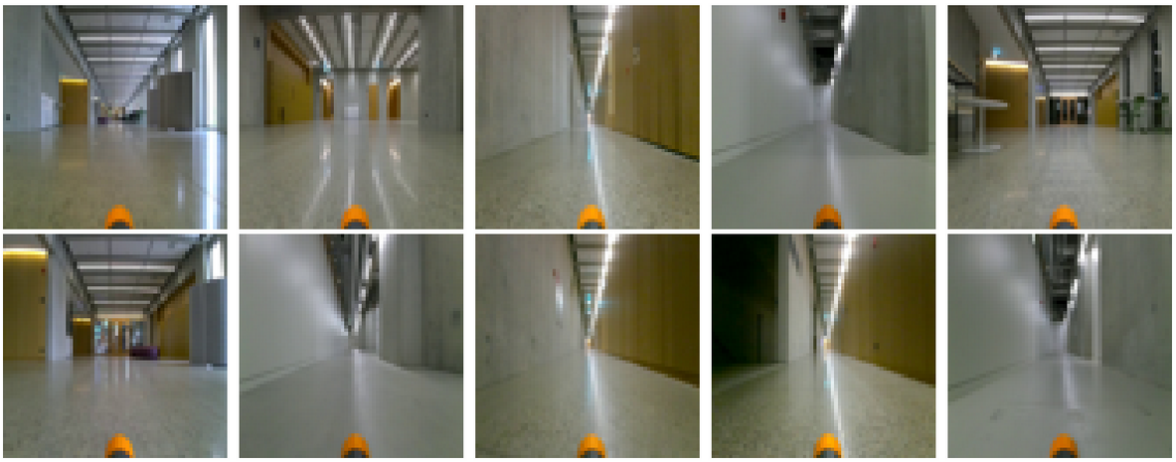

Figura 4.12: Imágenes originales

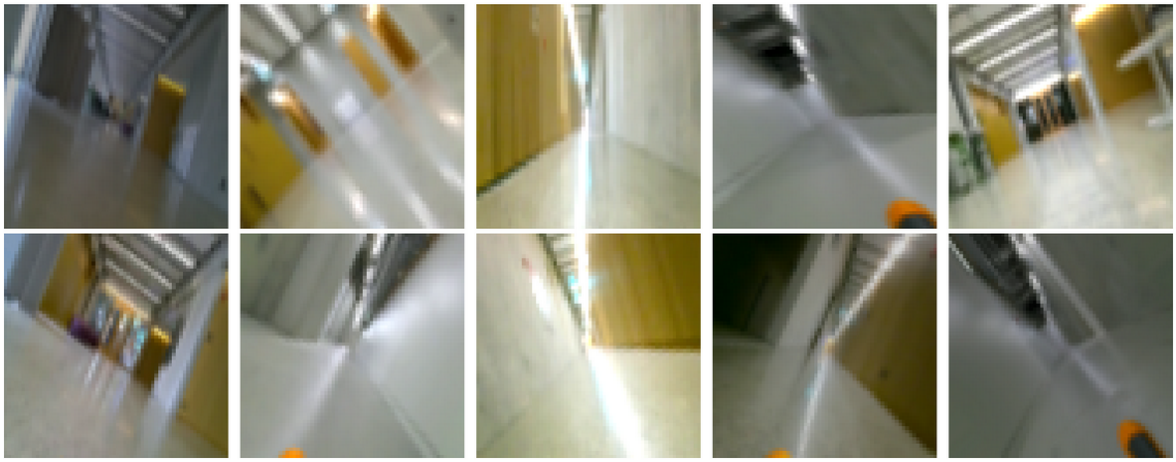

Figura 4.13: Imágenes modificadas

4.5.3. Regularización L1 y L2

La regularización de pesos se basa en añadir una penalización en la función de pérdida. [13, páginas 2-3]

- La función de pérdida con regularización L1 se define como:

$$\text{Loss} = \text{Loss}_{\text{original}} + \lambda \sum_i |w_i|$$

La regularización L1 agrega una penalización proporcional a la suma de los valores absolutos de los coeficientes (pesos) del modelo. Esta técnica se conoce como Lasso (Least Absolute Shrinkage and Selection Operator).

-La función de pérdida con regularización L2 se define como:

$$\text{Loss} = \text{Loss}_{\text{original}} + \lambda \sum_i w_i^2$$

La regularización L2 agrega una penalización proporcional a la suma de los cuadrados de los coeficientes (pesos) del modelo. Esta técnica se conoce como Ridge Regression.

De esta manera los pesos van cada vez aprendiendo menos para así impedir que se ajusten demasiado a características específicas del conjunto de entrenamiento.

4.5.4. Early stopping, ReduceLROnPlateau y gradient clipping

Otra práctica común para evitar el sobreajuste es añadir callbacks al entrenamiento. Un callback básicamente es un mecanismo de control que puede aplicarse durante el entrenamiento para personalizar y controlar su comportamiento.

En este caso usaremos early stopping que como su propio nombre indica consiste en parar el entrenamiento antes de que se complete al cien por cien. Para ello establecemos qué parámetro estará supervisando el callback y si se da un empeoramiento durante un número de épocas determinadas (al que llamamos paciencia) el entrenamiento para, normalmente restaurando los pesos que ofrecen un mejor resultado.

Por otra parte ReduceLROnPlateau es un callback que al igual que early stopping irá monitoreando un parámetro preestablecido. Sin embargo, en vez de parar el entrenamiento, cuando el parámetro empeora n épocas seguidas, lo que hará ReduceLROnPlateau, será reducir el factor de aprendizaje en un factor preestablecido.

Por último el gradient clipping es una técnica que consiste en limitar el valor máximo de los gradientes durante el entrenamiento.

4.5.5. Reducción de las dimensiones de la red

El último método que hemos utilizado para tratar de combatir el sobreajuste es reducir las dimensiones de la red. Al eliminar capas, tenemos menos parámetros lo cual beneficia de manera directa la reducción del sobreajuste ya que reduce la capacidad de aprender patrones complejos. [15, página 2]

4.5.6. Validación cruzada

La validación cruzada es una técnica que consiste en dividir el conjunto de entrenamiento en k subconjuntos. Una vez realizada la división, se utilizan k-1 conjuntos para entrenamiento y el restante como conjunto de validación. El conjunto de validación se utiliza para procesar los datos de este con el modelo y monitorizar si el modelo está entrenando correctamente. La distribución se va permutando k iteraciones hasta haber realizado todos los subentrenamientos. De esta manera se consigue reducir el sobreajuste[16].

Una vez explicados estos conceptos podemos proceder a desarrollar la explicación del trabajo realizado.

5. Ejecución del trabajo

Una vez introducidos los conceptos previos, vamos a pasar a desarrollar la explicación del trabajo realizado.

Empecemos definiendo cuál es el objetivo de este trabajo. La idea es, utilizando como referencia la arquitectura presentada en el artículo *An Outlier Exposure Approach to Improve Visual Anomaly Detection Performance for Mobile Robots* [1], hacer una traducción de esta, que originalmente está implementada en PyTorch, a Tensorflow y ver si podían mejorar los resultados de esta manera.

A continuación vamos a explicar todo el proceso que ha conllevado la traducción del código. Desde el estudio de la arquitectura, pasando por el conjunto de decisiones que se han tenido que ir tomando sobre la marcha para solventar las diferentes complicaciones que iban surgiendo y finalmente expondremos los resultados obtenidos.

5.1. Conjunto de datos

Para poder entender tanto la arquitectura como los problemas encontrados durante el desarrollo tenemos que explorar el conjunto de datos.

Este esta formado por 132838 imágenes 512 × 512 RGB, 105122 normales y 27716 con anomalías [1, página 4, apartado A]. El dataset se puede encontrar en https://zenodo.org/records/7074958.

Ejemplos de estas imágenes son las siguientes 5.1:

Figura 5.1: Ejemplos de imágenes, [1, página 5]

Las imágenes están divididas de la siguiente manera 5.1:

-	**Training set**	**Validation set**	**Testing set**
Normal frames	56066	2071	46585
Anomalous frames	18201	-	9515

Tabla 5.1: Dataset

Entrenaremos el autoencoder con el conjunto de imágenes normales del conjunto de entrenamiento.

Una vez que el autoencoder esté entrenado lo usaremos de las siguientes formas:

1. Como detector de anomalías.
2. Reductor de dimensiones para generar un conjunto de entrenamiento para un perceptrón multicapa.

Vamos ahora a describir con un poco más de profundidad ambas arquitecturas objetivo.

5.2. Estudio y definición de la arquitectura objetivo

Una vez tuvimos una comprensión inicial del paper [1], definimos el que sería el objetivo del trabajo: realizar la implementación de un autoencoder capaz de reducir la dimensionalidad de las imágenes tomadas por un robot y usar esta reducción como conjunto de entrenamiento para modelos más simples como es el perceptrón multicapa. La utilidad de esto es conseguir una estructura capaz de detectar si en la imagen hay anomalías prácticamente en tiempo real. Además el autoencoder puede ser usado de por sí solo como clasificador de anomalías como veremos más adelante.

En los siguientes subapartados, vamos a hacer ahora una explicación más extensa de los dos modelos que he implementado, propuestos en [1]. Por un lado he desarrollado el autoencoder como detector de anomalías y por otro he combinado el autoencoder, usándolo como reductor de la dimensión los elementos del conjunto de entrenamiento, con un perceptrón multicapa que funcionará como clasificador dando así pie a dos arquitecturas distintas que cumplen con la misma función.

1. Autoencoder
2. Autoencoder + MLP

5.2.1. Autoencoder como detector de anomalías

Como ya hemos mencionado previamente en este trabajo, un autoencoder codifica y luego reconstruye imágenes reduciendo el error cuadrático medio [1, página 3]. Podemos usar este error en la reconstrucción como puntuación de anomalía de la siguiente manera:

1. Entrenamos el autoencoder con un conjunto de datos en el que las imágenes estén libres de anomalías.
2. Cuando pasamos una imagen por la red si la reconstruye con un valor bajo en la función de pérdida tendremos una imagen sin anomalías. Por otra parte si el valor es alto significa que la imagen original dista mucho de la reconstruida ya que el autoencoder no es capaz de reconstruir anomalías .

La estructura propuesta por los investigadores es la siguiente:

Figura 5.2: Estructura autoencoder paper, [1, página 5]

Se usa LeakyRelu 5.1 como función de activación en todas sus capas menos en las que específicamente usa una función lineal. Después de cada convolución el tamaño de la imagen queda reducido a la mitad y en cada upsample queda multiplicado por dos [1, página 5].

$$f(x) = \begin{cases} x & \text{si } x \geq 0 \\ \alpha x & \text{si } x < 0 \end{cases} \tag{5.1}$$

5.2.2. Autoencoder + MLP

Por otra parte utilizaremos el autoencoder entrenado para reducir la dimensionalidad del conjunto de entrenamiento que utilizamos para entrenar un perceptrón multicapa muy simple que funcionará como clasificador.

De esta manera las imágenes 64x64x3 quedan reducidas a un vector 128 haciendo que su procesamiento sea mucho menos costoso.

El conjunto de entrenamiento del perceptrón está formado por imágenes codificadas que contienen anomalías y otras que no las contienen [1, página 5, apartado D].

El perceptrón propuesto consta de tres capas de 256, 64 y 1 neuronas respectivamente con activaciones Relu en las dos primeras y sigmoide en la última capa y usa como función de pérdida la entropía binaria cruzada.

Al ser un modelo tan simple se ha implementado sin ninguna modificación en mi código.

```
MLP = Sequential([
    Dense(256, input_shape=(128,), activation='relu'),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

opt = keras.optimizers.Adam(learning_rate=0.0001)
MLP.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
MLP.summary()
```

Extracto de código 5.1: Perceptrón multicapa

5.3. Decisión del entorno de desarrollo

Para desarrollar el código primero tuvimos que decidir en qué entorno íbamos a desarrollarlo. En una de las primeras reuniones se decidió que se iba a generar un notebook de Python en el cual se realizaría el trabajo. Por tanto, esto nos dejó con dos opciones principalmente:

1. Trabajar en local con JupyterNotebook.

2. Usar GoogleColab, que como se ve en el apartado de planificación 3, fue la opción tomada.

La decisión de usar GoogleColab y no trabajar en local fue tomada ya que aportaba la ventaja de que no iba a ser el equipo del alumno el que trabajase los entrenamientos, quitándole una gran carga a este. Además nos solucionaba el trabajo de instalar las diferentes librerías utilizadas, ya que se pueden usar directamente en la herramienta de Google.

5.4. Desarrollo del código

En este apartado vamos a explicar cómo hemos llevado a cabo el desarrollo del código de la traducción a Tensorflow. El código está disponible en el siguiente github. Estará dividido en los siguientes subapartados:

1. Importación de los datos a GoogleColab.
2. Versiones y entrenamiento de los autoencoders.
3. Uso del autoencoder como detector de anomalías.
4. Codificación de imágenes.
5. Optimización de la importación de los datos.
6. Perceptrón multicapa.

Durante la explicación de ambos subapartados iremos viendo las diferentes dificultades que hemos enfrentado.

5.4.1. Importación de los datos a GoogleColab

El primer problema que enfrentamos a la hora de llevar los datos a la nube fue que nos equivocamos de dataset. En el github de los investigadores hay diferentes dataset correspondientes a diferentes papers. Inicialmente usamos un dataset de otro paper que a pesar de usar imágenes similares a las del dataset usado en [1], era mucho más grande pesando 20GB aproximadamente. Esto hacía que subirlo al entorno de ejecución cada vez que fuésemos a trabajar no fuese una opción, por tanto solo quedaba utilizar el enlace entre Google Drive y GoogleColab. Adquirimos pues GoogleOne para poder tener un mayor almacenamiento disponible en la nube y procedimos a subir los datos. Comentar que debían estar comprimidos en zip para que la importación a GoogleColab fuese más eficiente.

```
extract_path = '/content/'

data = '/content/drive/MyDrive/Corridors v2.0.zip'

weighs = '/content/drive/MyDrive/autoencoder_weights_my_version_1st_train.zip'

with zipfile.ZipFile(data, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

with zipfile.ZipFile(weighs, 'r') as zip_ref:
    zip_ref.extractall(extract_path)
```

Extracto de código 5.2: Código para importar las imágenes

Una vez teníamos ese dataset en GoogleColab ya podemos hacer uso de nuestras imágenes pero antes tuvimos que preprocesarlas para reducir la dimensión de $512 \times 512 \times 3$ a $64 \times 64 \times 3$ para poder usarlas como inputs en el autoencoder.

Esto lo hicimos mediante el siguiente código 5.3:

```
# Define el path a la carpeta donde están las imágenes
training_autoencoder_folder_path = '/content/s1_frames/training_set'

# Obtiene una lista de los nombres de las imágenes
image_files = os.listdir(training_autoencoder_folder_path)

# Carga las imágenes en una lista
image_list = []
for filename in image_files:
    img = Image.open(os.path.join(training_autoencoder_folder_path, filename))
    img = img.resize((64, 64))
    img = np.array(img)
    image_list.append(img)

# Convierte la lista en un array que puede procesar el autoencoder
X_train_autoencoder = np.array(image_list)

print(X_train_autoencoder.shape)
```

Extracto de código 5.3: Resize de las imágenes

Una vez detectamos el problema de que estábamos realizando los experimentos con un dataset que no correspondía, cambiamos de dataset al propio y usamos la estructura de importe ya montada para usarlos.

5.4.2. Versiones y entrenamiento de los modelos

Vamos a pasar ahora a lo que más tiempo nos ha llevado del trabajo, generar versiones de los autoencoder que funcionasen y además que consiguiesen entrenar hasta minimizar la función de pérdida hasta valores aceptables.

Versión 1

La primera versión del autoencoder que propusimos fue la siguiente: 5.4. Esta versión añade capas de pooling y upsample a la estructura básica del autoencoder propuesta en [1].

Los problemas que presentaba esta versión es que al tener un mayor número de capas los entrenamientos eran muy largos llegando a demorar hasta cuatro horas ya que para poder bajar la pérdida en la reconstrucción tuvimos que entrenar por 300 épocas consiguiendo bajarla hasta 17 tanto en el conjunto de entrenamiento como en el de validación.

```
# Encoder
encoder_input = tf.keras.Input(shape=(64, 64, 3))

x = layers.Conv2D(128, (3, 3), activation=tf.nn.leaky_relu, padding='same',name="conv1")(encoder_input)
x = MaxPooling2D((2, 2), padding='same',name="pool1")(x)

x = layers.Conv2D(256, (3, 3), activation=tf.nn.leaky_relu, padding='same',name="conv2")(x)
x = MaxPooling2D((2, 2), padding='same',name="pool2")(x)

x = layers.Conv2D(512, (3, 3), activation=tf.nn.leaky_relu, padding='same',name="conv3")(x)
x = MaxPooling2D((2, 2), padding='same',name="pool3")(x)

x = layers.Conv2D(1024, (3, 3), activation=tf.nn.leaky_relu, padding='same',name="conv4")(x)
x = MaxPooling2D((2, 2), padding='same',name="pool4")(x)

x = layers.Flatten()(x)
encoded = layers.Dense(128, activation="linear", name="dense_layer")(x)
encoder = Model(encoder_input, encoded, name='encoder')


# Decoder
decoder_input = layers.Input(shape=(128,))

x = layers.Dense(16384, activation=tf.nn.leaky_relu)(decoder_input)
x = layers.Reshape((8, 8, 256))(x)

x = layers.Conv2DTranspose(512, (3, 3), padding='same', activation=tf.nn.leaky_relu,name="conv5")(x)
x = UpSampling2D((2, 2),name="up1")(x)

x = layers.Conv2DTranspose(256, (3, 3), padding='same', activation=tf.nn.leaky_relu,name="conv6")(x)
x = UpSampling2D((2, 2),name="up2")(x)

x = layers.Conv2DTranspose(128, (3, 3),  padding='same', activation=tf.nn.leaky_relu,name="conv7")(x)
x = UpSampling2D((2, 2),name="up3")(x)

decoded = layers.Conv2DTranspose(3, (3, 3), activation='linear', padding='same', name="final_conv")(x)


decoder = Model(decoder_input, decoded, name='decoder')

# Autoencoder model
autoencoder_input = tf.keras.Input(shape=(64, 64, 3))
encoded_img = encoder(autoencoder_input)
decoded_img = decoder(encoded_img)
autoencoder = Model(autoencoder_input, decoded_img, name='autoencoder')

opt = keras.optimizers.Adam(learning_rate=0.001)
autoencoder.compile(optimizer=opt, loss='mse')  # Using mean squared error as the loss function
```

Extracto de código 5.4: Autoencoderv1

Sin embargo a pesar de que conseguía clasificar el conjunto de testeo con un 65 por ciento de precisión, es decir clasificaba las imágenes no anómalas como 0 en la mayoría de los casos y las anómalas como 1. Descartamos este modelo ya que al usar la métrica AUC [1, página 6, Apartado F] obteníamos un valor de 0.5. La métrica AUC es una medida utilizada para evaluar el rendimiento de un modelo de clasificación binaria. Esta muestra un gráfico que enfrenta los verdaderos positivos frente a los falsos positivos. Obtener una puntuación de un 1 sería tener un clasificador perfecto y un 0.5 representaría acertar solo la mitad de las veces.

Versión 2

La siguiente versión es la traducción directa capa por capa del modelo 5.5. Esta traduce directamente las capas del modelo original a Tensorflow.

```
# ENCODER
encoder = keras.Sequential(name='encoder')

encoder.add(keras.layers.Conv2D(filters=128, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu", name
    ='input', input_shape=(64, 64, 3)))

encoder.add(keras.layers.Conv2D(filters=256, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu", name
    ='conv1'))

encoder.add(keras.layers.Conv2D(filters=512, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu", name
    ='conv2'))

encoder.add(keras.layers.Conv2D(filters=1024, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
    name='conv3'))

encoder.add(keras.layers.Flatten(name='flatten'))
encoder.add(keras.layers.Dense(units=128, activation='linear', name='dense'))
encoder.add(keras.layers.Embedding(input_dim=128, output_dim=128, embeddings_initializer='glorot_uniform',
    embeddings_regularizer='L1L2'))

# DECODER
decoder = keras.Sequential(name='decoder')

decoder.add(keras.layers.Input(shape=(128, 128), name='input'))
decoder.add(keras.layers.Reshape((4, 4, 1024)))

decoder.add(keras.layers.Conv2D(filters=512, kernel_size=(3, 3), activation='leaky_relu', padding='same'))

decoder.add(keras.layers.Conv2DTranspose(filters=256, kernel_size=(3, 3), strides=(2, 2),  activation="
    leaky_relu", padding='same', name='deconvconv1'))

decoder.add(keras.layers.Conv2DTranspose(filters=128, kernel_size=(3, 3), strides=(2, 2), activation="
    leaky_relu", padding='same', name='deconvconv2'))

decoder.add(keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), strides=(2, 2),  activation="
    leaky_relu", padding='same', name='deconvconv3'))

decoder.add(keras.layers.Conv2DTranspose(filters=3, kernel_size=(3, 3), strides=(2, 2), activation="linear",
    padding='same', name='output'))

# AUTOENCODER
autoencoder_input = keras.layers.Input(shape=(64, 64, 3), name='autoencoder_input')
encoded_repr = encoder(autoencoder_input)
decoded_repr = decoder(encoded_repr)

autoencoder = keras.models.Model(inputs=autoencoder_input, outputs=decoded_repr, name='autoencoder')

opt = keras.optimizers.Adam(learning_rate=0.001)
autoencoder.compile(optimizer=opt, loss='mse')
```

Extracto de código 5.5: Autoencoder v2

El problema que presentó esta arquitectura fue que a partir de tempranos épocas el modelo se sobreajustaba demasiado al conjunto de entrenamiento. Cuando el entrenamiento llevaba aproximadamente 15 épocas dejaba de entrenar y la pérdida se estancaba en 50 aproximadamente. Por ello pasamos a la tercera versión.

Versión 3

La tercera versión 5.6 implementa la regularización L1L2 en la capa en la que se genera la codificación. Simultáneamente se implementa en la misma capa la inicialización de los pesos de esta. Inicialmente se probó inicializarlos con *he-normal* y conseguimos de esta manera retrasar el sobreajuste hasta el épocas 100 sin embargo la pérdida quedaba en 45 que seguía siendo elevada para los resultados que queremos obtener. Luego probamos la inicialización de los pesos con *glorot-uniform* y conseguimos que el sobreajuste se diera a las 115 épocas. Sin embargo, la pérdida se estancaba aproximadamente en el mismo valor.

```
# ENCODER
encoder = keras.Sequential(name='encoder')

encoder.add(keras.layers.Conv2D(filters=128, kernel_size=(3, 3), strides=(2, 2),
                                activation="leaky_relu", name='input', input_shape=(64, 64, 3)))

encoder.add(keras.layers.Conv2D(filters=256, kernel_size=(3, 3), strides=(2, 2),
                                activation="leaky_relu", name='conv1'))

encoder.add(keras.layers.Conv2D(filters=512, kernel_size=(3, 3), strides=(2, 2),
                                activation="leaky_relu", name='conv2'))

encoder.add(keras.layers.Conv2D(filters=1024, kernel_size=(3, 3), strides=(2, 2),
                                activation="leaky_relu", name='conv3',
                                kernel_regularizer=regularizers.l2(0.01)))

encoder.add(keras.layers.Flatten(name='flatten'))
encoder.add(keras.layers.Dense(units=128, activation='linear', name='dense',
                               kernel_regularizer=regularizers.l2(0.01)))
encoder.add(keras.layers.Embedding(input_dim=128, output_dim=128, embeddings_initializer='glorot_uniform',
    embeddings_regularizer='L1L2'))

# DECODER
decoder = keras.Sequential(name='decoder')

decoder.add(keras.layers.Input(shape=(128, 128), name='input'))
decoder.add(keras.layers.Reshape((4, 4, 1024)))

decoder.add(keras.layers.Conv2D(filters=512, kernel_size=(3, 3), activation='leaky_relu', padding='same'))

decoder.add(keras.layers.Conv2DTranspose(filters=256, kernel_size=(3, 3), strides=(2, 2),
                                         activation="leaky_relu", padding='same', name='deconvconv1',
                                         kernel_regularizer=regularizers.l2(0.01)))

decoder.add(keras.layers.Conv2DTranspose(filters=128, kernel_size=(3, 3), strides=(2, 2),
                                         activation="leaky_relu", padding='same', name='deconvconv2'))

decoder.add(keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), strides=(2, 2),
                                         activation="leaky_relu", padding='same', name='deconvconv3'))

decoder.add(keras.layers.Conv2DTranspose(filters=3, kernel_size=(3, 3), strides=(2, 2),
                                         activation="linear", padding='same', name='output'))

# AUTOENCODER
autoencoder_input = keras.layers.Input(shape=(64, 64, 3), name='autoencoder_input')
encoded_repr = encoder(autoencoder_input)
decoded_repr = decoder(encoded_repr)
autoencoder = keras.models.Model(inputs=autoencoder_input, outputs=decoded_repr, name='autoencoder')
opt = keras.optimizers.Adam(learning_rate=0.001)
autoencoder.compile(optimizer=opt, loss='mse')
```

Extracto de código 5.6: Autoencoder v3

También se probó a aplicar dropout y data augmentation. Para ambas aplicaciones el modelo no era capaz de reducir la función de perdida.

El aumento de datos se hizo de la siguiente manera 5.7. Aplicamos rotaciones de 40º, flip horizontal, cambios al brillo y zoom:

```
images = X_train_autoencoder
data_format = 'channels_last'

datagen = ImageDataGenerator(
    rotation_range=40,
    horizontal_flip=True,
    brightness_range=[0.6, 1.7],
    zoom_range=[0.5, 1.0],
    fill_mode='nearest'
)

augmented_images = datagen.flow(images, batch_size=len(images), shuffle=False).next()
```

Extracto de código 5.7: Data augmentation

Aun así, para poder mejorar los resultados obtenidos tuvimos que tomar otra alternativa que fue probar a entrenar el modelo mediante la técnica de validación cruzada 5.8.

```
# Parámetros de cross-validation
n_splits = 10
kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)

model = autoencoder

mse_scores = []
i = 0
for train_index, val_index in kf.split(X_train_autoencoder):
    early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=4, min_lr=1e-7)

    X_train_fold, X_val_fold = X_train_autoencoder[train_index], X_train_autoencoder[val_index]
    # Entrenar el modelo
    model.fit(X_train_fold, X_train_fold, epochs=10, batch_size=32, validation_data=(X_val_fold, X_val_fold),
     verbose=1, callbacks=[early_stopping, reduce_lr])

    # Evaluar el modelo
    mse_loss = model.evaluate(X_val_fold, X_val_fold, verbose=0)
    mse_scores.append(mse_loss)
    i += 1
    print(f'Vamos por el split {i}')

print(f"Mean MSE over {n_splits} folds: {np.mean(mse_scores)}")
```

Extracto de código 5.8: Cross Validation

Esta técnica consiguió aumentar los épocas en los que se producía el sobreajuste considerablemente. Sin embargo el modelo seguía estancado en la misma pérdida. Por tanto pasamos a la siguiente versión.

Versión 4

La cuarta versión del autoencoder 5.9 elimina la capa de embeddings y genera la codificación directamente de un vector 128 como hacíamos en la versión original. De esta forma y aplicando la inicialización de pesos en *he-normal* y *gradient-clipping*, sumado al uso de *cross-validation* para el entrenamiento consiguieron bajar la pérdida hasta el valor de 35. De todas formas, aún se podía bajar aún más ya que con la primera versión conseguimos llegar a 17.

```
 # Normalización de la entrada (suponiendo que tus datos están en formato numpy array)

 # Supongamos que X_train es tu conjunto de datos de entrenamiento

 # ENCODER
encoder = models.Sequential(name='encoder')
encoder.add(layers.Conv2D(filters=64, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), input_shape=(64, 64, 3),
     name='input'))
encoder.add(layers.Conv2D(filters=128, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), name='conv1'))
encoder.add(layers.Conv2D(filters=256, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), name='conv2'))
encoder.add(layers.Flatten(name='flatten'))
encoder.add(layers.Dense(units=128, activation='linear', kernel_initializer='glorot_uniform',
     kernel_regularizer=regularizers.l2(0.01), name='dense'))

 # DECODER
decoder = models.Sequential(name='decoder')
decoder.add(layers.Input(shape=(128,), name='input'))
decoder.add(layers.Dense(4*4*256, activation='linear', kernel_initializer='glorot_uniform', kernel_regularizer
     =regularizers.l2(0.01), name='dense'))
decoder.add(layers.Reshape((4, 4, 256)))
decoder.add(layers.Conv2DTranspose(filters=128, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), padding='same', name='
     deconvconv1'))
decoder.add(layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), padding='same', name='
     deconvconv2'))
decoder.add(layers.Conv2DTranspose(filters=32, kernel_size=(3, 3), strides=(2, 2), activation="leaky_relu",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), padding='same', name='
     deconvconv3'))
decoder.add(layers.Conv2DTranspose(filters=3, kernel_size=(3, 3), strides=(2, 2), activation="linear",
     kernel_initializer='glorot_uniform', kernel_regularizer=regularizers.l2(0.01), padding='same', name='
     output'))

 # AUTOENCODER
autoencoder_input = layers.Input(shape=(64, 64, 3), name='autoencoder_input')
encoded_repr = encoder(autoencoder_input)
decoded_repr = decoder(encoded_repr)

autoencoder = models.Model(inputs=autoencoder_input, outputs=decoded_repr, name='autoencoder')

 # Compile the model
opt = Adam(learning_rate=0.001, clipnorm=1.0)
autoencoder.compile(optimizer=opt, loss='mse')
```

Extracto de código 5.9: Autoencoder v4

Versión 5

Pasamos así a la versión final del autoencoder 5.10. En esta versión eliminamos capas de la estructura del autoencoder para así reducir los parámetros de la red y reducir el sobreajuste. Esto combinado con la inicialización de los pesos de las capas densas con *glorot-uniform* y el uso de *cross-validation* en el entrenamiento consiguió bajar la pérdida del modelo a 22. Decidimos aceptarlo ya que probando con las diferentes combinatorias durante los entrenamientos no conseguimos obtener mejores resultados.

```
encoder = keras.Sequential(name='encoder')

encoder.add(keras.layers.Conv2D(filters=32, kernel_size=(3, 3),strides=(2, 2),padding='same',activation='leaky_relu',name='conv1',input_shape=(64, 64, 3)))

encoder.add(keras.layers.Conv2D(filters=64,kernel_size=(3, 3),strides=(2, 2),padding='same',activation='leaky_relu',name='conv2',
))

encoder.add(keras.layers.Conv2D(filters=128,kernel_size=(3, 3),strides=(2, 2),padding='same',activation='leaky_relu',name='conv3',
))

encoder.add(layers.Flatten(name='flatten'))
encoder.add(layers.Dense(units=128, activation='linear', kernel_initializer='glorot_uniform', name='dense'))

decoder = keras.Sequential(name='decoder')
decoder.add(layers.Dense(units=8 * 8 * 128, activation='linear', kernel_initializer='glorot_uniform', name='dense2'))
decoder.add(layers.Reshape((8, 8, 128), name='reshape'))

decoder.add(keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3),strides=(2, 2), padding='same', activation='leaky_relu', name='conv5',
))

decoder.add(keras.layers.Conv2DTranspose( filters=32, kernel_size=(3, 3),strides=(2, 2),padding='same', activation='leaky_relu',name='conv6',
))

decoder.add(keras.layers.Conv2DTranspose(filters=3,kernel_size=(3, 3),strides=(2, 2),padding='same',activation='linear',name='output'
))

autoencoder_input = layers.Input(shape=(64, 64, 3), name='autoencoder_input')
encoded_repr = encoder(autoencoder_input)
decoded_repr = decoder(encoded_repr)

autoencoder = models.Model(inputs=autoencoder_input, outputs=decoded_repr, name='autoencoder')
opt = Adam(learning_rate=0.001, clipnorm=1.0)
autoencoder.compile(optimizer=opt, loss='mse')
```

Extracto de código 5.10: Autoencoder v5

El entrenamiento de las versiones 2 en adelante tarda aproximadamente unos 25 minutos con el entorno L4, que es de pago en GoogleColab. Esto mejora considerablemente las 4 horas de entrenamiento de la versión 1. El motivo de la adquisición de GoogleColab premium fue que, en la versión gratuita, la gráfica se quedaba sin memoria al ser modelos de un tamaño relativamente elevado.

5.4.3. Ejemplo de uso del autoencoder

Como hemos establecido, queremos utilizar el autoencoder como detector de anomalías utilizando el error en la reconstrucción de la imagen a modo de puntuación de anomalía. Si la imagen no tiene anomalía la reconstrucción cometerá un error bajo. Por otro lado, si la imagen contiene anomalías la reconstrucción cometerá un error elevado ya que no reconstruirá la anomalía en sí debido a que no está entrenado para ello.

Vamos a ver ahora como realiza estas reconstrucciones 5.11:

```
image_array1 = img
image_array2 = test1_decode[0]

fig, axes = plt.subplots(1, 2)

axes[0].imshow(image_array1.astype('uint8'))
axes[0].axis('off')
axes[0].set_title('Original')
axes[1].imshow(image_array2.astype('uint8'))
axes[1].axis('off')
axes[1].set_title('Reconstruida')
plt.show()
```

Extracto de código 5.11: Reconstrucción sin anomalía

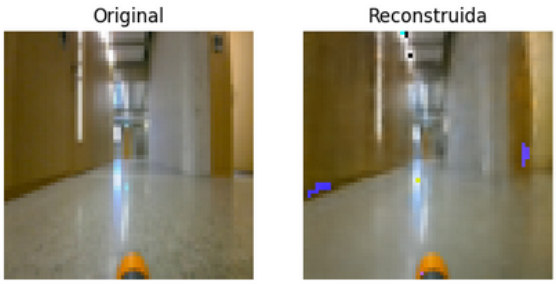

Figura 5.3: Reconstrucción sin anomalía

Como podemos ver la reconstrucción de la imagen es bastante fiable. Concretamente el error cometido es de un valor de 46.

Veamos ahora la reconstrucción de una imagen que contiene una anomalía 5.12.

```
image_array1 = img2
image_array2 = test2_decode[0]

fig, axes = plt.subplots(1, 2)

axes[0].imshow(image_array1.astype('uint8'))
axes[0].axis('off')
axes[0].set_title('Original')

axes[1].imshow(image_array2.astype('uint8'))
axes[1].axis('off')
axes[1].set_title('Reconstruida')

plt.show()
```

Extracto de código 5.12: Reconstrucción con anomalía

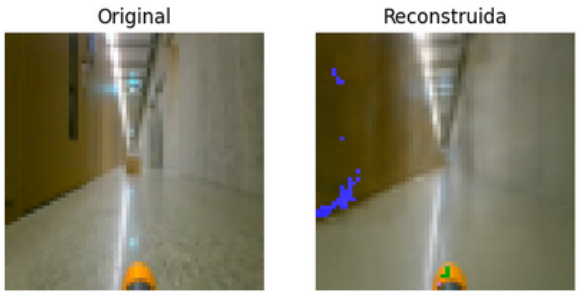

Figura 5.4: Reconstrucción con anomalía

Como podemos ver al reconstruir la imagen la caja que se observa en la imagen original ha sido eliminada en la reconstrucción haciendo que el error cometido aumente considerablemente, dependiendo de la imagen el error de reconstrucción de una imagen que contiene una anomalía va de los 200 hacia arriba. Este aumento es lo que usaremos como indicador de que en la imagen hay una anomalía.

5.4.4. Autoencoder como detector de anomalías

Para poder usar el autoencoder de esta manera hemos creado un par de funciones auxiliares que dada una imagen, predice si es una anomalía.

La primera función simplemente calcula el error medio cuadrático que se produce al reconstruir la imagen (que es la función que minimiza la pérdida).

```
def mse_autoencoder_predict(image,path):
    img = Image.open(os.path.join(path, image))
    img = img.resize((64, 64))
    img = np.array(img)
    img_array_reshaped = np.expand_dims(img, axis=0)
    encoder_model = autoencoder.get_layer('encoder')
    decoder_model = autoencoder.get_layer('decoder')
    encoded_img = encoder_model.predict(img_array_reshaped)
    decoded_img = decoder_model.predict(encoded_img)
    mse = np.mean(np.square(img- decoded_img[0]))
    return mse
```

Extracto de código 5.13: Error cuadrático medio

La función codifica, decodifica y finalmente calcula el error cometido. Vamos a mostrar ahora dos ejemplos 5.5 . En el primero la imagen contiene una anomalía y en el segundo no. Observamos como el error del primer cálculo es mucho mayor.

```
[ ]  mse_autoencoder_predict('010775_512_512.jpg',test_models_folder_path)

     1/1 [==============================] - 0s 22ms/step
     1/1 [==============================] - 0s 20ms/step
     232.28996

[ ]  mse_autoencoder_predict('000700_512_512.jpg',test_models_folder_path)

     1/1 [==============================] - 0s 23ms/step
     1/1 [==============================] - 0s 21ms/step
     60.518055
```

Figura 5.5: Ejemplos error cuadrático medio

La siguiente función 5.14 es la que realiza la clasificación de la imagen. Simplemente si la función 5.13 devuelve un error mayor a un umbral, clasifica la imagen como anomalía. En este caso a base de diferentes testeos hemos establecido que el umbral de detección sea 200 ya que es con el que hemos obtenido los mejores resultados.

```
def classifier_AE(name_list,path):
  y_predict = []
  for i in range(len(name_list)):
    mse = mse_autoencoder_predict(name_list[i],path)
    print(mse)
    if mse >= 200:
      y_predict.append(1)
    else:
      y_predict.append(0)
    print(i)
  return y_predict
```

Extracto de código 5.14: Clasificador autoencoder

Para predecir el conjunto de datos entero con un bucle *for* que computase ambas funciones tardamos 8 horas. Por lo tanto hicimos una versión de las funciones de forma que hiciera la predicción del conjunto de datos por lotes 5.15. De esta forma la predicción pasó a tardar 2 horas.

```python

def mse_autoencoder_predict(images, path, autoencoder):
    imgs = []
    for image in images:
        img = Image.open(os.path.join(path, image))
        img = img.resize((64, 64))
        img = np.array(img)
        imgs.append(img)

    imgs_array = np.array(imgs)
    encoder_model = autoencoder.get_layer('encoder')
    decoder_model = autoencoder.get_layer('decoder')
    encoded_imgs = encoder_model.predict(imgs_array)
    decoded_imgs = decoder_model.predict(encoded_imgs)

    mses = []
    for img, decoded_img in zip(imgs_array, decoded_imgs):
        img_flat = img.flatten()
        decoded_img_flat = decoded_img.flatten()
        mse = mean_squared_error(img_flat, decoded_img_flat)
        mses.append(mse)

    return mses

def classifier_AE(name_list, path, autoencoder, batch_size=32):
    y_predict = []
    for i in range(0, len(name_list), batch_size):
        batch_names = name_list[i:i+batch_size]
        mses = mse_autoencoder_predict(batch_names, path, autoencoder)
        print(batch_names)
        print(mses)
        for mse in mses:
            if mse >= 200:
                y_predict.append(1)
            else:
                y_predict.append(0)
        print(f'Processed batch {i//batch_size + 1}')
    return y_predict
```

Extracto de código 5.15: Clasificador autoencoder por batches

Una vez hechas las predicciones calculamos el porcentaje de acierto y obtuvimos aproximadamente un 63 por ciento, valor muy parecido a la primera versión.

De nuevo, al aplicar de nuevo la métrica de AUC, volvemos a obtener un 0.5.

Esto es debido a que a pesar de que clasifica correctamente de forma general, cuando las anomalías son muy pequeñas aun en las imágenes no es capaz de detectar que están ahí ya que la reconstrucción es prácticamente igual y no sobrepasa el umbral en la clasificación.

5.4.5. Codificación de las imágenes

Para realizar la codificación de las imágenes del conjunto de testeo simplemente aplicamos un bucle for que va codificando por lotes 5.16.

```
images = X_train_models

encoder_model = autoencoder.get_layer('encoder')

batch_size = 32
encoded_images1 = []

for i in range(0, len(images), batch_size):
    batch_images1 = images[i:i+batch_size]
    encoded_imgs1 = encoder_model.predict(batch_images1)
    encoded_images1.append(encoded_imgs1)

encoded_images1 = np.concatenate(encoded_images1, axis=0)


print(encoded_images1.shape)
```

Extracto de código 5.16: Codificación de las imágenes

Hacemos esto tanto sobre el conjunto de entrenamiento del perceptrón multicapa como sobre el conjunto de testeo para realizar las pruebas finales.

5.4.6. Perceptrón multicapa

Como vimos en 4.3 la estructura del perceptrón es muy simple constando solo de tres capas densas que hará de clasificador de las imágenes codificadas.

El perceptrón obtiene una reducción de su función de error de 2.3941e-12 obteniendo un porcentaje de acierto del 100 por cien sobre el conjunto de entrenamiento.

Una vez entrenado el modelo realizamos las predicciones sobre el conjunto de prueba 5.17, para el que obtenemos una precisión aproximada del 60 por ciento. Tomamos como precisión el número de aciertos dividido entre el número total de predicciones.

```
predictions = MLP.predict(X_test)

y_predict=[]
for e in predictions:
  if e > 0.5:
    y_predict.append(1)
  else:
    y_predict.append(0)
y_predict = np.array(y_predict)

coincidences=[]

for elem1,elem2 in zip(y_test,y_predict):
  if elem1 ==elem2:
    coincidences.append(1)

len(coincidences)/len(y_test) * 100
```

Extracto de código 5.17: Predicciones MLP

Sin embargo al evaluar el modelo bajo el AUC, volvemos a obtener un 0.5 debido a que acarreamos el problema del autoencoder en la codificación. Los vectores generados en los que las anomalías son muy pequeñas acaban clasificados como no anómalos en vez de anómalos.

6. Contenido del github del proyecto

Esta sección es un pequeño resumen del contenido del github en el que se encuentra el código de este proyecto.

En este encontraremos los siguientes archivos:

1. Este mismo documento.
2. Los pesos para la versión 4 del autoencoder 5.9.
3. El modelo entrenado de la versión 5 del autoencoder 5.10 .
4. El modelo entrenado del perceptrón multicapa 5.1.
5. La carpeta metadata, perteneciente al código original.
6. Un pequeño resumen del trabajo.
7. El readme.
8. El archivo .ipynb, que es el notebook en el que se desarrolla el código.

Adjunto el documento del proyecto para que, si alguien quiere comprender mejor el código desarrollado en el notebook, pueda hacerlo.

Por otra parte he subido los modelos entrenados y los pesos por si se quieren probar sin tener que pasar por la fase del entrenamiento de estos.

La carpeta metadata contiene principalmente datos de las etiquetas de las imágenes de los conjuntos de datos. Es usada en el notebook para realizar las pruebas de precisión de los modelos.

Finalmente y siendo este el contenido mas importante del github, se presenta el notebook en el que hemos desarrollado el trabajo. El proceso de ejecución de este queda desarrollado en el propio notebook.

7. Conclusiones

Tras analizar la arquitectura propuesta en [1] realicé una traducción de su código que se encuentra en el siguiente github, consiguiendo producir dos modelos de predicción de anomalías que, cuando estas se hacen realmente notables en la imagen, son capaces de detectarlas.

En primer lugar, he implementado varias versiones del autoencoder aplicando las diferentes técnicas mencionadas para reducir el sobreajuste de los entrenamientos, consiguiendo mejorar la primera versión tanto en tiempo de entrenamiento como en su capacidad de reconstrucción de las imágenes. Por otra parte he combinado esta primera arquitectura con un perceptrón multicapa, dando así lugar a una segunda arquitectura con la misma funcionalidad y un rendimiento similar.

Durante el desarrollo de este trabajo he realizado una revisión bibliográfica que me ha permitido llevar a cabo la implementación y entrenamiento de los modelos presentados aplicando numerosas técnicas que se usan en deep learning para reducir el sobreajuste de estos. Algunas de estas técnicas ya se usaban en la versión del artículo, como el data augmentation. Sin embargo muchas otras mencionadas en la sección 4 han sido buscadas e implementadas desde cero.

Además, he montado un entorno explotando al máximo las funcionalidades de GoogleDrive y GoogleColab capaz de soportar las necesidades de este proyecto, proceso que ha presentado numerosas complicaciones que he podido ir solventando.

El notebook en el que he realizado los experimentos queda publicado en el siguiente github.

Bibliografía

[1] Dario Mantegazza, Alessandro Giusti, Luca Maria Gambardella, and Jérôme Guzzi. An outlier exposure approach to improve visual anomaly detection performance for mobile robots. *IEEE Robotics and Automation Letters*, 7(4): 11354–11361, 2022.

[2] OpenWebinars. Qué son las redes neuronales y sus aplicaciones, 2023. URL openwebinars.net.

[3] F. J. Martín Mateos and J. L. Ruiz Reina. Inteligencía artificial. tema 7: Introducción a las redes neuronales. In *Apuntes del DPTO. Ciencias de la Computación e Inteligencia Artificial*, Sevilla, España, 2022-2023. Universidad de Sevilla.

[4] Estefanía Argente Oscar Sapena Vicente Botti and JM Serra. Aplicación de una red neuronal para la predicción de la reacción catalítica isomerización del n-octano. *Valencia: Universitat Politècnica de València*, 2001.

[5] Eduardo Sánchez Karhunen. Ampliación de inteligencia artificial, deep learning sesión 3 cnns. In *Apuntes del DPTO. Ciencias de la Computación e Inteligencia Artificial*, Escuela Técnica Superior de Ingeniería Informática, Sevilla, España, 2022-2023.

[6] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[7] Diogo Nunes Gonçalves, Vanessa Aparecida de Moares Weber, Julia Gindri Bragato Pistori, Rodrigo da Costa Gomes, Anderson Viçoso de Araujo, Marcelo Fontes Pereira, Wesley Nunes Gonçalves, and Hemerson Pistori. Carcass image segmentation using cnn-based methods. *Information Processing in Agriculture*, 8(4):560–572, 2021.

[8] Joseph Rocca. Understanding variational autoencoders (vaes), 2019. URL towardsdatascience.com.

[9] PyTorch. Pytorch, 2024. URL https://pytorch.org/.

[10] TensorFlow. Tensorflow, 2024. URL https://www.tensorflow.org/.

[11] Google Colab. Google colaboratory, 2024. URL https://colab.research.google.com/.

[12] Yifei Zhang. A better autoencoder for image: Convolutional autoencoder. In *ICONIP17-DCEC. Available online: http://users. cecs. anu. edu. au/Tom. Gedeon/conf/ABCs2018/paper/ABCs2018_paper_58. pdf (accessed on 23 March 2017)*, 2018.

[13] Michael Cogswell, Faruk Ahmed, Ross Girshick, Larry Zitnick, and Dhruv Batra. Reducing overfitting in deep networks by decorrelating representations. *arXiv preprint arXiv:1511.06068*, 2015.

[14] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[15] Mustafa Abdul Salam, Ahmad Taher Azar, Mustafa Samy Elgendy, and Khaled Mohamed Fouad. The effect of different dimensionality reduction techniques on machine learning overfitting problem. *Int. J. Adv. Comput. Sci. Appl*, 12(4):641–655, 2021.

[16] Daniel Berrar et al. Cross-validation., 2019.