Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Detección de anomalías mediante redes neuronales convolucionales (CNN) en radio espectrómetros solares

Autor: Eneko Casado Moreno

Tutor: Pablo Muñoz Martínez
Cotutor: Manuel Prieto Mateo

2022

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado
Detección de anomalías mediante redes neuronales convolucionales (CNN) en radio espectrómetros solares

Autor: Eneko Casado Moreno

Tutor/es: Pablo Muñoz Martínez
Cotutor: Manuel Prieto Mateo

TRIBUNAL:

Presidente: José Raúl Durán Díaz

Vocal 1º: Antonio José de Vicente Rodríguez

Vocal 2º: Pablo Muñoz Martínez

FECHA: 20/09/2022

Agradecimientos

A todas las personas que me han ayudado y acompañado durante este camino.

A mi familia por estar siempre a mi lado, haberme educado como persona e inculcarme unos principios y valores que me definen como persona. A mis amigos y amigas por su apoyo incondicional en todos los momentos en los que lo he necesitado.

Resumen

En este Trabajo Fin de Grado se va a desarrollar una red neuronal profunda, basada en capas convolucionales, cuyo objetivo será detectar anomalías en radio espectrómetros solares. Actualmente, esta tarea es realizada por una persona que analiza manualmente los espectros de radio. Sin embargo, gracias al avance de la ciencia en el campo de la inteligencia artificial, se espera que sea posible automatizar este trabajo mediante el uso de una red neuronal. Para ello, se utilizarán los datos recopilados por la Red Científica Internacional e-Callisto.

Palabras claves: autoencoder, Keras, redes neuronales convolucionales, anomalías, radiación solar.

Abstract

In this Final Degree Project we are going to develop a deep neural network, based on convolutional layers, whose objective will be to detect anomalies in solar radio spectrometers. Currently, this task is done manually by a person who analyzes all data. However, thanks to the advancement of science in the field of artificial intelligence, it is expected that it will be possible to automate this work by using a neural network. For this purpose, we will use the data collected by the International Scientific Network e-Calisto.

Keywords: autoencoder, Keras, convolutional neural networks, anomalies, solar radiation.

Resumen extendido

La radiación solar es la energía producida en el núcleo del Sol por las reacciones de fusión nuclear del hidrógeno que es emitida por la superficie solar. Esta energía se propaga en todas las direcciones a través del espacio mediante ondas electromagnéticas. La radiación solar se distribuye desde la radiación infrarroja hasta la ultravioleta, sin embargo, no toda la radiación alcanza la superficie de la Tierra, ya que las ondas ultravioletas más cortas, las que son peligrosas para la salud, son absorbidas por los gases de la atmósfera.

Un amplio rango de aplicaciones es afectado por la radiación solar como, por ejemplo, diversas áreas de la meteorología, ingeniería, agricultura, ganadería, arquitectura… También, es el motor que determina la dinámica de los procesos atmosféricos y climatológicos, además de ser responsable de eventos de nuestro día a día que permiten la vida en la tierra.

Los fenómenos que ocurren en el Sol afectan directamente a las mediciones recogidas de la radiación solar que son medidas por radioespectrómetros solares. Un radioespectrómetro solar es un instrumento que recoge el espectro de emisiones de radio producidas por el Sol, gracias a él podemos medir las propiedades de la luz en una parte determinada del espectro electromagnético. También se llaman así a las mediciones que recoge dicho instrumento.

Estos sucesos que causan las anomalías producen flujos de partículas que afectan al viento solar y a la magnetosfera terrestre. También provocan perturbaciones en la red eléctrica, en la comunicación por satélite o en la ubicación GPS. Además, puede suponer un peligro para los astronautas y las naves espaciales debido a la radiación. Asimismo, influyen en la creación de fenómenos naturales como las auroras boreales y las australes. Por último, estos fenómenos liberan un flujo enorme de protones que pueden causar daño bioquímico en el cuerpo humano. Normalmente, para identificar la existencia de anomalías de los eventos solares, hay que tener en cuenta la frecuencia, la duración y la distribución de la radiación [1].

Contenido

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

La emisión de radio solar son las ondas de radio producidas por el Sol. En caso de que la emisión de radio del Sol se eleve por encima del nivel de fondo de microondas, dichas emisiones se llamarán ráfagas de radio solar y serán las anomalías que queremos detectar. Estas anomalías serán causadas por eventos solares como las erupciones solares, las eyecciones de masa coronal o las manchas solares.

Durante estos fenómenos el Sol emite partículas cargadas eléctricamente, que pueden alcanzar nuestro planeta a gran velocidad. Esto puede provocar interferencias en las señales de onda de radio, en las líneas de comunicaciones, en la red eléctrica o en los sistemas de geolocalización. Además, estos sistemas pueden sufrir sobrecargas que dañen los transformadores que los componen y reemplazarlos supone una gran pérdida de tiempo y dinero.

Para evitar estos problemas, la única solución viable es conseguir un sistema de alerta que se encargue de vigilar el comportamiento del Sol que nos avise con el suficiente tiempo para poder implementar las medidas necesarias que atenúen las consecuencias ya comentadas.

Debido a la importancia que tienen las ráfagas de radio solar, la comunidad científica ha ido recopilando y almacenando las mediciones realizadas sobre la radiación solar, para realizar un análisis, a posteriori, con el objetivo de clasificar e identificar, manualmente, las distintas anomalías encontradas. De esta forma nace e-Callisto, un proyecto internacional que recopila la actividad solar desde 2002 y la almacena en un servicio de libre acceso. En la actualidad existen más de 60 radiotelescopios que realizan una cobertura completa y constante del Sol en cualquier momento del día. Para nuestro trabajo, utilizaremos los datos de dicho proyecto.

Como se puede suponer, la tarea de identificación de las anomalías existentes en los radioespectrómetros solares es bastante compleja, por eso se requiere de una persona que realice este trabajo, analizando todas las mediciones obtenidas.

Actualmente, esta tarea es realizada manualmente, revisando las imágenes una a una, por Christian Monstein, responsable de e-Callisto. Sin embargo, gracias al importante desarrollo de los últimos años en el campo de la inteligencia artificial, se espera que se pueda automatizar este proceso utilizando una red neuronal.

La solución que se va a implementar en este trabajo, con el objetivo de resolver el problema planteado, es la siguiente: se va a desarrollar un autoencoder que entrenaremos utilizando los datos recogidos de una estación de las que se encuentran activas de la red de e-Callisto. Los datos de entrenamiento utilizados para entrenar la red neuronal serán datos sin anomalías, de esta manera, el autoencoder aprenderá a representar solamente datos sin anomalías.

Con el objetivo de saber si una imagen contiene un evento, calcularemos el error entre la imagen original y la imagen de salida del autoencoder. Como el autoencoder no ha aprendido a representar imágenes con anomalías, la diferencia entre las dos imágenes será considerablemente mayor que cuando no haya un evento en la imagen. Para determinar cuando la diferencia es mayor o no, estableceremos un “umbral de detección” que se calculará en función de los errores encontrados para el conjunto de validación de las imágenes con anomalías. Por lo tanto, como nuestro autoencoder no es capaz de regenerar patrones de las imágenes con eventos, porque no ha sido entrenado con ellos, la diferencia entre la imagen original y la imagen generada será notablemente superior que en los casos de las imágenes sin eventos.

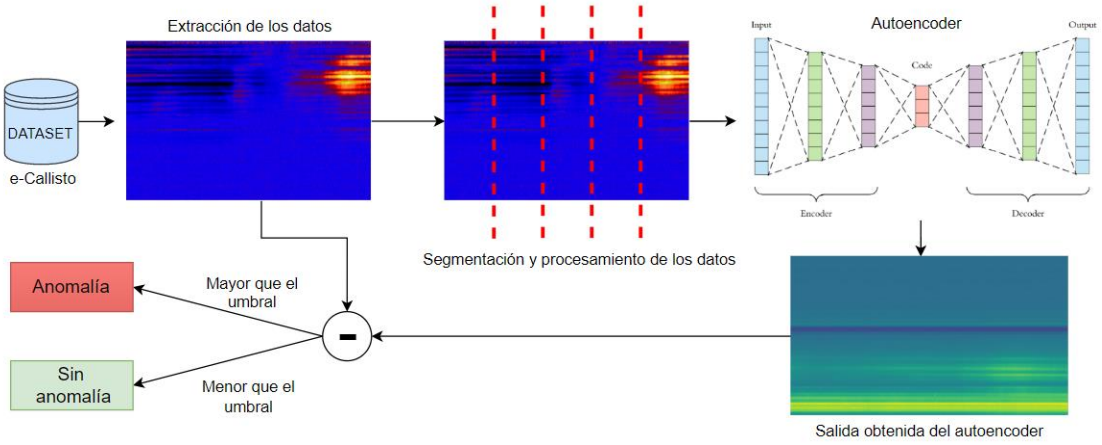


Figura 1.1: Esquema general de la arquitectura de nuestro proyecto

Tendremos que estudiar e investigar qué tipo de arquitectura del autoencoder es mejor para resolver este problema y con cuál obtenemos los mejores resultados.

1.1 Objetivos a conseguir

Debido a la gran importancia que tiene la radiación solar en nuestras vidas, aparece la necesidad de detectar la existencia de anomalías existentes en las mediciones de la radiación solar, que será el objetivo principal de este proyecto. Para lograrlo, se va a realizar un tipo de red neuronal llamado autoencoder que se entrenará para conseguir dicho objetivo.

Los datos que vamos a usar para nuestro proyecto son los obtenidos por la red científica internacional e-Callisto que monitoriza la actividad energética solar. E-Callisto es una red de antenas conectadas por Internet y repartidas por todo el mundo que miden las emisiones de ondas de radio solares. Estas mediciones son representadas por imágenes recogidas por espectrómetros solares, que podemos descargar desde la página oficial de e-Callisto.

Gracias a la cobertura realizada por los países que forman parte del proyecto, es posible hacer un seguimiento de la actividad solar 24 horas al día usando la red de antenas situada por todo el planeta. El objetivo de la red es entender las causas de los sucesos que ocurren en la atmósfera solar tales como las fulguraciones o las emisiones coronales de masa.

Este trabajo consistirá en programar la red neuronal y encontrar la mejor configuración para que detecte correctamente las anomalías existentes en los radioespectrómetros solares usando redes neuronales convolucionales (CNN).

1.2 Organización

La estructura de este trabajo es la que detallamos a continuación:

- **Capítulo 1**: Introducción: en este capítulo introduciremos el problema, explicaremos el contexto del tema y detallaremos los objetivos del proyecto.

- **Capítulo 2**: Estado del arte: en este capítulo detallaremos los estudios previos realizados para este problema, definiremos lo que es una anomalía, sus tipos y las técnicas utilizadas en la detección de anomalías.

- **Capítulo 3**: Estudio del Machine Learning y los autoencoders: en este punto explicaremos el tipo de red neuronal que vamos a utilizar y las herramientas que emplearemos para desarrollarla.

- **Capítulo 4:** Obtención y preprocesado de los datos: explicaremos la fuente de datos que utilizaremos, el proceso seguido para poder obtenerlos y qué transformaciones realizaremos para utilizarlos como entrada en nuestra red neuronal.

- **Capítulo 5**: Desarrollo de la red neuronal: en este capítulo, realizaremos el desarrollo de la red neuronal y explicaremos las distintas arquitecturas implementadas. También, compararemos y probaremos el rendimiento de cada modelo y valoraremos su efectividad para ver cuál es la arquitectura que mejor detecta las anomalías causadas por la radiación solar.

- **Capítulo 6**: Conclusiones y trabajo futuro: en este punto comentaremos los resultados que hemos obtenido del trabajo realizado. También detallaremos las posibles futuras líneas de investigación.

Capítulo 2

Estado del arte

En este campo ya se han realizado investigaciones previas para tratar de solucionar el problema de la detección de anomalías, objetivo que tenemos para nuestro proyecto. En primer lugar, debemos destacar el trabajo publicado en 2018 en el que se analizan las técnicas en el procesamiento de imágenes utilizadas para la detección de eventos solares [2].

En 2019, se publicó un artículo en el que se usaba un sistema de visión artificial con el objetivo de detectar anomalías en los radios espectrómetros solares [3]. Este proyecto fue creado para detectar automáticamente la presencia de anomalías de la radiación solar en los radios espectrómetros solares utilizando MATLAB. Para ello, se procesaron 1491 imágenes de múltiples estaciones diferentes del 11-02-2014, obteniendo un porcentaje de éxito en la detección de eventos solares del 89%.

También en 2019, se realizó una investigación sobre la detección automática de anomalías solares mediante un método estadístico [4]. El algoritmo consiste en primer lugar, en eliminar el fondo de los radios espectrómetros para quedarse con las características más importantes de la imagen. Después, se binariza la imagen y se calcula el ASI (índice de inclinación del área). Para determinar la existencia de una anomalía, se calcula la probabilidad de que exista un evento solar para un determinado valor de ASI dado. Los datos utilizados han sido las imágenes recogidas por el observatorio de Gauribidanur en India durante 2013 y 2014. La precisión obtenida para la detección de anomalías utilizando este método es de un 50%.

Recientemente, en junio de 2022, se ha llevado a cabo un estudio para la detección automática de las anomalías solares utilizando redes neuronales profundas [5]. Este estudio utiliza las imágenes recogidas por la estación de Glasgow de los días 22 y 24 de mayo de 2021, dos días en los que el Sol tuvo bastante actividad. La red neuronal utilizada ha sido AlexNet, una red neuronal convolucional, con la que se consiguió una precisión del 88%.

2.1 Detección de anomalías

En el entorno del análisis de datos, una anomalía es una observación anormal que cuenta con características que lo diferencian del resto de elementos del conjunto de datos. Las anomalías surgen por cambios en el comportamiento de los sistemas, errores humanos, errores mecánicos o desviaciones naturales en el conjunto de datos. En la actualidad la detección de anomalías es un problema capital porque comúnmente suelen indicar información crítica o valiosa sobre nuestro conjunto de datos. Estas anomalías pueden ser fallos del sistema, por lo que una rápida detección puede evitar consecuencias potencialmente catastróficas. Por ejemplo, estas posibles anomalías pueden ser incertidumbres médicas, errores de texto o fallos relacionados con el fraude bancario [6].

Podemos clasificar las anomalías en las siguientes tres categorías [7]:

- Anomalías puntuales: si una instancia de un dato individual se puede considerar como anómalo debido a que tiene un valor muy distinto que el resto de los elementos del conjunto de datos, la instancia la denominaremos como una anomalía puntual. Este tipo de anomalías es el más simple de todos y es el foco de la mayoría de las investigaciones en detección de anomalías. Por ejemplo, en la Figura 2.1, los puntos A1 y A2 se encuentran fuera del límite de las regiones “normales” y, por lo tanto, son anomalías puntuales ya que son distintos que el resto de los puntos.

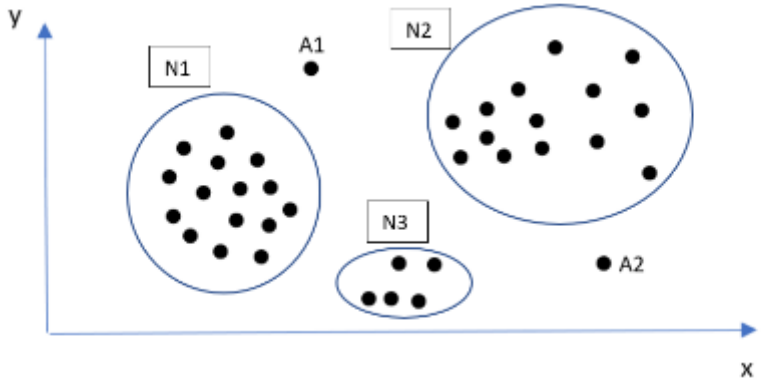


Figura 2.1: Gráfico bidimensional de anomalías puntuales

- Anomalías contextuales: son los datos que se consideran anomalías dentro de un contexto específico de un conjunto de datos. Dicho contexto debe ser inducido por la estructura del conjunto de datos y tiene que ser una especificación del problema. Para cada instancia de los datos hay que definir un atributo de comportamiento y un atributo contextual. Una instancia de datos puede ser considerada como una anomalía contextual en un determinado contexto, pero la misma instancia podría considerarse como normal en un contexto diferente. Este tipo de anomalías se han estudiado más comúnmente en datos de series temporales [8] y datos espaciales [9].

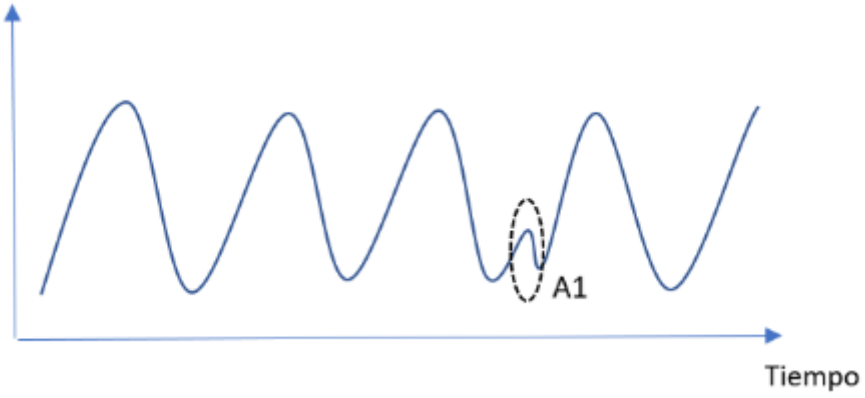


Figura 2.2: Gráfico con anomalía contextual

- Anomalías colectivas: si una colección de instancias de datos relacionados es anómala con respecto a todo el conjunto de datos, se denomina anomalía colectiva. Las instancias individuales de una anomalía colectiva pueden no ser anomalías por sí mismas, pero su aparición conjunta como colección es anómala. En la Figura 2.3 se muestra la salida de un electrocardiograma humano donde encontramos un ejemplo de este tipo de anomalía. La parte resaltada en rojo indica la existencia de una anomalía colectiva ya que existe el mismo valor para un tiempo particularmente largo, a pesar de que la cuantía de ese valor no es una anomalía.

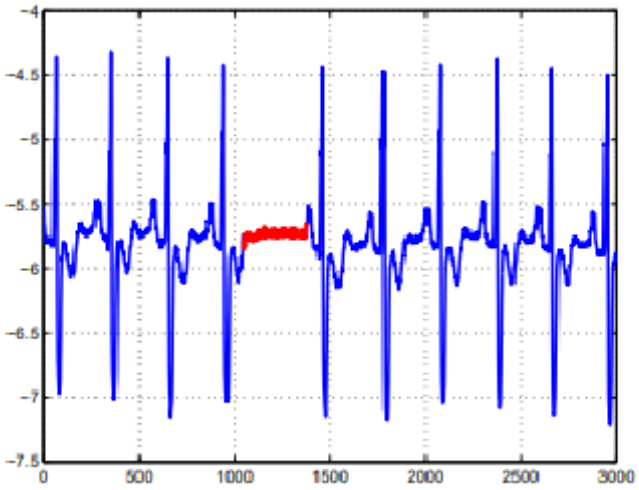


Figura 2.3: Salida de electrocardiograma humano con anomalía colectiva

A pesar de que podemos realizar la categorización de las anomalías, el proceso más complicado es su identificación. Las anomalías se consideran una variación del comportamiento "normal" de los datos, por lo que identificar qué datos tienen ese comportamiento distinto que, además, depende del contexto en el que se encuentren esos datos, no es una tarea sencilla. Por este motivo, el tratamiento de los datos anómalos debemos hacerlo con cuidado porque habrá anomalías que se pueden descartar debido a que son errores de medición y en otros casos serán datos que nos aporten información muy útil sobre el campo estudiado.

2.2 Técnicas y aplicaciones de la detección de anomalías

Con el objetivo de detectar las características jerárquicas de los datos sin utilizar métodos "manuales", podemos utilizar métodos supervisados, semisupervisados, no supervisados, redes neuronales de una sola clase (OC-NN) o usando modelos híbridos [10].

El uso de estos métodos para detectar anomalías se basa en los siguientes puntos:

- Tipo del entrenamiento: dependiendo del objetivo del entrenamiento utilizaremos unas técnicas u otras.
- El tipo de datos recibidos como entrada: la elección de la arquitectura de la red neuronal depende principalmente del tipo de datos de entrada, su naturaleza, el número de características que tienen… En la Tabla 1.1 podemos observar cual es la arquitectura más común utilizada en función del tipo de datos de entrada.
- El tipo de las anomalías: en función del tipo de anomalías del problema a abordar, se usará un método u otro. Como ya hemos comentado anteriormente, los tipos existentes son: anomalías puntuales, anomalías contextuales y anomalías colectivas.
- La existencia de datos etiquetados: las etiquetas de los datos indican si es una instancia normal o atípica y como las anomalías, normalmente, son difíciles de detectar, no es habitual que los datos estén etiquetados.
- Resultado de los métodos de detección de anomalías: como no todos los métodos identifican de la misma manera las anomalías, las salidas producidas por cada uno de ellos influye en la elección del método a utilizar.

Tipo de datos	Arquitectura de detección de anomalías	Ejemplos de datos
Datos no secuenciales	CNN, Autoencoder	Imágenes, datos
Datos secuenciales	CNN, RNN, LSTM	Video, Series Temporales

Tabla 2.1: Arquitecturas utilizadas para la detección de anomalías en función del tipo de datos de entrada.

Los métodos de detección de anomalías son implementados en un rango amplio de sectores. Por ejemplo, algunos de los campos de aplicación en los que estos métodos se utilizan serían:

1. Astronomía: en este campo se han usado técnicas de detección de anomalías ya que, en la adquisición de datos astronómicos, los datos identificados como anómalos pueden ser producidos por nuevos elementos de interés para el campo que se está estudiando. Algunos ejemplos son: la detección de anomalías en fenómenos transitorios [11] o para detectar datos anómalos en imágenes ópticas de galaxias [12]

2. Industria: este tipo de técnicas se utilizan en la industria para detectar fallos en los procesos industriales durante el control de calidad. Por ejemplo: la detección de anomalías en el proceso de una planta de fabricación de metal [13].

3. Salud: en este campo la inteligencia artificial es de gran ayuda porque en bastantes campos que tienen relación con la salud humana requieren de estas técnicas de detección de anomalías en los procesos de diagnóstico y detección de enfermedades. Además, es altamente probable que en un futuro la inteligencia artificial dé un mayor soporte debido a que cada vez el volumen de datos que se manejan de los pacientes es más extenso. Por ejemplo: para detectar anomalías en la frecuencia cardíaca [14], para detectar planes de tratamiento de radioterapia erróneos [15] o bien para la detección de melanoma en la piel [16].

4. Finanzas: en el mundo de las finanzas debido a la gran cantidad de datos almacenados de las transacciones, ingresos, transferencias o clientes, también existen diversas aplicaciones tales como: detectar transacciones fraudulentas [17] o detectar anomalías en operaciones bancarias [18].

Capítulo 3

Estudio del Machine Learning y los autoencoders

En este apartado vamos a explicar el funcionamiento de las redes neuronales, centrándonos en las redes neuronales convolucionales que será el tipo de red de nuestro modelo predictivo, ya que será un autoencoder. Por último, comentaremos las herramientas utilizadas y la solución propuesta para resolver el problema planteado en este trabajo.

3.1 Inteligencia Artificial y Machine Learning

La inteligencia artificial es una rama de las ciencias de la computación que tiene como objetivo desarrollar sistemas capaces de imitar la inteligencia humana para resolver problemas y aprender a partir de los datos que recopilan.

La primera vez que se intenta medir la capacidad de una máquina para desempeñar un comportamiento inteligente parecido al de los seres humanos, fue en 1950 cuando el matemático Alan Turing presentó el Test de Turing, una prueba para evaluar dicha capacidad. Este test fue propuesto en un artículo llamado "Computing Machinery and Intelligence" y exponía que una máquina posee verdadera inteligencia, si es capaz de responder imitando los comportamientos humanos ante ciertas condiciones determinadas. La prueba original consistía en que un humano realizaba una serie de preguntas, mientras que una máquina y otro ser humano debían contestarlas. Las preguntas debían ser sobre un tema y un contexto concreto. Al finalizar el interrogatorio, el humano que preguntaba debía decidir qué respuestas eran de la máquina y cuáles del ser humano. En caso de que no pudiera identificar más de la mitad de las respuestas, se consideraba que la máquina contaba con un comportamiento inteligente.

El concepto de inteligencia artificial aparece por primera vez en 1956 en la conferencia de Dartmouth, que es considerado el evento germen de la IA donde participaron varios científicos y matemáticos encabezados por John McCarthy [19]. Sin embargo, hasta la década de los noventa, que se popularizaron los computadores de uso personal no se realizaron grandes avances, produciéndose un estancamiento en el estudio de esta disciplina. No obstante, a partir del año 2000, la IA no ha dejado de evolucionar gracias a los avances tecnológicos y al interés de numerosas empresas en desarrollar esta disciplina de las ciencias de la computación.

Además, ya ha quedado demostrado que la IA es capaz de mejorar al ser humano en ciertas tareas, como por ejemplo: en 1997 el campeón mundial de ajedrez perdió contra una supercomputadora autónoma desarrollada por IBM llamada Deep Blue. En 2011, también IBM, desarrolló una computadora llamada "Watson" que consiguió ganar en un concurso de preguntas y respuestas estadounidense a los dos máximos campeones humanos.

Un ejemplo de un avance logrado gracias a la inteligencia artificial se ha conseguido predecir la estructura de casi todas las proteínas conocidas en el planeta. El proyecto, que se llama "AlphaFold", permite a los investigadores detectar tendencias y patrones en la estructura de las proteínas que servirán para desarrollar nuevos tratamientos para enfermedades como el cáncer o el alzhéimer.

En 1969, el biólogo estadounidense Cyrus Levinthal calculó que se necesitarían más de 14.000 millones de años, que es el tiempo transcurrido desde el origen del universo, para averiguar las posibles configuraciones de una proteína partiendo de su secuencia de aminoácidos. Este proceso normalmente se realizaba mediante experimentos en el laboratorio, los cuales son más costosos y largos, mientras que el sistema desarrollado por DeepMind logra hacerlo en escasos minutos. Ad

Además, ofrece toda la información de manera abierta y gratuita a través de una base de datos con la que se abordarán problemas como enfermedades raras, entender procesos naturales o luchar contra la contaminación.

El machine learning o aprendizaje automático es una rama de la inteligencia artificial que tiene como objetivo desarrollar métodos para que las máquinas aprendan. Definiremos el aprendizaje como el proceso en el que un agente mejora su desempeño en una determinada tarea gracias a la experiencia por medio del uso de datos. Para ello, gracias a la utilización de algoritmos que proporcionan la capacidad de identificar patrones en grandes conjuntos de datos, las computadoras son capaces de completar tareas específicas para las que han sido entrenadas de manera autónoma.

Estos sistemas tienen la capacidad de aprender con el tiempo, sin necesidad de intervención humana. Esto supone una gran ventaja en el momento de controlar una enorme cantidad de datos de una manera más efectiva. Los algoritmos de aprendizaje automático están diseñados para clasificar datos, identificar patrones y tomar decisiones.

Como un modelo de machine learning se sirve de evidencias en forma de datos utilizados para comprender los patrones existentes, usaremos un alto número de datos con los que entrenaremos al modelo. Una vez finalizado el entrenamiento, seremos capaces de generalizar el comportamiento observado y predecir el valor en el futuro de los datos basándonos en su comportamiento anterior.

El aprendizaje automático se utiliza en numerosos campos como por ejemplo para realizar diagnósticos médicos, reconocimiento de voz y del lenguaje escrito, detección de fraude, recomendaciones personalizadas, vehículos inteligentes, motores de búsqueda o ciberseguridad.

3.1.1 Tipos de algoritmos de Machine Learning

Existen cuatro tipos de aprendizaje automático: supervisado, no supervisado, semisupervisado y por refuerzo. En función de los datos disponibles y el tipo de problema que tenemos que abordar usaremos un tipo u otro.

- Aprendizaje supervisado: es el tipo de aprendizaje más común y se basa en el entrenamiento de los computadores con datos etiquetados previamente. De esta manera, se les permite realizar predicciones para cualquier objeto de entrada. Los datos de entrenamiento utilizados son un par de valores en el que un elemento es el dato de entrada y el otro, los resultados que deseamos obtener. En función del tipo de salida del algoritmo, podemos definir el problema: si la salida es un valor numérico, el problema es de regresión, en cambio, si la salida es una etiqueta, el problema será de clasificación. Un ejemplo de este tipo de aprendizaje sería un modelo que intentara detectar si un correo electrónico es spam o no. Los datos de entrenamiento estarían etiquetados ya previamente y el modelo intentaría aprender en base a las características internas de los correos cuando es un correo basura.

- Aprendizaje no supervisado: estos algoritmos no utilizan para su entrenamiento datos etiquetados, por lo que no cuentan con un conocimiento previo. Su objetivo es explorar los datos para buscar patrones y similitudes que les permitan obtener sus características más importantes. Una vez encontradas dichas características, se utilizarán para clasificar y etiquetar los datos. Un ejemplo que utilice este tipo de entrenamiento sería un modelo de reconocimiento facial, en el que su objetivo es encontrar un conjunto de patrones comunes que logren identificar a un rostro.

- Aprendizaje semisupervisado: este tipo combina el aprendizaje supervisado y el no supervisado para conseguir clasificar de manera correcta. Con respecto al tipo de datos

usado, se emplean tanto datos de entrenamiento etiquetados como no etiquetados. Normalmente, se emplean una reducida cantidad de datos etiquetados y una mayor cantidad de datos no etiquetados. El objetivo de este tipo de entrenamiento es explorar la información de los datos no etiquetados evaluando las iteraciones del algoritmo con el subconjunto de datos etiquetados. Un ejemplo de este tipo puede ser un modelo que analice conversaciones telefónicas. Para extraer características de los interlocutores como pueden ser su estado de ánimo o el motivo de la llamada, es imprescindible contar con un conjunto de datos ya etiquetados que sirvan para que el modelo aprenda los patrones existentes en las llamadas.

- Aprendizaje por refuerzo: el objetivo de este tipo de aprendizaje es que la red neuronal consiga aprender a partir de su propia experiencia. Este tipo no necesita grandes cantidades de datos de entrenamiento para funcionar, ya que funciona con una serie de indicaciones para aprender mediante prueba y error. En este sistema, el modelo se ve recompensado con un valor numérico cuando se toman las decisiones correctas. Este tipo de aprendizaje se utiliza, por ejemplo, cuando entrenamos un modelo de conducción autónoma para un coche.

3.2 Elementos de Machine Learning

Para que nuestro modelo sea capaz de aprender a predecir el comportamiento de los datos, es necesario que hayan realizado un correcto entrenamiento para que no se limiten a memorizar el comportamiento observado. Por este motivo, se compara el ajuste de cada distinto modelo para poder elegir el que mejor rendimiento proporciona. También introduciremos los conceptos de overfitting y underfitting que son las principales causas de que nuestro modelo no se ajuste como esperamos.

Una vez realizado el proceso de entrenamiento de la red neuronal, tendremos que evaluarlo para verificar si nuestro modelo es capaz de hacer predicciones correctamente. Para este proceso se utilizan las métricas que nos servirán para comparar el rendimiento de la red neuronal y poder identificar el mejor modelo. Este paso es muy importante debido a que, si no se hace una evaluación adecuada del modelo, es posible que cuando se realicen predicciones de datos no vistos por la red neuronal, se obtengan malas predicciones.

Nuestro programa al final va a clasificar una imagen en función de si detecta una anomalía o no. Por lo tanto, los errores posibles ocurren si identifica incorrectamente una imagen normal como anómala o si una imagen anómala la identifica como si no tuviera eventos. El objetivo de nuestro programa será intentar reducir al mínimo posible los falsos negativos porque, como ya hemos comentado, actualmente se revisan todas las imágenes manualmente una a una, por lo que la finalidad del programa será disminuir el número de imágenes a revisar.

3.2.1 Métricas de clasificación

Los algoritmos de clasificación se utilizan cuando el resultado a predecir es una etiqueta discreta, es decir, cuando la respuesta del problema se encuentra en un conjunto finito de resultados posibles. Si solo hay dos clases objetivos, el algoritmo será un clasificador binario, en caso de que haya más clases, será un clasificador multicategoría. Un ejemplo de este tipo de algoritmos es un modelo que estime si un cliente comprará un producto o no.

Para evaluar un modelo de clasificación, que devuelve como salida si un dato pertenece a una clase o no, comparamos la salida obtenida con la esperada y la clasificamos en una de estas cuatro opciones:

- **TP (True Positive):** cuando los valores son clasificados por el algoritmo como positivos y realmente son positivos.

- **FP (False Positive):** cuando los valores son clasificados por el algoritmo como positivos y realmente son negativos.
- **TN (True Negative):** cuando los valores son clasificados por el algoritmo como negativos y realmente son negativos.
- **FN (False Negative):** cuando los valores son clasificados por el algoritmo como negativos y realmente son positivos.

Estas opciones se recogen en una tabla llamada matriz de confusión.

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (TP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (TN)

Tabla 3.1: Matriz de confusión

Algunas métricas que se pueden calcular utilizando los valores de la matriz de confusión son:

- Accuracy: mide la proporción de los casos que el modelo ha acertado con la predicción, tanto positivos como negativos. Representa el número de predicciones correctas en relación con el total.

$$\frac{TP + TN}{TP + FP + TN + FN}$$

- Precisión: esta métrica mide la cantidad de cuantos valores clasificados como positivos son realmente positivos. Es una medida que representa la proporción de los datos positivos en relación con el total de predicciones.

$$\frac{TP}{TP + TN}$$

- Exhaustividad: es la proporción de cuantos elementos positivos se han identificado correctamente. Esta medida representa la cantidad de datos positivos que son de verdad positivos en relación con el total de predicciones correctas.

$$\frac{TP}{TP + FN}$$

- Especificidad: mide la capacidad de nuestro modelo para identificar los casos negativos. Calcula la proporción de los casos negativos detectados con los casos negativos totales.

$$\frac{TN}{FP + TN}$$

- F1 Score: sirve para medir la precisión del modelo. Es una media armónica que se calcula combinando los valores de la exhaustividad y la precisión.

$$\frac{2 * Precisión * Exhaustividad}{Precisión + Exhaustividad}$$

3.2.2 Métricas de regresión

Un algoritmo de regresión devuelve como salida un valor numérico dentro de un conjunto infinito de posibles resultados. La salida del modelo queda determinada por los valores de una serie de variables introducidos en la entrada del modelo, en vez de estar limitado a un conjunto de etiquetas. Este tipo de algoritmo es útil para predecir productos que son continuos. Un ejemplo de este tipo de algoritmo es un modelo que intente estimar el precio de las viviendas.

Como este tipo de algoritmos predicen un valor numérico, no se pueden utilizar las métricas de clasificación para evaluar su rendimiento. Por este motivo, existen métricas de error diseñadas para evaluar este tipo de modelos. Existen una gran cantidad de métricas de error usadas para evaluar el desempeño de un algoritmo de regresión, sin embargo, las más utilizadas son:

- Error absoluto medio (MAE): es una medida que se calcula como la diferencia en términos absolutos entre dos variables continuas. Con esta métrica la puntuación de la medida aumenta linealmente con los incrementos del error y las unidades del error coinciden con las del valor objetivo que se quiere predecir. Matemáticamente lo podemos definir en una fórmula donde "n" es el número de datos del conjunto de entrenamiento, y_i son las predicciones del modelo **y** x_i el valor real esperado.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i|$$

- Error cuadrático medio (MSE): se calcula como el promedio de los errores al cuadrado entre los datos estimados y los datos reales. Esta métrica a pesar de ser la más utilizada para problemas de regresión, no es muy intuitiva debido a que su valor es error medio de los datos al cuadrado.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2$$

- Raíz del error cuadrático medio (RMSE): esta medida se calcula como la raíz cuadrada de las diferencias entre los valores observados y los valores previstos. Se puede calcular siguiendo la siguiente fórmula:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2}$$

3.2.3 Overfitting y underfitting

En el campo del machine learning, el overfitting es un fenómeno que provoca un bajo porcentaje de acierto en los resultados del modelo. Esto se produce porque se ha sobreentrenado la red neuronal con un conjunto de datos para los que se conoce el resultado deseado, por lo que el modelo queda ajustado a unas características muy concretas de los datos de entrenamiento. Al final lo que ocurre es que memoriza una determinada cantidad de ejemplos y aprende de manera muy precisa sus propiedades, en vez de aprender a identificar sus características. Como consecuencia de ello, realizará malas predicciones cuando se introduzcan datos nuevos, como vemos en la Figura 3.1. En esta imagen vemos una línea azul que representa un modelo sobreentrenado y una línea negra que representa un modelo regularizado. Aunque la línea azul se ajusta mejor a los datos de entrenamiento, va a tener una tasa de error más alta en los nuevos datos no vistos, en comparación con la línea negra.

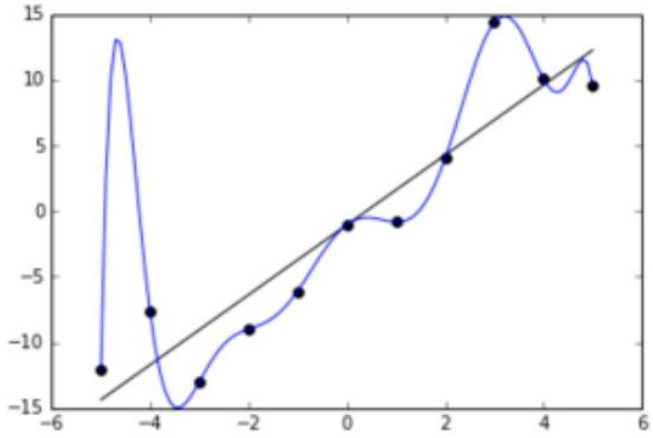


Figura 3.1: Representación gráfica de un modelo sobreentrenado

Al contrario que en el overfitting, el underfitting ocurre cuando el modelo no ha sido adecuadamente entrenado y, por lo tanto, no es capaz de identificar los patrones de los datos. Debido a esto, los resultados de sus predicciones no serán correctos. El underfitting puede ser causado por un entrenamiento demasiado pobre o por no utilizar suficientes datos de entrenamiento.

El objetivo del proceso de entrenamiento es encontrar una configuración del modelo capaz de predecir el resultado de unos datos dados, generalizando a partir de lo aprendido con los datos de entrenamiento. De esta manera, se podrán realizar correctamente predicciones de datos desconocidos para la red neuronal.

3.3 Redes neuronales

Una red neuronal es un modelo computacional compuesto por neuronas artificiales conectadas entre sí, que procesan datos de una manera inspirada en la manera en la que lo hace el cerebro humano.

El perceptrón es la unidad básica del campo de las redes neuronales y corresponde a la estructura a la cual llamamos neuronas. Existen dos tipos de perceptrones: el perceptrón simple y el multicapa. El perceptrón simple está formado por una única capa de entrada a la que se le introduce un dato y otra de salida. Para calcular el output al dato introducido se le multiplicará por un peso “W” y se le sumará después un valor “b”. El resultado obtenido se introducirá en una función de activación y esa será la salida del perceptrón. Sin embargo, el problema del perceptrón simple es que sólo es capaz de resolver problemas de clasificación binaria.

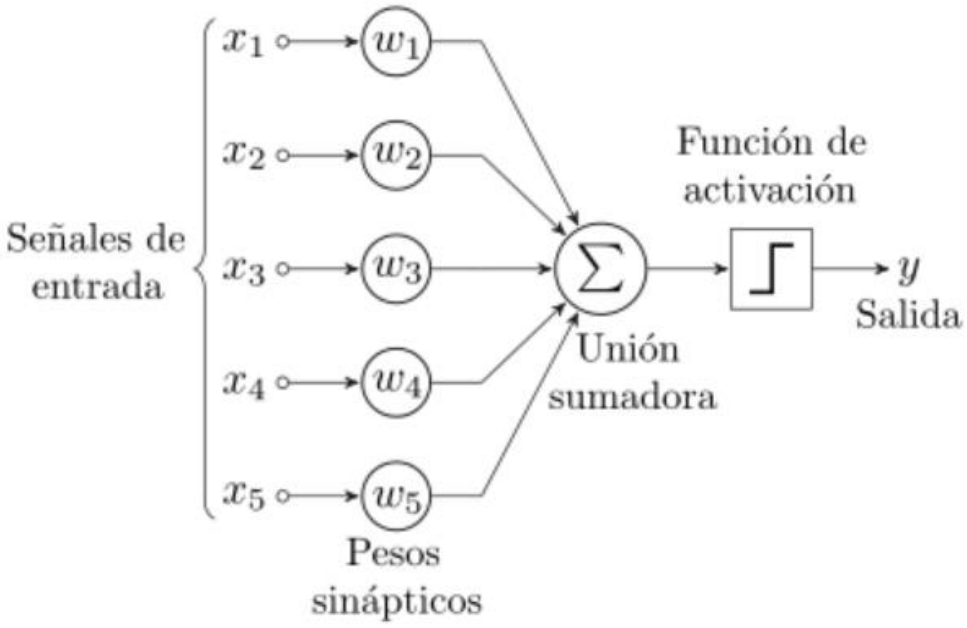


Figura 3.2: Arquitectura del perceptrón simple

El perceptrón multicapa tiene la misma estructura que el perceptrón simple, pero está formado por varias neuronas situadas en la capa oculta, entre la capa de entrada y la de salida. Cada neurona está conectada con la neurona de la siguiente capa y cada una cuenta con una función de activación diferente. A diferencia que el perceptrón simple, el multicapa puede clasificar los datos de entrada en múltiples categorías.

Normalmente, una red neuronal se estructura en tres capas: una capa de entrada, una o varias capas ocultas donde se procesa la información y una capa de salida. Los datos de entrada se reciben en la primera capa y se propagan los valores a través de las neuronas hasta la capa de salida que devuelve un resultado. Cada capa agrupa un conjunto de neuronas artificiales y el conjunto de una o más capas constituye la red neuronal.

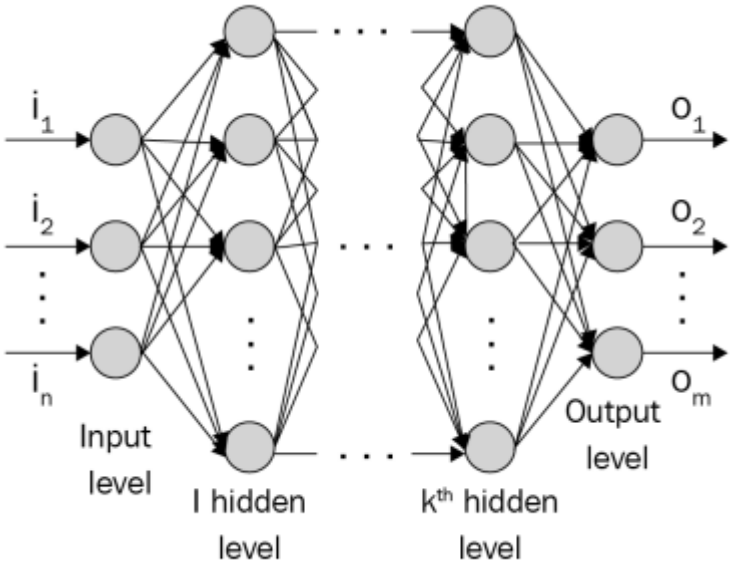


Figura 3.3: Arquitectura de una red neuronal multicapa

Estos métodos en lugar de ser programados explícitamente aprenden por sí solos basándose en la experiencia sin un conocimiento previo de las relaciones entre las variables de entrada y de salida. Además, destacan en su capacidad para aprender y modelar las relaciones entre los datos de entrada y salida que son complejos y no lineales.

Las neuronas de la red neuronal están interconectadas por enlaces que modifican el valor de la salida de la neurona anterior multiplicándolo por un valor llamado peso. Estos pesos aumentan o disminuye la influencia de cada neurona y se van modificando durante el aprendizaje.

A la salida de cada neurona, puede existir una función de activación que modifica el resultado o establece un límite que no puede sobrepasar el valor dado para pasar a la siguiente neurona. Por lo tanto, una función de activación es una función matemática que recibe como entrada la información generada por la combinación lineal de los pesos y las entradas y transfiere el resultado por las conexiones de salida. Cada capa del modelo cuenta con una función de activación que usarán todas sus neuronas.

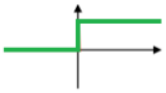
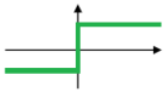
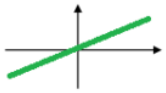
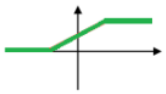
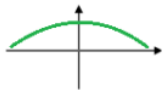
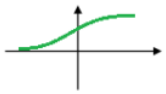
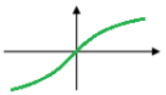
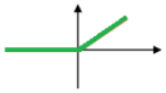
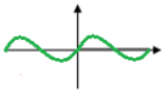
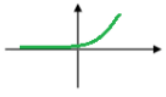
TIPO DE FUNCIÓN DE ACTIVACIÓN	ECUACIÓN	EJEMPLO O MODELO	GRÁFICO
Función "Step" o Heaviside	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	PERCEPTRÓN	
Función Signo	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	PERCEPTRÓN	
Función Lineal	$\phi(z) = z$	ADALINE	
Función lineal definida a trozos	$= \begin{cases} 1 & z \geq \frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} < z < \frac{1}{2} \\ 0 & z \leq -\frac{1}{2} \end{cases}$	MÁQUINAS DE VECCTOR SOPORTE	
Función Gaussiana	$\phi(z) = Ae^{-Bz^2}$	REDES NEURONALES RBF	
Función Sigmoide o Logística (Curva "S")	$\phi(z) = \frac{1}{1 + e^{-z}}$	REDES NEURONALES MULTICAPA	
Tangente hiperbólica	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	REDES NEURONALES MULTICAPA	
Función rectificadora o función ReLU Unidad Lineal Rectificada	$\phi(z) = max(0, z)$	REDES NEURONALES MULTICAPA	
Función Sinusoidal	$\phi(z) = A\,\mathrm{sen}(\omega z + \varphi)$	CLASIFICACIÓN DE PATRONES	
Función rectificadora suavizada (softplus)	$\phi(z) = \ln(1 + e^{z})$	REDES NEURONALES MULTICAPA	

Figura 3.4: Lista de las principales funciones de activación

Como ya hemos comentado anteriormente, una red neuronal está formada por tres capas:

1. **Capa de entrada:** en esta capa se introducen los datos que recibirá como entrada la red neuronal, por lo tanto, tendrá un formato preciso.
2. **Capa(s) oculta(s):** esta parte de la red contará con una o varias capas ocultas, cada una con una o varias neuronas, donde se procesará la información.
3. **Capa de salida:** esta capa tendrá como salida las variables que queremos predecir, por lo que tendrá un formato determinado.

3.4 Redes neuronales convolucionales (CNN)

Una red neuronal convolucional es un tipo de red neuronal artificial que cuenta con múltiples capas para calcular la salida a partir de un conjunto de datos. Las neuronas artificiales que componen esta red corresponden a campos receptivos que simulan el funcionamiento de las neuronas de la corteza visual de un cerebro biológico. Este tipo de red está diseñado para trabajar con imágenes, algunas de las aplicaciones de estas redes son: reconocimiento de patrones, clasificación y segmentación de imágenes, análisis de video, procesamiento de lenguaje natural o reconocimiento de voz.

Las redes neuronales convolucionales están formadas por múltiples capas de filtros convolucionales que pueden tener una o más dimensiones. Las capas del principio de la red tienen como objetivo extraer las características buscadas y las del final realizan la clasificación final de las características extraídas. La etapa de extracción de características se parece al proceso que siguen las células de la corteza visual del cerebro. En esta fase existen capas de neuronas convolucionales y neuronas de reducción de muestreo organizadas de manera alterna. Conforme se van procesando los datos a través de esta fase, se reduce la dimensionalidad de los datos. Esto causa que las neuronas que componen las capas lejanas sean menos sensibles a perturbaciones en los datos recibidos como entrada, pero a la vez estas son activadas por características más complejas.

La convolución consiste en una operación que se realiza a nuestros datos utilizando una matriz llamada filtro o kernel, con el objetivo de modificar y transformar algunas de las características de los datos en función de los valores del kernel. Usando distintos filtros logramos desenfocar, realzar, enfocar o detectar bordes en imágenes.

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figura 3.5: Ejemplos de filtros para detección de bordes

Los filtros son las operaciones, representados como matrices, que se aplican a nivel de píxel de las imágenes. El valor resultante de realizar la convolución a un píxel es el dado por la siguiente fórmula:

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

En la fórmula tenemos que:

- G[m,n] de “m” filas y “n” columnas es la imagen filtrada.
- h[j,k] de “j” filas y “k” columnas es el filtro.
- f[m,n] es la imagen original.

En esta operación intervienen dos matrices: la propia imagen y el filtro. Como resultado obtenemos una matriz con las mismas dimensiones que la imagen original que calculamos

desplazando el filtro por cada elemento de la matriz que representa la imagen y le damos el valor de la suma del producto de cada peso del filtro con el píxel correspondiente de la imagen.

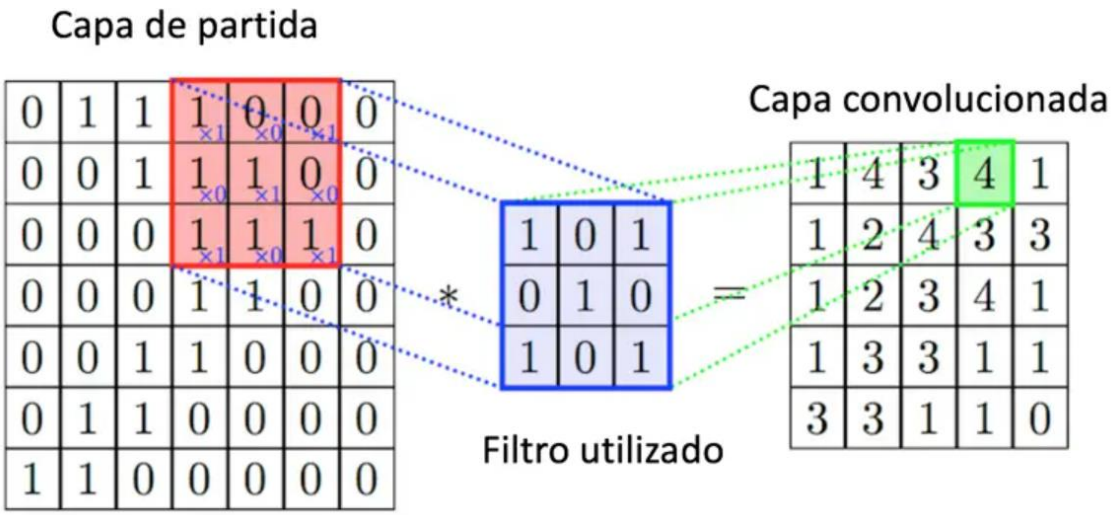


Figura 3.6: Proceso de convolución de matrices

Al usar la convolución de matrices nos permite reducir la cantidad de parámetros que necesita aprender la red neuronal porque en vez de aprender un peso por cada píxel, solo es necesario aprender los pesos que componen el filtro.

Estas redes neuronales se caracterizan porque en las primeras capas es capaz de aprender las características básicas de la imagen, por ejemplo, líneas o formas simples. En cambio, en las capas posteriores aprende a identificar elementos más complejos.

3.5 Autoencoders

Un autoencoder es un tipo de red neuronal artificial de aprendizaje no supervisado, utilizada para aprender codificaciones de datos de manera eficiente. Este tipo de red neuronal está formado por una capa de entrada, seguido de capas ocultas y una última capa de salida, que cuenta con el mismo número de nodos que la primera capa. Normalmente, las capas ocultas tienen una dimensión menor que la primera capa y que la última, lo que ocasiona un cuello de botella que provoca una reducción en la dimensionalidad de los datos.

El objetivo del autoencoder es reproducir en la última capa los mismos datos que recibe en la primera capa de la manera más aproximada posible. Para ello, aprenderá una representación para un conjunto de datos de menor dimensionalidad, ya que se reduce la dimensionalidad de los datos con el objetivo de entrenar a la red con las características más importantes de los datos.

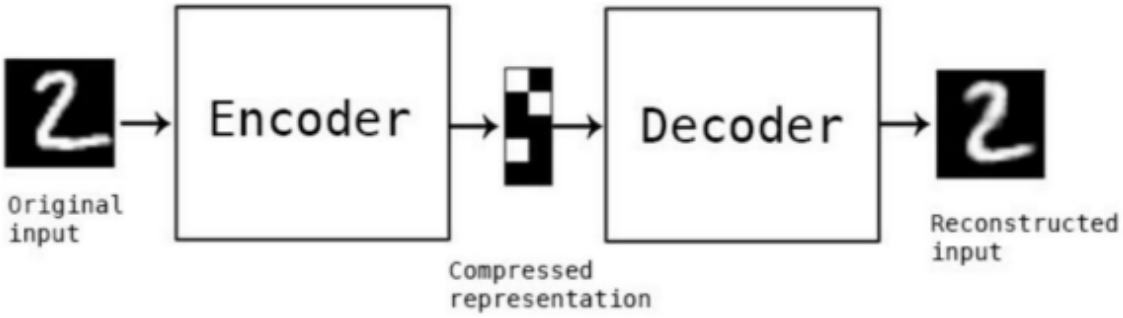


Figura 3.7: Esquema de la estructura de un autoencoder

3.5.1 Arquitectura de un autoencoder

Un autoencoder consta de dos partes principales: el encoder y el decoder. El encoder está formado por las capas situadas a la izquierda del cuello de botella y su función es aprender a codificar los datos de entrada en una representación (encoding) de menor tamaño. Esta representación se realiza extrayendo las características más importantes que permitan su posterior reconstrucción. El decoder que está formado por las capas a la derecha del cuello de botella, se encarga de convertir la codificación realizada por el encoder en una representación de las mismas dimensiones que la original. Tanto el encoder como el decoder son redes neuronales

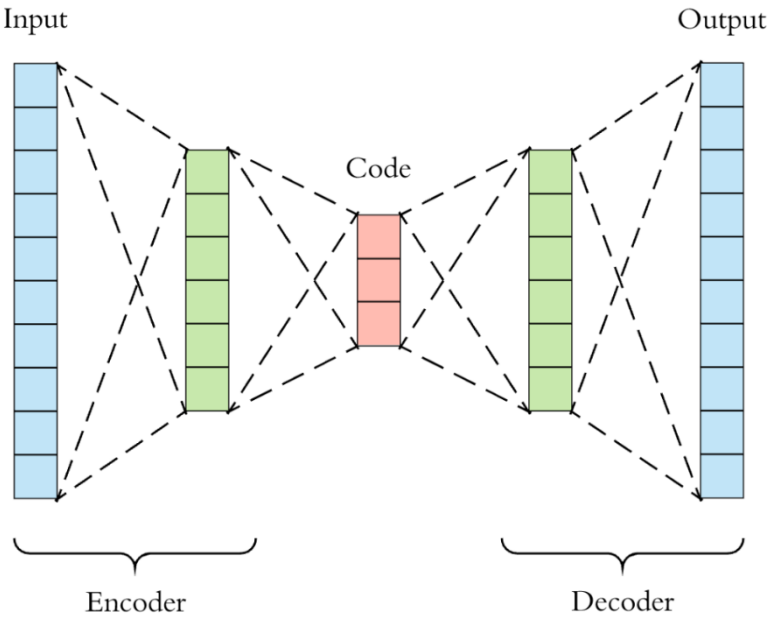


Figura 3.8: Arquitectura del autoencoder

Las funciones del encoder y el decoder tienen las siguientes características:

1. Son específicas de los datos: los autoencoders son específicos de los datos, por lo que sólo funcionarán correctamente usando datos similares a los que han recibido con antelación. Es decir, si por ejemplo, la red ha sido entrenada con imágenes de coches, no funcionará para imágenes de otro tipo porque las características aprendidas han sido para datos de imágenes de coches.
2. Tienen pérdida de información: la salida descomprimida del decoder será deteriorada en comparación con la entrada original.
3. Aprenden automáticamente a partir de ejemplos: los autoencoder aprenden a partir de ejemplos de datos, por lo que no requiere acción humana de ningún tipo, solamente contar los datos de entrenamiento apropiados.

3.5.2 Capas que componen la red y tipos de autoencoder

Tanto el encoder como el decoder están formados por capas ocultas que realizan ciertas operaciones con los datos. Encontramos los siguientes tipos:

- Capa de convolución: el objetivo de esta capa es analizar las imágenes recibidas como entrada e identificar la presencia de un conjunto de características. Esta capa crea un filtro de convolución que se aplicará a los datos de entrada de la red.
- Capa de pooling: el proceso de pooling es una operación que reducirá el tamaño de la salida resultante de las capas convolucionales con el objetivo de disminuir la potencia computacional necesaria para realizar la convolución. Gracias a esta capa, que

normalmente se aplica entre dos capas de convolución, se logra reducir el tamaño de las imágenes, preservando sus características más importantes. Existen dos tipos de pooling:

- Max pooling: que realiza la media entre los elementos, como vemos en el apartado a) de la Figura 3.9.
- Average pooling: que selecciona el mayor de ellos, como observamos en el apartado b) de la Figura 3.9.

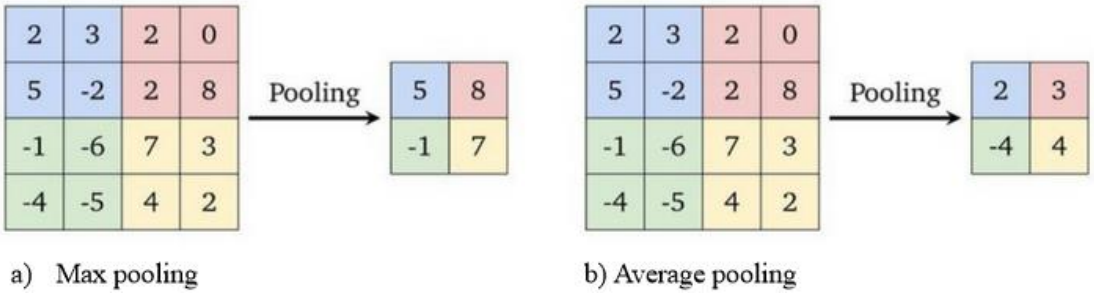


Figura 3.9: Tipos de pooling

- Capa de convolución transpuesta: este tipo de capa sirve para hacer la operación de convolución de manera inversa, es decir, de una entrada producida como salida de alguna capa de convolución, se transforma a su dimensión original.

Como hemos visto, los autoencoders funcionan recibiendo como entrada una serie de datos y devuelven una salida los más parecida posible al input inicial. Sin embargo, existen otros tipos de autoencoders:

- *Variational autoencoders*: este tipo de autoencoder parte de la misma idea que un autoencoder normal, pero en vez de comprimir la información, el objetivo que tienen es el de generar nuevos datos. Para ello, la red infiere una distribución continua de los datos empleados para el entrenamiento usando un campo de la estadística llamado inferencia variacional. El encoder calcula una media y una desviación estándar con el que se calcula una distribución Gaussiana a partir de la cual el decoder puede generar datos muy parecidos a los reales.

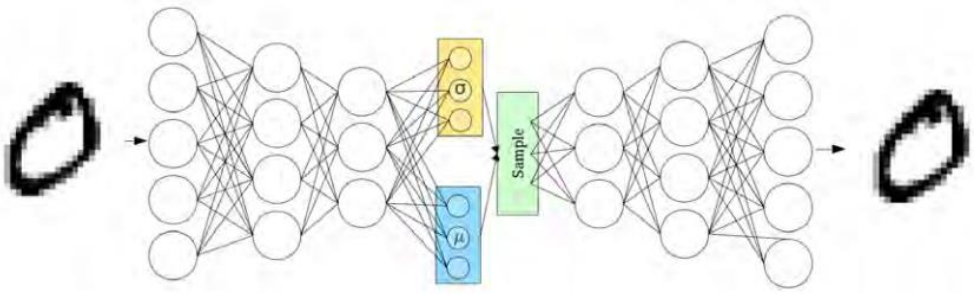
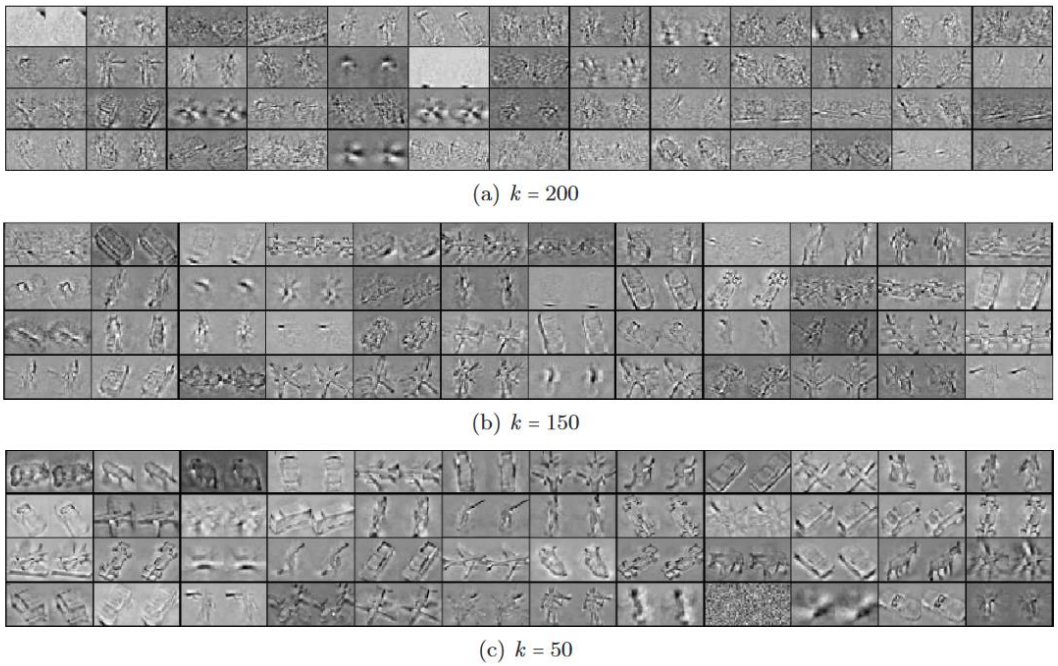


Figura 3.10: Estructura de un variational autoencoder. Podemos ver que, en la codificación, se calculan la media y la desviación estándar.

- *Sparse autoencoders:* este tipo de red se basa en la restricción de activación de los nodos que componen la red. Gracias a esta limitación, este tipo de autoencoder es capaz de aprender patrones más complejos. Para ello, se restringe la dispersión añadiendo una

variable en la función de coste que representa la especialización de las neuronas. Esta variable se denota con la letra *k*, originando los *k-sparse autoencoders.*



(a) $k = 200$

(b) $k = 150$

(c) $k = 50$

Figura 3.11: Filtros de un k-sparse autoencoder para distintos valores de k.

- *Denoising autoencoders:* esta red utiliza ruido, calculado siguiendo una distribución gaussiana aleatoria, para conseguir alterar levemente los datos de entrenamiento. Con esta arquitectura, se consigue forzar a la red para aprender patrones complejos de los datos y se evita que el autoencoder se comporte como una simple función de identidad. Como los datos de entrada han sido alterados con ruido, la red no puede devolver como output los mismos datos introducidos como entrada.

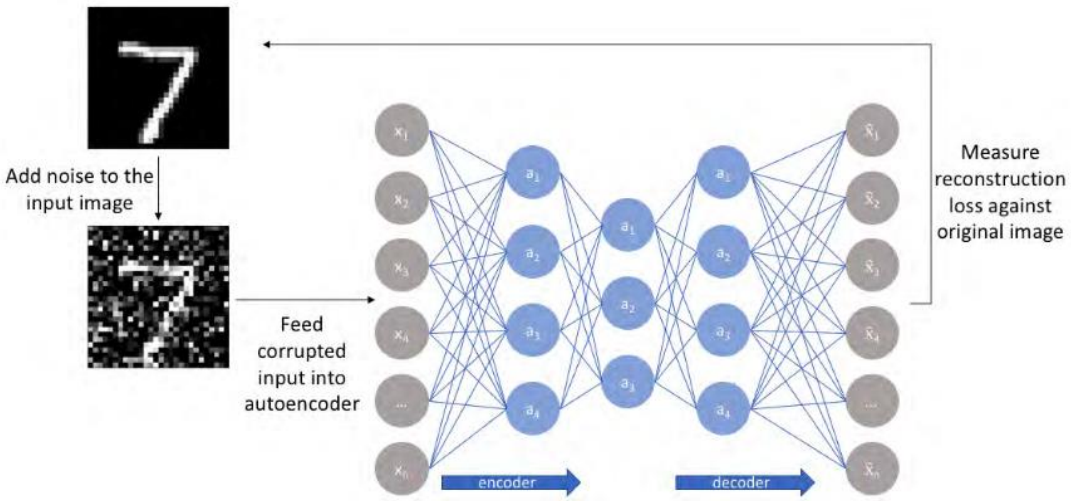


Figura 3.12: Estructura del funcionamiento de un denoising autoencoder.

3.5.3 Aplicaciones del autoencoder

Antiguamente, durante los años 80, los autoencoders tenían dos aplicaciones principales: la recuperación de información [20] y la reducción de la dimensionalidad de los datos [21]. Sin embargo, en la actualidad tienen múltiples utilidades en diversos sectores. Por ejemplo, este tipo

de red neuronal se ha aplicado a nuevos problemas tales como el reconocimiento facial [22], la detección de características, generación de imágenes [23], la detección de anomalías [24][25], la eliminación de ruido [26][27] e incluso criptografía [28].

En el caso de la detección de anomalías, el autoencoder aprende con los datos sin instancias anómalas para que de esta manera logre reproducir las características más importantes de los datos de entrenamiento. De esta manera, cuando reciben un dato anómalo, el autoencoder no reproduce de forma correcta la instancia porque no ha sido entrenado con datos anómalos.

3.6 Herramientas utilizadas

3.6.1 Python

Para este proyecto, hemos decidido desarrollarlo utilizando el lenguaje de programación Python debido a su versatilidad y flexibilidad. Este lenguaje, que es de código abierto, fue creado por Guido van Rossum a principios de los años 90 y actualmente es de los más populares del mundo.

Python es un lenguaje de alto nivel de programación interpretado, dinámico, fuertemente tipado y multiplataforma. Los paradigmas que cumple son: orientado a objetos, imperativo, funcional y reflexivo. Python es uno de los lenguajes más utilizados para programar redes neuronales debido a que cuenta con numerosas librerías y frameworks para analizar y procesar los datos como son Pandas y Numpy o para utilizar técnicas de deep learning como es Tensorflow. Una librería es un conjunto de implementaciones que se utilizan para facilitar la creación de nuevos programas.

Además, gracias a su facilidad de uso, a su sencillez de escritura y su filosofía de diseño que prima la legibilidad del código, ha provocado que exista una gran comunidad de usuarios.

3.6.2 NumPy

NumPy es una librería para el lenguaje de programación Python que proporciona soporte para crear matrices y vectores de múltiples dimensiones. Esta librería está especializada en el análisis de datos y el cálculo numérico, en particular, para cuando trabajamos con grandes volúmenes de datos. Fue lanzada en 1995 y es un software de código abierto que cuenta con una gran comunidad de colaboradores.

Para dar soporte a las matrices y vectores de múltiples dimensiones, NumPy incorpora una nueva estructura de datos llamada “ndarray” (n-dimension array), para datos del mismo tipo y de “n” dimensiones. La principal ventaja que ofrece es que el procesamiento de estos arrays es considerablemente más rápido que el de las listas originales de Python. Gracias a esto, se facilita el procesamiento de matrices de numerosas dimensiones.

Como NumPy garantiza una mayor eficiencia al trabajar con esta estructura de datos, permite un acceso más rápido para realizar lecturas y escrituras, por lo que esta librería es muy utilizada en problemas de ciencia e ingeniería.

3.6.3 Tensorflow

Tensorflow es una biblioteca de código abierto para inteligencia artificial y aprendizaje automático. Gracias a ella, es posible desarrollar y entrenar redes neuronales utilizadas para descifrar y detectar patrones y correlaciones. Esta biblioteca fue publicada el 9 de noviembre de

2015 y está desarrollada por Google. Funciona utilizando grafos de flujo de datos, en el que cada nodo representa operaciones matemáticas y cada arista, las matrices de datos (llamadas tensores) utilizadas en las operaciones. Una de sus principales ventajas es que Tensorflow ejecuta los gráficos computacionales en paralelo y, de esta manera, se puede tener un control total sobre la ejecución y es posible programar distintas operaciones en diferentes procesadores como GPU, CPU…

Esta biblioteca se utiliza para un gran abanico de aplicaciones como: en la búsqueda de voz de Google o la aplicación de fotos de Google, que implementan modelos creados con Tensorflow. Gracias a esta biblioteca, es posible compilar y entrenar modelos de machine learning de manera simple utilizando una API que, debido a su complejidad, facilitan su uso. Entre estas APIs, destaca Keras que la comentaremos en el siguiente punto.

3.6.4 Keras

Keras es una biblioteca desarrollada en Python de Redes Neuronales de código abierto. Sus principales características es que es sencilla de utilizar, extensible y modular. Es capaz de correr sobre los frameworks TensorFlow, Theano o Microsoft Cognitive Toolkit. Fue creada en 2015 por François Chollet, un ingeniero de Google, con el objetivo de ser una interfaz, en vez de una framework de machine learning.

Esta librería destaca por su simplicidad para desarrollar un modelo, debido a que con unas pocas líneas es posible crear una red neuronal. Además, Keras permite el desarrollo de las redes neuronales tradicionales, las redes neuronales convolucionales y las redes neuronales recurrentes.

Las principales características de Keras son:

- Compatibilidad con múltiples frameworks: como ya hemos comentado, es capaz de correr sobre distintos frameworks, además de que permite la posibilidad de combinarlos.
- Soporte para múltiples GPU: Keras permite que los recursos necesarios para desarrollar el aprendizaje se ejecuten en varias tarjetas gráficas de manera distribuida.
- Extensible: además de todos los módulos que ofrece Keras por defecto, también es posible crear nuevos modelos, facilitando su uso en la investigación.
- Modulable: los modelos desarrollados están compuestos por distintos bloques interconectables con relativa facilidad.

3.6.5 Jupyter Notebook

Project Jupyter es un software de código abierto de servicios para computación interactiva creado para trabajar con ciencia de datos e informática científica. Este proyecto soporta entornos de ejecución en varios lenguajes, como son: Julia, R, Ruby Python y Haskell. Entre los productos que engloba este proyecto, encontramos a Jupyter Notebook.

Jupyter Notebook es un entorno de programación interactivo basado en la web. Los documentos con los que trabaja son documentos de tipo JSON compuesto por celdas de entrada o salida con código, gráficos, fórmulas matemáticas, texto… Es una aplicación cliente servidor, que permite editar el código utilizando un navegador web. Soporta más de 40 lenguajes de programación y ofrece una plataforma donde desarrollar código en diferentes lenguajes, agrupados todos en un mismo lugar.

3.6.6 CuDNN

La biblioteca Nvidia CUDA Deep Neural Network (cuDNN) es una librería para redes neuronales profundas con aceleración de GPU. Como trabajamos con Keras y Tensorflow, como ya hemos mencionado anteriormente, este tipo de configuración permite utilizar la GPU para acelerar el proceso de entrenamiento del modelo.

CUDA (Compute Unified Device Architecture) es una plataforma de computación paralela creada por Nvidia en 2007. Proporciona un entorno de desarrollo para optimizar, desarrollar e implementar aplicaciones aceleradas por GPU, en lugar del uso tradicional realizado con la CPU. Gracias al uso de esta tecnología, podemos distribuir los cálculos en múltiples GPUs, lo que provoca que científicos e investigadores utilicen CUDA para desarrollar sus aplicaciones debido a la escalabilidad que ofrece.

CuDNN proporciona implementaciones de las redes neuronales para rutinas estándar muy ajustadas como capas de convolución, normalización, pooling, y activación. Esto permite que los desarrolladores se puedan concentrar en entrenar las redes neuronales y desarrollar el software sin tener que dedicar tiempo a ajustar el rendimiento de la GPU. Por lo tanto, el uso de esta librería permite acelerar los softwares de aprendizaje automático como Keras y Tensorflow.

3.7 Compatibilidad entre versiones

Con el objetivo de instalar el software necesario para poder entrenar el modelo utilizando la GPU, debemos tener en cuenta la compatibilidad entre las distintas versiones del software. Para ello, hemos seguido la tabla con las versiones compatibles de Python, Tensorflow, CUDA y CuDNN, que aparece en la documentación de Tensorflow para Windows [29]. Las versiones del software utilizado son las siguientes:

- Tensorflow 2.6.0
- Keras 2.6.0
- Python 3.9.5
- CUDA 11.2
- CuDNN 8.1

Capítulo 4

Obtención y preprocesado de los datos

En este punto vamos a explicar el ciclo de vida del dato, la fuente de datos usada y el proceso seguido para poder trabajar con ellos, desde su extracción hasta la creación de los datasets utilizados.

4.1 Ciclo de vida del dato

Actualmente, debido a la gran cantidad de dispositivos interconectados en todo el mundo generando cantidades enormes de datos, la gestión del ciclo de vida de los datos se ha vuelto más y más importante. Podemos definir el ciclo de vida de los datos como el periodo que engloba desde la entrada hasta la eliminación de los datos, siendo su gestión imprescindible para poder obtener un valor a partir de ellos.

Todo el proceso de la gestión del ciclo de vida de los datos debemos abordarlo desde un enfoque técnico que garantice reducir al mínimo posible su tasa de errores y maximizar el valor de los datos. Este proceso abarca distintos aspectos como son el desarrollo y mantenimiento de la base de datos, el diseño de la arquitectura, la integridad de los datos, su seguridad, su disponibilidad…

El ciclo de vida de los datos cuenta con las siguientes fases:

1. Extracción de los datos: en esta fase se identifican y se recopilan los datos que se quieren obtener. Para ello, hay que planificar cuál es la mejor forma para extraerlos y que datos son innecesarios porque no aportan valor para nuestro proyecto.
2. Almacenamiento de los datos: en función del tipo de datos, su tipo almacenamiento también se ve influenciado. Los datos estructurados, aquellos que tienen una estructura bien definida, suelen almacenarse en bases de datos relacionales. En cambio, los datos no estructurados, tienden a utilizar bases de datos no relacionales o NoSQL.
3. Preprocesamiento de los datos: en este punto se realizan determinadas tareas, como son: la limpieza de los datos, la integración de los datos, su transformación o la reducción de los datos eliminando la información duplicada o inconsistente.
4. Análisis de los datos: en el momento en el que ya tenemos los datos preparados, realizamos un análisis con el objetivo de dar una visión unificada del conjunto de datos. La utilización de modelos estadísticos puede ayudar considerablemente a aumentar la efectividad del análisis.
5. Utilización de los datos: los datos se aprovechan para obtener un conocimiento valioso a partir de ellos. Los científicos de datos utilizan distintas herramientas y métodos para realizar esta tarea. Algunas de estas herramientas son algoritmos de inteligencia artificial o modelos de aprendizaje automático.
6. Visualización de los datos: consiste en la presentación de los datos de manera ilustrada facilitando la comprensión de los datos. Gracias a la utilización de medios gráficos como grafos, diagramas o gráfica

4.2 Fuente de datos

Como ya hemos comentado, los datos utilizados serán los recogidos por la red e-Callisto que podemos encontrarlos en su página web: https://www.e-callisto.org/Data/data.html ordenados cronológicamente.

Para nuestro trabajo, usaremos los datos recopilados por la estación de Assa en Australia de los años 2020 y 2021. Hemos elegido esta estación debido a que es la que más anomalías detectó durante todo el año de 2021, como podemos observar en la Figura 4.1:

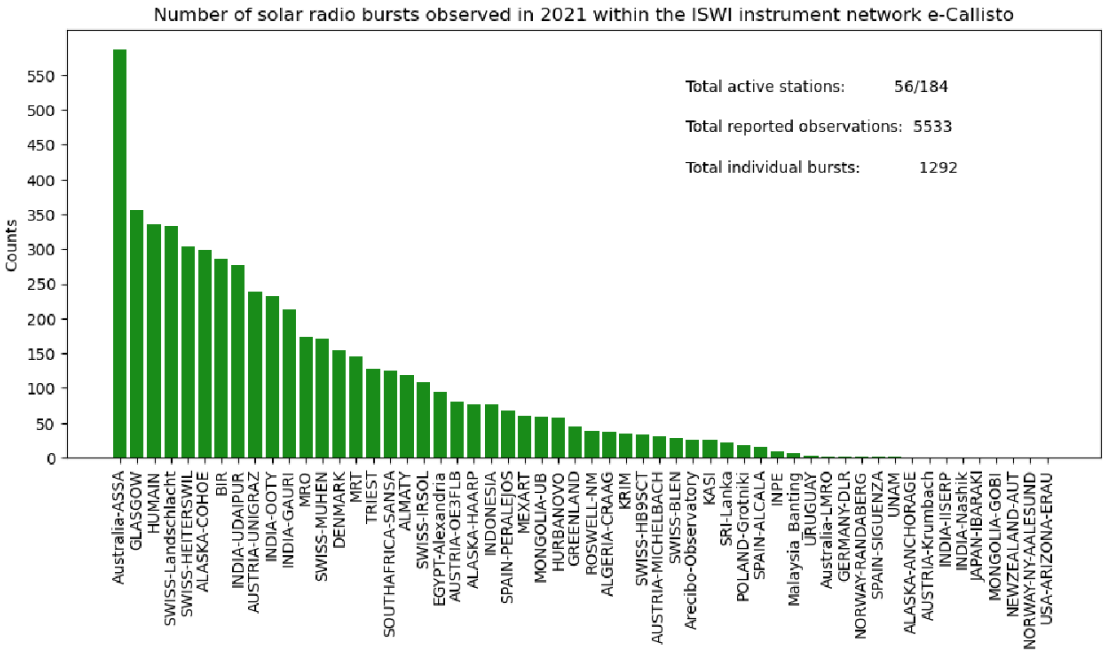


Figura 4.1: Recopilación del número de anomalías detectadas en 2021 por las estaciones que proporcionan datos a la red de e-Callisto [30].

Los datos son imágenes con formato ".fit" que representan las mediciones recogidas por la estación en un periodo de 15 minutos. Un archivo FITS es un estándar de archivo portátil utilizado en la comunidad científica para almacenar imágenes y tablas.

Cada imagen será representada como una matriz de 200x3600. El eje x indica el momento de la medición en horas, minutos y segundos y el eje y la frecuencia de la radiación en un rango de 220 a 420 megahercios. Por último, cada píxel de la imagen representa la intensidad de la señal recogida.

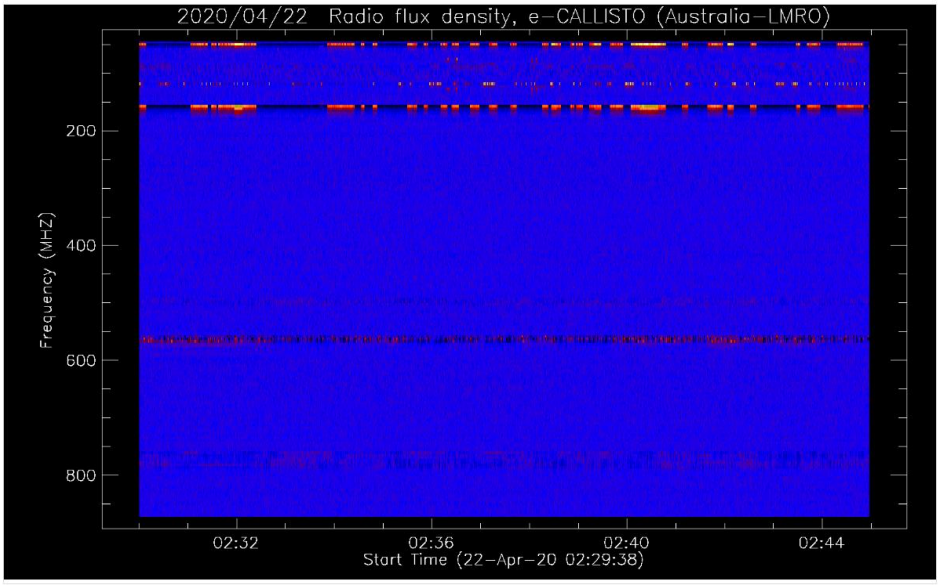


Figura 4.2: Ejemplo de una imagen recogida el 22/04/2020.

4.3 Extracción y carga de los datos

Con el objetivo de realizar la extracción de datos, se ha realizado un script en Python que usaremos para obtener las imágenes correspondientes entre dos años dados. El programa recorre cada día, mes y año y va descargando cada imagen. Para ello, utilizaremos la librería "requests" con la que abriremos la url donde se encuentra cada imagen y la descargaremos después en una carpeta con el nombre de la estación, ubicada en la misma ruta que el script.

Una vez tenemos los datos ya descargados, para leerlos guardaremos en una variable las rutas locales de todas las imágenes descargadas. A continuación, separaremos las rutas de las imágenes con eventos de las que no los tienen. Para saber la existencia de anomalías en cada una de las imágenes, buscaremos en la página de e-Callisto las anomalías detectadas en la fecha de los datos.

Después, dividiremos las rutas de las imágenes sin eventos en 3 listas de la siguiente manera: un 60% de las rutas serán de imágenes de entrenamiento, un 20% serán de las imágenes de validación y otro 20% de las imágenes de test. Las rutas de las imágenes con eventos las dividiremos en 2 listas: un 70% de las rutas serán rutas de imágenes de validación y un 30% serán de imágenes de test.

De estos arrays crearemos los conjuntos con las imágenes pertinentes. Para obtener la imagen a partir de su ruta, utilizaremos la librería "astropy" que nos devolverá cada imagen representada como una matriz.

4.4 Preprocesado de los datos

En este momento, ya tenemos las imágenes cargadas, por lo que vamos a comprobar las dimensiones de las matrices que representan las imágenes. Estas matrices deberían tener una dimensión de 200x3600 (frecuencia por tiempo), sin embargo, existen casos en los que pueden tener una dimensión superior o inferior. Para ello, se comprueba que en el caso de que la dimensión sea mayor, se eliminen los valores sobrantes y en el caso de que la dimensión sea menor, se añaden elementos con el valor de la media del resto de elementos.

El siguiente paso que realizamos es dividir las imágenes en particiones más pequeñas para que la carga computacional de entrenar la red neuronal sea inferior. Los conjuntos de datos estarán formados por estas particiones que tendrán toda la misma dimensión. Cada imagen la dividiremos en 10 partes, reduciendo así el intervalo de tiempo que representa cada partición. Por lo tanto, los datasets estarán formados por matrices de 200x360.

4.5 Creación de los conjuntos de datos

En este punto vamos a comentar cómo hemos creado los conjuntos de datos y para qué usaremos cada uno de ellos. Para dividir los datos que pertenecen a cada conjunto de datos, designaremos el número de elementos en base a un porcentaje con respecto al total de los datos. En nuestro caso, hemos creado tres conjuntos:

1. Datos de entrenamiento: son los datos que usaremos para entrenar a la red neuronal. Con este conjunto de datos, la red será capaz de ajustar sus parámetros internos para conseguir el output esperado.
   De los datos que componen este conjunto, conocemos tanto su valor de entrada, como el dato de salida a predecir. El modelo calcula, en base a distintas métricas, la diferencia de la salida obtenida con el dato esperado y se modifican los pesos de la red para conseguir obtener un mejor ajuste y, por ende, un mejor modelo.

2. Datos de validación: son aquellos datos que reservaremos para verificar si el modelo desarrollado funciona correctamente. Con este dataset comprobaremos si las salidas generadas por el modelo para casos completamente nuevos son válidas o no. Hay que tener en cuenta que este conjunto de datos cuente con un volumen suficiente para poder generar resultados estadísticos relevantes y ser, al mismo tiempo, un conjunto representativo de todos los datos globales.

   Es importante también que este conjunto de datos sea distinto del conjunto de datos de entrenamiento porque, si comparamos las predicciones realizadas por la red con los datos de entrenamiento introducidos como entrada con la salida esperada de los datos de entrenamiento, es posible que la red neuronal se acabe sobreentrenando.
   Hay que destacar que, para este conjunto de datos, al igual que para los datos de entrenamiento, conocemos la entrada y la salida reales de los datos.

3. Datos de test: utilizaremos este dataset para aplicar a nuestro modelo una vez haya sido entrenado para confirmar su comportamiento. Los datos que forman este conjunto son datos desconocidos para la red neuronal con los que podemos comprobar que precisión real tiene. Este tercer conjunto se crea debido a que es posible que las métricas del conjunto de datos de validación hayan influido en el proceso de entrenamiento de la red neuronal.

En nuestro proyecto en particular, utilizaremos 1033 imágenes, de las cuales 70 contienen anomalías. Con dichos datos formaremos los siguientes datasets: dos compuestos por imágenes sin anomalías, uno de imágenes con anomalías y otro mixto.

- Conjuntos de imágenes sin anomalías
  - Conjunto de imágenes de entrenamiento: estará compuesto por el 60% de las imágenes sin eventos.
  - Conjunto de imágenes de validación: estará formado por un 20% de las imágenes sin eventos.

    Los conjuntos de entrenamiento y validación de imágenes sin anomalías los usaremos para entrenar el modelo.

- Conjunto de imágenes con anomalías
  - Conjunto de imágenes de validación: lo compondrá el 70% de las imágenes con eventos. Este conjunto lo usaremos para predecir la salida del modelo.
- Conjunto compuesto por imágenes mixtas
  - Conjunto de imágenes de test: estará compuesto por un 20% de imágenes sin eventos y un 30% de las imágenes con eventos. Este conjunto servirá para comprobar el funcionamiento de la red neuronal.

Capítulo 5

Desarrollo de la red neuronal

En este capítulo explicaremos la arquitectura en particular con la que estará formada nuestra red neuronal y desarrollaremos una comparativa entre las distintas arquitecturas creadas para encontrar la que tiene un mejor rendimiento.

5.1 Arquitectura del autoencoder

Con el objetivo de encontrar la mejor configuración del autoencoder, probaremos diferentes tipos de arquitectura y variaremos los distintos parámetros de las capas internas. Nuestro autoencoder estará formado por varias capas, ya que son los componentes básicos de las redes neuronales en Keras. A continuación, explicaremos por qué tipos de capas están compuesto y las posibles configuraciones de sus parámetros.

5.1.1 Arquitectura del encoder

El encoder estará formado por capas convolucionales y capas de pooling. Las capas convolucionales en Keras que usaremos nosotros, son objetos de la clase "Conv2D" que crea una serie de filtros que aplica a la entrada de la capa para devolver un tensor con la salida de la convolución.
Por cada capa convolucional, tendremos una capa de pooling que serán objetos de la clase "MaxPooling2D" o "AveragePooling2D", en función del tipo de pooling que queramos aplicar.

A continuación, vamos a explicar los parámetros de las capas que componen el encoder. En las capas de convolución nos encontramos los siguientes parámetros:

- Número de filtros: será un entero que especifica el número de filtros de salida de la convolución.
- Tamaño de los filtros: un entero o una lista de dos enteros que servirán para especificar la altura y ancho de la ventana de convolución. En caso de que sea un entero, tendrá ese valor para las dos dimensiones.
- Padding: tendrá el valor de "valid" o "same". Si es "valid", no existirá padding. En cambio, si su valor es "same", se rellenará con ceros uniformemente a la izquierda/derecha o arriba/abajo de la entrada.
- Activation: será la función de activación a utilizar. En caso de no especificar nada, no se aplicará ninguna función. Las funciones disponibles en Keras son: "relu", "sigmoid", "softmax", "softplus", "softsign", "tanh", "selu", "elu" y "exponential".
- Strides: es un entero o una lista de dos enteros que especificará los pasos de la convolución a lo largo de la altura y el ancho. Si el parámetro introducido es un entero, el valor será igual para las dos dimensiones.

En las capas de pooling tenemos el parámetro de padding, ya definido anteriormente, y el tamaño del pooling:

- Tamaño del pooling: tamaño de la ventana de la que se elegirá un representante de los elementos que la componen, ya sea el máximo o la media de los valores.

5.1.2 Arquitectura del decoder

El decoder estará formado por capas convolucionales y capas de muestreo ascendente. Las capas convolucionales que utilizaremos para el decoder, pueden ser objetos de la clase

"Conv2D" (como las del encoder) o de la clase "Conv2DTranspose", que son capas convolucionales transpuesta. Este tipo de capa, como ya hemos explicado anteriormente, sirven para realizar una convolución en la dirección opuesta a la dirección normal. También, se utilizarán capas de muestreo ascendente de tipo "UpSampling2D" para duplicar las dimensiones de la entrada, repitiendo las filas y columnas de los datos introducidos como entrada.

A continuación, vamos a comentar los parámetros de las capas que componen el decoder. En las capas de convolución nos encontramos los mismos parámetros que ya hemos comentado arriba, en las capas de convolución del encoder. En cambio, en las capas de muestreo ascendente, solo hay un parámetro:

- Tamaño del muestreo: un entero o una tupla de dos enteros que especifiquen el número de repeticiones de las filas y las columnas de la entrada. En caso de ser solo un entero, se aplicará ese número tanto para las filas como para las columnas.

5.2 Parámetros de entrenamiento del autoencoder

Antes de empezar a desarrollar y evaluar las diferentes configuraciones, vamos a definir los parámetros de entrenamiento utilizados:

- Datos de entrenamiento: un Numpy array de las imágenes de entrenamiento sin eventos solares.
- Datos objetivos: en este parámetro debemos pasar un dataset con los datos a predecir. En nuestro caso, como queremos que el autoencoder devuelva una representación lo más parecida a los datos introducidos como entrada, los datos a predecir de nuestro modelo serán los datos de entrada. Por lo tanto, pasaremos el mismo Numpy array de las imágenes de entrenamiento que en el punto anterior.
- Número de iteraciones (epoch): entero que indicará el número de iteraciones que van a pasar los datos por la red neuronal durante el proceso de aprendizaje. Hay que tener en cuenta que un valor pequeño en este campo puede ocasionar que no se logre realizar un correcto aprendizaje y por lo tanto, se obtenga un mal rendimiento. En cambio, si el valor del epoch es demasiado elevado puede ocasionar que la red neuronal se sobreentrene. En nuestro caso, el modelo realizará 20 iteraciones.
- Tamaño del lote (batch size): es un entero que define el número de ejemplos que serán propagados por la red en cada iteración del aprendizaje. El valor predeterminado si no se especifica ninguno es 32. Para nuestro modelo utilizaremos un batch size de 15.
- Datos de validación: un Numpy array de las imágenes de validación con las que el modelo evaluará el rendimiento de la red neuronal y ajustará sus pesos internos.

Por último, vamos a comentar el algoritmo que utilizará el autoencoder, que será el parámetro "Optimizer" y el tipo de error utilizado que será el parámetro "loss":

- Optimizer: será una cadena con el nombre de un algoritmo de los que vienen ya definidos o una instancia creada de la clase "Optimizer". Para nuestro modelo utilizaremos el algoritmo "Adam". Adam, que es un algoritmo creado en 2014, es la versión extendida del descenso del gradiente estocástico que se utiliza en varias aplicaciones de aprendizaje profundo como en el procesamiento del lenguaje natural o en el campo de visión artificial. Este algoritmo es uno de los más usados debido a su veloz convergencia a la hora de buscar el mínimo de la función de coste y, en consecuencia, su rápido entrenamiento.
- Loss: puede ser una cadena con el nombre de la función de pérdida de las ya definidas en Tensorflow o una instancia creada de la clase "Loss". Utilizaremos el error cuadrático medio (MAE) que ya hemos explicado anteriormente**.**

5.3 Parámetros de entrenamiento del autoencoder

En primer lugar, para desarrollar nuestro modelo, que como ya hemos comentado será un autoencoder, vamos a partir de la base de la arquitectura que aparece en la documentación de Keras [31].

Esta arquitectura cuenta con dos capas convolucionales en el encoder y otras dos en el decoder. El encoder tendrá una capa de pooling después de cada capa convolucional para reducir la cantidad de parámetros al quedarse con las características principales. Por la parte del decoder, además de las capas convolucionales normales, se utilizarán capas de convolución transpuesta (Conv2DTranspose) que realiza una operación de convolución inversa.

Una vez entrenado el modelo, calculamos el umbral de detección. Para ello, introduciremos como entrada el conjunto formado por los datos de validación de las imágenes con eventos y utilizaremos su predicción para calcular el error cuadrático medio de cada imagen.

Toda predicción cuya diferencia con la imagen de entrada sea mayor que dicho umbral, será considerada como una anomalía.

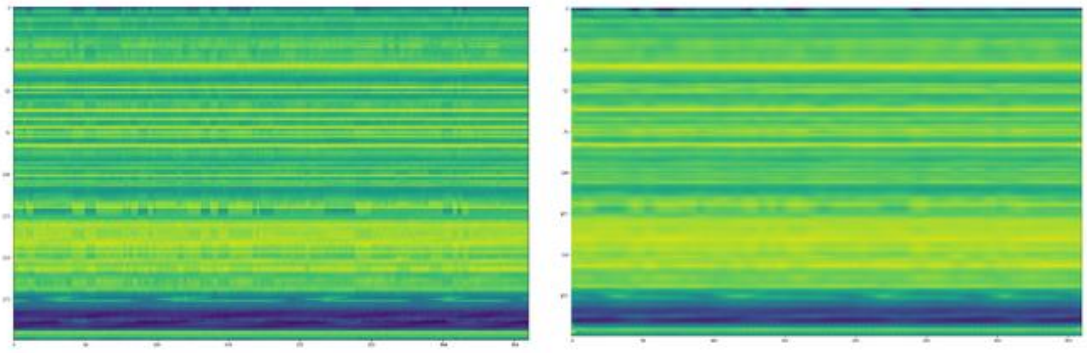


Figura 5.1: A la izquierda la imagen original y a la derecha la imagen obtenida como salida del autoencoder

Por último, calcularemos las métricas de clasificación para poder comparar el rendimiento de las distintas arquitecturas.

5.4 Elección del umbral de detección

En este punto vamos a realizar una comparativa entre distintos valores que puede tener el umbral de detección para elegir el que mejor logre clasificar los datos de entrada. Este umbral se va a utilizar el conjunto de validación de los datos con anomalías y se va a calcular en función de los errores entre las imágenes originales y las imágenes obtenidas como salida del autoencoder.

Para ello vamos a estudiar el rendimiento de distintos umbrales para 3 configuraciones distintas de autoencoders:

CONFIGURACIÓN 1					
ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

CONFIGURACIÓN 2					
ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	64	(3,3)	1	Softplus	Same
MaxPooling2D		(2,2)			Same
Conv2D	64	(3,3)	1	Softplus	Same
MaxPooling2D		(2,2)			Same
DECODER					
Conv2DTranspose	64	(3,3)	2	Softplus	Same
Conv2DTranspose	64	(3,3)	2	Softplus	Same
Conv2D	1	(3,3)		Softplus	Same

CONFIGURACIÓN 3					
ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Sigmoid	Same
AveragePooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Sigmoid	Same
AveragePooling2D		(2,2)			Same
DECODER					
Conv2DTranspose	32	(3,3)	2	Sigmoid	Same
Conv2DTranspose	32	(3,3)	2	Sigmoid	Same
Conv2D	1	(3,3)		Relu	Same

En la tabla vamos a comparar el Accuracy y los Falsos Negativos de cada umbral para quedarnos con el que dé mejor resultado. Los umbrales serán una proporción calculada en función de la media de los errores de las imágenes.

Accuracy	Media de los errores	Media de los errores + 10%	Media de los errores - 10%	Media de los errores - 20%	Media de los errores - 25%	Media de los errores - 30%	Media de los errores - 35%
Configuración 1	87.2%	89.7%	85.2%	85.1%	85.0%	84.5%	84.0%
Configuración 2	90.8%	89.6%	90.2%	86.3%	86.1%	85.7%	85.1%
Configuración 3	85.0%	84.6%	85.0%	85.0%	85.0%	84.6%	84.1%

Falsos Negativos	Media de los errores	Media de los errores + 10%	Media de los errores - 10%	Media de los errores - 20%	Media de los errores - 25%	Media de los errores - 30%	Media de los errores - 35%
Configuración 1	78	105	74	68	60	59	59
Configuración 2	109	106	100	77	70	70	69
Configuración 3	91	100	91	90	90	88	87

En el cómputo global, obtenemos que el umbral que mejor resultado tiene es el calculado como la media de los errores menos el 25% de la media. Este será el umbral que utilizaremos para la comparativa de las arquitecturas.

5.5 Pruebas de las arquitecturas del autoencoder

En este punto vamos a realizar las pruebas de dos tipos de autoencoder: los compuestas por dos capas convolucionales y los que tienen tres. Para realizar la comparativa, nos fijaremos en la cantidad de falsos negativos que detecta el modelo y el porcentaje de accuracy. Primeramente, analizaremos individualmente que resultado da cada parámetro, para después combinar los parámetros con mejor rendimiento para obtener la mejor arquitectura posible.

5.5.1 Arquitectura de dos capas convolucionales

En primer lugar, vamos a empezar a evaluar un autoencoder compuesto por un encoder y decoder de dos capas convolucionales, seguidas cada una de una capa de pooling de tipo máximo. El decoder por su parte estará compuesto por dos capas convolucionales transpuesta y una de tipo normal. Los parámetros de la arquitectura utilizada son los siguientes:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_18 (InputLayer)        [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_70 (Conv2D)           (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_33 (MaxPooling (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_71 (Conv2D)           (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_34 (MaxPooling (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_25 (Conv2DT (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_26 (Conv2DT (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_72 (Conv2D)           (None, 200, 360, 1)       289
=================================================================
Total params: 28,353
Trainable params: 28,353
Non-trainable params: 0
```

Debemos comentar que la última capa debe tener un único hilo para que la salida del autoencoder tenga las mismas dimensiones que la entrada, como podemos ver en la columna del medio de la imagen anterior.

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
170	284	1716	40	85.339%

Obtenemos 40 falsos negativos y un más que aceptable accuracy del 85.339%.

A continuación, vamos a ir probando a modificar la arquitectura anterior, para ver con que configuraciones obtenemos mejor rendimiento. Lo primero que vamos a cambiar es el número de filtros utilizados aumentando el número a 64 en todas las capas:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	64	(3,3)	2	Relu	Same
Conv2DTranspose	64	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_19 (InputLayer)        [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_73 (Conv2D)           (None, 200, 360, 64)      640
_________________________________________________________________
max_pooling2d_35 (MaxPooling (None, 100, 180, 64)      0
_________________________________________________________________
conv2d_74 (Conv2D)           (None, 100, 180, 64)      36928
_________________________________________________________________
max_pooling2d_36 (MaxPooling (None, 50, 90, 64)        0
_________________________________________________________________
conv2d_transpose_27 (Conv2DT (None, 100, 180, 64)      36928
_________________________________________________________________
conv2d_transpose_28 (Conv2DT (None, 200, 360, 64)      36928
_________________________________________________________________
conv2d_75 (Conv2D)           (None, 200, 360, 1)       577
=================================================================
Total params: 112,001
Trainable params: 112,001
Non-trainable params: 0
```

Los resultados obtenidos para esta arquitectura son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
171	240	1760	39	87.376%

Obtenemos 39 falsos negativos y un accuracy del 87.376%.

Seguidamente, vamos a observar el resultado de disminuir a 16 el número de hilos utilizados:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	16	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	16	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	16	(3,3)	2	Relu	Same
Conv2DTranspose	16	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de este modelo tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 200, 360, 16)      160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 100, 180, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 100, 180, 16)      2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 50, 90, 16)        0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 100, 180, 16)      2320
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 200, 360, 16)      2320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 200, 360, 1)       145
=================================================================
Total params: 7,265
Trainable params: 7,265
Non-trainable params: 0
```

Los resultados obtenidos después de entrenar el modelo son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
161	270	1730	49	85.566%

Obtenemos 49 falsos negativos y un accuracy del 85.566%.

A continuación, vamos a aumentar el número de hilos en la primera capa convolucional del encoder y en la última capa convolucional transpuesta del decoder:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	64	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 200, 360, 64)      640
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 100, 180, 64)      0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 100, 180, 32)      18464
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_14 (Conv2DT (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_15 (Conv2DT (None, 200, 360, 64)      18496
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 200, 360, 1)       65
=================================================================
Total params: 46,913
Trainable params: 46,913
Non-trainable params: 0
```

Los resultados de este modelo son:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
160	270	1730	50	85.52%

Obtenemos 50 falsos negativos y un accuracy del 85.52%.

A continuación, vamos a probar a aumentar el número de hilos en la última capa convolucional del encoder y en la primera capa convolucional transpuesta del decoder:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	64	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_10 (InputLayer)        [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 100, 180, 64)      18496
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 50, 90, 64)        0
_________________________________________________________________
conv2d_transpose_18 (Conv2DT (None, 100, 180, 64)      36928
_________________________________________________________________
conv2d_transpose_19 (Conv2DT (None, 200, 360, 32)      18464
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 200, 360, 1)       33
=================================================================
Total params: 74,241
Trainable params: 74,241
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
170	259	1741	40	86.471%

Obtenemos 40 falsos negativos y un accuracy del 86.471%.

Podemos observar que, como era de esperar, cuando mayor es el número de hilos, mejor resultado obtenemos tanto en el Accuracy como en el número de falsos negativos. Sin embargo, también aumenta considerablemente el coste computacional del entrenamiento. Además, para combinar distintos números de hilos utilizado, según nuestro análisis, es mejor utilizar un número superior en las capas internas del autoencoder.

Ahora vamos a probar a modificar la función de activación utilizada, vamos a ver qué resultado tenemos utilizando la función de activación "Softplus":

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Softplus	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Softplus	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Softplus	Same
Conv2DTranspose	32	(3,3)	2	Softplus	Same
Conv2D	1	(3,3)		Softplus	Same

La arquitectura interna de esta red neuronal tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_10 (InputLayer)        [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_15 (Conv2DT (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_16 (Conv2DT (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 200, 360, 1)       33
=================================================================
Total params: 28,097
Trainable params: 28,097
Non-trainable params: 0
```

Los resultados que predice este modelo son:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
160	263	1737	50	85.837%

Obtenemos 50 falsos negativos y un accuracy del 85.837%.

Seguidamente, vamos a probar a ver qué resultado obtenemos utilizando la función de activación “Selu”:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Selu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Selu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Selu	Same
Conv2DTranspose	32	(3,3)	2	Selu	Same
Conv2D	1	(3,3)		Selu	Same

La arquitectura interna de este modelo tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 200, 360, 1)       33
=================================================================
Total params: 28,097
Trainable params: 28,097
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
136	290	1710	74	83.529%

Obtenemos un buen resultado con 74 falsos negativos y un accuracy del 83.529%.

Ahora vamos a probar ahora con la función de activación “Elu”:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Elu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Elu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Elu	Same
Conv2DTranspose	32	(3,3)	2	Elu	Same
Conv2D	1	(3,3)		Elu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 200, 360, 1)       33
=================================================================
Total params: 28,097
Trainable params: 28,097
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados conseguidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
165	284	1716	45	85.113%

Obtenemos 45 falsos negativos y un accuracy del 85.113%.

Con respecto a la función de activación utilizada, como hemos visto, las que mejor resultado nos han dado han sido la “Relu” y la “Elu”. El resto de las funciones de activación que no aparecen analizadas es porque la salida que obteníamos de la red neuronal era una imagen binaria, por lo que se ha descartado su utilización.

Vamos a probar ahora a cambiar el tipo de capa de pooling de “max pooling” a “average pooling”:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
AveragePooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
AveragePooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 200, 360, 32)      320
_________________________________________________________________
average_pooling2d (AveragePo (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 100, 180, 32)      9248
_________________________________________________________________
average_pooling2d_1 (Average (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 200, 360, 1)       33
=================================================================
Total params: 28,097
Trainable params: 28,097
Non-trainable params: 0
```

Los resultados de este modelo son:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
161	287	1713	49	84.796%

Obtenemos 49 falsos negativos y un accuracy del 84.796%, por lo que no mejora el resultado obtenido con el pooling de tipo máximo.

A continuación, vamos a probar a aumentar el tamaño del filtro de convolución del encoder:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(5,5)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(5,5)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red neuronal tiene la siguiente estructura:

```
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 200, 360, 32)      832
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 100, 180, 32)      25632
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 200, 360, 1)       33
=================================================================
Total params: 44,993
Trainable params: 44,993
Non-trainable params: 0
```

Los resultados obtenidos con este modelo son:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
160	270	1730	50	85.52%

Obtenemos 50 falsos negativos y un accuracy del 85.52%, por lo que tampoco mejoramos los resultados conseguidos anteriormente

A continuación, vamos a probar a utilizar capas de muestreo ascendente de tipo "UpSampling2D" en el decoder, en vez de las capas de convolución transpuesta:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
UpSampling2D		(2,2)			
UpSampling2D		(2,2)			
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 100, 180, 32)      0
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 200, 360, 32)      0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 200, 360, 1)       33
=================================================================
Total params: 9,601
Trainable params: 9,601
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados que ha obtenido esta arquitectura son:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
156	240	1760	54	86.697%

No conseguimos un rendimiento destacable, ya que obtenemos 54 falsos negativos y un accuracy del 86.697%.

Por último, vamos a probar a utilizar las capas de muestreo ascendente de tipo "UpSampling2D" junto a las capas de convolución transpuesta:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Relu	Same
UpSampling2D		(2,2)			
Conv2DTranspose	32	(3,3)	2	Relu	Same
UpSampling2D		(2,2)			
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_transpose_13 (Conv2DT (None, 50, 90, 32)        9248
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_transpose_14 (Conv2DT (None, 100, 180, 32)      9248
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 200, 360, 32)      0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 200, 360, 1)       33
=================================================================
Total params: 28,097
Trainable params: 28,097
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
182	260	1740	28	86.968%

Con esta arquitectura, sí que hemos obtenido un rendimiento bastante destacable consiguiendo 28 falsos negativos y un accuracy del 86.968%.

Como resumen de los mejores parámetros que hemos analizado, la arquitectura que menos falsos negativos ha tenido con 28 ha sido la última en la que combinábamos las capas de tipo "Conv2DTranspose" y "UpSampling2D" Con respecto al accuracy, la mejor ha sido en la que se han utilizado 64 hilos por cada capa.

5.5.2 Arquitectura de tres capas convolucionales

En este punto vamos a comparar arquitecturas de tres capas convolucionales que vamos a desarrollar basándonos en los resultados obtenidos en el punto anterior.
En primer lugar, vamos a evaluar un autoencoder compuesto por un encoder de tres capas convolucionales, seguidas cada una de una capa de pooling de tipo máximo. El decoder por su parte estará compuesto por tres capas convolucionales transpuesta y una de tipo normal. Los parámetros de la arquitectura utilizada son los siguientes:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2DTranspose	32	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 50, 90, 32)        9248
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 25, 45, 32)        0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 50, 90, 32)        9248
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 200, 360, 1)       289
=================================================================
Total params: 46,849
Trainable params: 46,849
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
210	340	1660	9	84.208%

Con esta configuración obtenemos solamente 9 falsos negativos y un accuracy del 84.208%.

A continuación, vamos a probar la arquitectura anterior cambiando la función de activación a la de tipo “Elu”:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	32	(3,3)	1	Elu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Elu	Same
MaxPooling2D		(2,2)			Same
Conv2D	32	(3,3)	1	Elu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2DTranspose	32	(3,3)	2	Elu	Same
Conv2DTranspose	32	(3,3)	2	Elu	Same
Conv2DTranspose	32	(3,3)	2	Elu	Same
Conv2D	1	(3,3)		Elu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 200, 360, 32)      320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 100, 180, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 100, 180, 32)      9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 50, 90, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 50, 90, 32)        9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 25, 45, 32)        0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 50, 90, 32)        9248
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 100, 180, 32)      9248
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 200, 360, 32)      9248
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 200, 360, 1)       289
=================================================================
Total params: 46,849
Trainable params: 46,849
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
160	251	1749	50	86.38%

Obtenemos 50 falsos negativos y un accuracy del 86.38%.

Viendo los resultados, podemos afirmar que la función de activación “Relu” da mejores resultados que la función “Elu”, por lo que será la que utilizaremos.

A continuación, vamos a probar una arquitectura que cuenta con capas de tipo “UpSampling2D” y de tipo “Conv2D” en el decoder:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	64	(3,3)	2	Relu	Same
UpSampling2D		(2,2)			
Conv2D	64	(3,3)	2	Relu	Same
UpSampling2D		(2,2)			
Conv2D	64	(3,3)	2	Relu	Same
UpSampling2D		(2,2)			
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 200, 360, 64)      640
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 100, 180, 64)      0
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 100, 180, 64)      36928
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 50, 90, 64)        0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 50, 90, 64)        36928
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 25, 45, 64)        0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 25, 45, 64)        36928
_________________________________________________________________
up_sampling2d_6 (UpSampling2 (None, 50, 90, 64)        0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 50, 90, 64)        36928
_________________________________________________________________
up_sampling2d_7 (UpSampling2 (None, 100, 180, 64)      0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 100, 180, 64)      36928
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 200, 360, 64)      0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 200, 360, 1)       577
=================================================================
Total params: 185,857
Trainable params: 185,857
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
170	303	1697	40	84.48%

Obtenemos 40 falsos negativos y un accuracy del 84.48%.

Ahora vamos a probar a ejecutar una arquitectura con capas convolucionales transpuestas en el decoder con un mayor número de hilos en las capas internas:

ENCODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	64	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same
Conv2D	128	(3,3)	1	Relu	Same
MaxPooling2D		(2,2)			Same

DECODER					
Tipo de Capa	Nº filtros	Tamaño filtro	Stride	Activation	Padding
Conv2D	128	(3,3)	2	Relu	Same
Conv2D	64	(3,3)	2	Relu	Same
Conv2D	64	(3,3)	2	Relu	Same
Conv2D	1	(3,3)		Relu	Same

La arquitectura interna de esta red tiene la siguiente estructura:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 200, 360, 1)]     0
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 200, 360, 64)      640
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 100, 180, 64)      0
_________________________________________________________________
conv2d_30 (Conv2D)           (None, 100, 180, 64)      36928
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 50, 90, 64)        0
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 50, 90, 128)       73856
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 25, 45, 128)       0
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 50, 90, 128)       147584
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 100, 180, 64)      73792
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 200, 360, 64)      36928
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 200, 360, 1)       577
=================================================================
Total params: 370,305
Trainable params: 370,305
Non-trainable params: 0
```

Después de entrenar el modelo, los resultados obtenidos son los siguientes:

Matriz de confusión				Accuracy
TP	FP	TN	FN	
170	300	1700	40	84.615%

Obtenemos 40 falsos negativos y un accuracy del 84.615%.

Después de realizar la comparativa, la mejor arquitectura que hemos encontrado ha sido la compuesta por tres capas convolucionales en el encoder y tres capas convolucionales transpuestas en el decoder. Con esa estructura se ha conseguido obtener solamente 9 falsos negativos. Por este motivo, a pesar de no ser el modelo con mayor accuracy, es el que mejor rendimiento ha dado porque, como ya hemos comentado, el proceso de la revisión de los espectrómetros solares se realiza manualmente y es preferible que en las imágenes que la red neuronal identifica como anomalías, haya datos que no tengan eventos (falsos positivos) a que no detecte imágenes con eventos (falsos negativos). Por eso se ha intentado minimizar el número de falsos negativos y es la métrica que más valor hemos dado a la hora de analizar el rendimiento de las distintas arquitecturas.

Capítulo 6

Conclusiones y trabajo futuro

Como ya hemos comentado, la radiación solar tiene una importancia superior a la que la mayor parte de la población normalmente le otorga y, además, tiene consecuencias en numerosas actividades humanas. Los niveles de radiación solar son influidos por los eventos solares existentes, por eso la detección de anomalías en radioespectrómetros solares nos permitirá identificar los eventos solares existentes. De esta manera, gracias a conocer la presencia de los eventos, podremos prevenir sus daños y paliar las consecuencias.

Asimismo, hay que tener en cuenta que es una tarea compleja que, hasta ahora, no se había conseguido abordar correctamente. Además, la utilización de datos reales dificulta, aún más, la resolución del problema. Sin embargo, hemos conseguido encontrar un modelo que logra identificar con bastante precisión las anomalías existentes. Para crear nuestra red neuronal, nos hemos basado en un punto de vista de aprendizaje no supervisado como son los autoencoders.

Con respecto al trabajo futuro del proyecto, se podría profundizar más en el tipo de arquitectura de los modelos y, en caso de contar con una buena capacidad de cómputo, aumentar el número de capas y de parámetros para intentar mejorar, aún más, el modelo planteado. También se podrían realizar otras pruebas de configuraciones que no hemos analizado. Por otra parte, como continuación del proyecto, se podría desarrollar una red neuronal que clasifique las imágenes en función del tipo de anomalía existente.

Bibliografía

1. C. Monstein. Catalog of dynamic electromagnetic spectra, Physics, Astron. Electron. Work Bench, pp. 1–16, 2013.
2. N. Afandi, S. Hazmin, R. Umar, and A. Ishak, “Automated solar radio burst detection on radio spectrum: a review of techniques in image processing,” Journal of Fundamental and Applied Sciences, vol. 10, no. 1S, pp. 249–267, 2018.
3. Afandi, N.Z.M., Sabri, N.H., Umar, R. et al. Burst-Finder: burst recog-nition for E-CALLISTO spectra. Indian J Phys 94, 947–957, 2020.
4. D. Singh, K. Sasikumar Raja, P. Subramanian, R. Ramesh, and C. Monstein, “Automated detection of solar radio bursts using a statistical method,” Solar Physics, vol. 294, no. 8, pp. 1–14, 2019.
5. F. R. Mario, B. G. Javier, P. M. Manuel and M. Christian, "Automatic detection of e-Callisto solar radio bursts by Deep Neural Networks," 2022 3rd URSI Atlantic and Asia Pacific Radio Science Meeting (AT-AP-RASC), pp. 1-4, 2022.
6. V. Hodge and J. Austin. A survey of outlier detection methodologies. Artificial intelligence review, 22(2):85–126, 2004.
7. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. ACM computing surveys (CSUR), 2009.
8. A. S. Weigend, M. Mangeas, and A. N. Srivastava. Nonlinear gated experts for time-series - discovering regimes and avoiding overfitting. International Journal of Neural Systems 6, 4, 373–399, 1995.
9. S. Salvador, and P. Chan, Learning states and rules for time-series anomaly detection. Tech. Rep. Department of Computer Science, Florida Institute of Technology Melbourne. 2003.
10. R. Chalapathy and S. Chawla. Deep learning for anomaly detection: A survey, 2019.
11. D. Muthukrishna, K. S. Mandel, M. Lochner, S. Webb, G. Narayan. Real-time Detection of Anomalies in Multivariate Time Series of Astronomical Data. arXiv:2112.08415, 2021.
12. K. Storey-Fisher, M. Huertas-Company, N. Ramachandra, F. Lanusse, A. Leauthaud, Y. Luo, S. Huang. Anomaly Detection in Astronomical Images with Generative Adversarial Networks, arXiv:2012.08082, 2020.
13. B. Lindemann, F. Fesenmayr, N. Jazdi, & M. Weyrich. Anomaly detection in discrete manufacturing using self-learning approaches. Procedia CIRP. 79 pp. 313-318, 2019.
14. E. Šabić, D. Keeley, B. Henderson, et al. Healthcare and anomaly detection: using machine learning to predict anomalies in heart rate data. AI & Soc 36, 149–158, 2021.
15. T. Sipes, S. Jiang, K. Moore, N. Li, H. Karimabadi, J.R. Barr. Anomaly Detection in Healthcare: Detecting Erroneous Treatment Plans in Time Series Radiotherapy Data. International Journal of Semantic Computing, 257-278. 2014
16. S. A. Rai, J.M. Monteiro, V.L. Veigas, T.N. Kumar, Melanoma Skin Cancer Detection using CNN AlexNet Architecture. 2020.
17. A. Jain, M. Arora, A. Mehra, A. Munshi. Anomaly Detection Algorithms in Financial Data. International Journal of Engineering and Advanced Technology. 10. 76-78. 2021.
18. C. Mohan, K. Mehrotra. Anomaly detection in banking operations. 2017.
19. John McCarthy. Wikipedia, La enciclopedia libre. Fecha de consulta: 12:56, septiembre 11, 2022 desde https://es.wikipedia.org/w/index.php?title=John_McCarthy_(cient%C3%ADfico)&oldid=144549093.
20. B. K. Beaulieu-Jones, J. H. Moore. Missing data imputation in the electronic health record using deeply learned autoencoders. In Pacific symposium on biocomputing, pp.207-218. 2017.

21. G. E. Hinton, & R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. science, 313(5786), 504-507, 2016.
22. M. Usman, S. Latif, J. Qadir, Using Deep Autoencoders for Facial Expression Recognition, arXiv:1801.08329, 2018.
23. W. Xu, S. Keshmiri, & G. Wang. Adversarially approximated autoencoder for image generation and manipulation. IEEE Transactions on Multimedia, 21(9), pp. 2387-2396. 2019.
24. M. Sakurada, T. Yairi, Anomaly detection using autoencoders with nonlinear dimensionality reduction, in: Proceedings of the MLSDA Second Work- shop on Machine Learning for Sensory Data Analysis, ACM, 2014, pp. 4–11, 2014.
25. S. Park, M. Kim, S. Lee, Anomaly detection for HTTP using convolutional autoencoders, IEEE Access 6 70884–70901. 2018.
26. X. Lu, Y. Tsao , S. Matsuda , C. Hori , Speech enhancement based on deep de- noising autoencoder, Interspeech pp. 436–440. 2013.
27. P. Xiong, H. Wang, M. Liu, S. Zhou, Z. Hou, X. Liu, Ecg signal enhancement based on improved denoising auto-encoder, Eng. Appl. Artif. Intell. 52 pp.194–202. 2016.
28. F. Quinga-Socasi, R. Velastegui, L. Zhinin-Vera, R. Valencia-Ramos, F. Ortega-Zamorano, & O. Chang. Digital Cryptography Implementation using Neurocomputational Model with Autoencoder Architecture. In ICAART (2) pp. 865-872. 2020.
29. Documentación de Tensorflow, Compila con el código fuente en Windows. Fecha de consulta: 13:23, septiembre 11, 2022 desde https://www.tensorflow.org/install/source_windows
30. Callisto status report. Fecha de consulta: 17:12, septiembre 11, 2022 desde https://www.e-callisto.org/StatusReports/status_92_V01.pdf
31. Documentación de Keras, Building Autoencoders in Keras. Fecha de consulta: 18:15, septiembre 11, 2022 desde https://blog.keras.io/building-autoencoders-in-keras.html

Apéndice

Presupuesto

En este punto vamos a desglosar los costes necesarios para la realización del proyecto. En primer lugar, los recursos materiales que se necesitan son:

1. Equipo informático con GPU: un ordenador que cuente con una tarjeta gráfica dedicada que agilizará todo el proceso de entrenamiento del modelo.
2. Conexión a internet: una buena conexión será necesaria para poder ejecutar nuestro proyecto.

Con respecto a los gastos, encontramos solo uno:

1. Consumo de la electricidad: consumo de la red eléctrica indispensable para que el ordenador y la red de internet funcionen correctamente.

Por último, será necesario que alguien se encargue de todo el trabajo a realizar:

1. Sueldo de un ingeniero informático.

También habrá que considerar que será necesario tener en cuenta los beneficios de los encargados de llevar a cabo el proyecto.

En la siguiente tabla detallamos la cuantía de los recursos enumerados anteriormente:

Recursos			Total (en euros)
Nombre	Unidades	Precio por cada unidad	
Equipo informático con GPU	1	1000	1000
Conexión a internet (en meses)	4	30	120
Gastos			
Consumo de la electricidad	4	40	160
Sueldo ingeniero informático (en horas)	500	30	15.000
Total antes del beneficio			16.280
Otro			
Beneficio	1	30%	4.500
TOTAL			**20.780**

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR