Trabajo Fin de Grado en
Ingeniería de las Tecnologías Industriales

PREDICCIÓN DE LA RADIACIÓN SOLAR USANDO UNA CÁMARA ALL-SKY Y REDES NEURONALES

Autor:

Jose Barrientos de la Rosa

Tutores:

Jose Ramón Domínguez Frejo

Javier García Martín

Dpto. Ingeniería de Sistemas
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Trabajo Fin de Grado
Ingeniería de las Tecnologías Industriales

PREDICCIÓN DE LA RADIACIÓN SOLAR USANDO UNA CÁMARA ALL-SKY Y REDES NEURONALES

Autor:

Jose Barrientos de la Rosa

Tutores:

Javier García Martín

Jose Ramón Domínguez Frejo

Dpto. Ingeniería de Sistemas

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2023

Trabajo Fin de Grado: PREDICCIÓN DE LA RADIACIÓN SOLAR USANDO UNA CÁMARA ALL-SKY Y REDES NEURONALES

Autor: Jose Barrientos de la Rosa
Tutores: Jose Ramón Domínguez Frejo
Javier García Martín

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mi familia, amigos, y todos los que me han acompañado en esta gran etapa.

Simplemente, gracias.

Resumen

En este trabajo de fin de grado se desarrollarán diversos modelos de redes neuronales para ver la eficacia que tienen ante el problema de estimar la radiación solar a partir de imágenes tomadas del cielo y sus datos de radiación.

Se muestra el marco teórico sobre el que se apoyan los modelos, donde se detallan las características de los diferentes parámetros ajustables y la influencia que pueden llegar a tener sobre el resultado final. Finalmente, se realiza una extensa batería de 576 pruebas para analizar los resultados de forma experimental, en la que se llegan a obtener valores de precisión superiores al 90%, al mismo tiempo que se hacen notar las principales limitaciones: la insuficiencia de datos de partida y el requerimiento de mayor potencia de *hardware*.

Para el desarrollo de este trabajo, ha sido necesario adquirir nuevas capacidades y conocimientos sobre programación, optimización de recursos *hardware* e investigación en el campo de la inteligencia artificial, los cuales fueron adquiridos de forma autodidacta.

Abstract

In this document, various neural network models will be developed to assess their effectiveness in estimating solar radiation from sky images and radiation data.

The theoretical framework supporting the models is presented, detailing the characteristics of different adjustable parameters and their potential influence on the final outcome. Subsequently, an extensive battery of 576 tests is carried out to analyze the results experimentally, achieving accuracy values exceeding 90%. At the same time, the main limitations are noted: insufficient initial data and the need for more hardware power.

The development of this project required acquiring new skills and knowledge in programming, hardware resource optimization, and research in the field of artificial intelligence, which were self-taught.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

1.1 Energía Solar

La energía solar es una de las fuentes de energía renovable más importante en la actualidad y ha experimentado un rápido crecimiento en todo el mundo debido a su bajo impacto ambiental y al paulatino descenso de sus costes.

Para el aprovechamiento de este tipo de energía se utilizan diversas tecnologías, entre las que cabe destacar los paneles solares fotovoltaicos, Figura 1.1, compuestos de células semiconductoras que transforman la luz solar directamente en electricidad mediante el efecto fotovoltaico [1], y los concentradores solares usados en centrales termosolares, Figura 1.2, que concentran la luz solar para posteriormente calentar un fluido e impulsar una turbina con el vapor generado [2].

Figura 1.1 Central Fotovoltaica

Figura 1.2 Central Termosolar

Su uso e implantación en la red eléctrica ha promovido la investigación y el desarrollo de numerosos avances tecnológicos en búsqueda de optimizar y maximizar el rendimiento en la producción de electricidad, sin embargo, existen aún obstáculos importantes en la utilización de la energía solar a gran escala.

Uno de los principales problemas es la variabilidad, ya que esta forma de producción de energía depende fuertemente de la disponibilidad de luz solar, y el paso de nubes o climas adversos como lluvias repercuten de forma directa en el rendimiento de los sistemas de producción.

Para intentar solventar este problema se ha hecho uso de diferentes tecnologías, como baterías de reserva y generadores de apoyo. Sin embargo, su efectividad depende de una predicción de la fluctuación de la nubosidad local a corto plazo, en la que la calidad y la precisión de las mediciones tomadas de radiación tiene un papel fundamental.

La medición de la radiación solar se ha llevado a cabo generalmente mediante el uso de piranómetros de termopila o fotodiodo, instrumentos que proporcionan mediciones puntuales que pueden ser insuficientes en ciertos casos, como en grandes plantas fotovoltaicas, donde la información espacial cobra importancia y la implantación de una red de piranómetros resultaría costosa. Otra forma de monitorizar la radiación solar es mediante el uso de imágenes satelitales, una alternativa que ofrece buenos resultados en horizontes temporales de entre media hora a varias horas, pero no resulta adecuada para su aplicación en áreas pequeñas o de alta variabilidad de la nubosidad, en la que se precisan horizontes temporales mucho menores.

En los últimos años se ha propuesto el uso de Cámaras de Cielo Completo (All Sky Imagers, ASI) [3], capaces de captar toda la distribución espacial en áreas de pocos kilómetros y monitorizar el cielo en tiempo real. El uso de estas cámaras se complementa con técnicas como algoritmos basados en funciones polinómicas o la visión por computador para la estimación de la radiación [4].

1.2 Aprendizaje Automático

Una de las técnicas que ha ganado popularidad en la actualidad [5], dado su crecimiento e implantación en las nuevas tecnologías de hoy día, es el uso de modelos de aprendizaje automático.

Este método consiste en seleccionar características de las imágenes tomadas con ASIs y entrenar un algoritmo de aprendizaje automático junto con variables meteorológicas y datos históricos del área de aplicación. La forma de proceder será proporcionarle al modelo un conjunto de imágenes tomadas del cielo y etiquetadas con la radiación presente en cada una de ellas.

El modelo trata de reconocer patrones y hallar correlaciones que pueda haber entre las características de las imágenes y la radiación solar, con el objetivo de estimar el efecto de la oclusión solar e intentar modelar el movimiento de las nubes en el futuro, para así lograr una predicción fiable de la radiación a corto plazo.

1.3 Objetivos

En el presente trabajo se tiene como objetivo analizar la eficacia y el rendimiento que tiene el uso de modelos de aprendizaje automático basados en redes neuronales para estimar la radiación solar. Para ello:

1. Se utilizarán dos tipos de modelos de redes neuronales artificiales, el Perceptrón Multicapa (en adelante MLP, *Multilayer Perceptron*) y las Redes Neuronales Convolucionales (en adelante CNN, *Convolutional Neural Network*), evaluando su precisión a la hora de estimar la radiación en dos casos de estudio diferentes: la radiación solar directa, cuyo uso está enfocado en el aprovechamiento térmico de la radiación en centrales termo solares, y la radiación solar difusa, que cobra interés en las centrales fotovoltaicas.

2. Se tendrá en cuenta, a parte de la precisión obtenida en la estimación, otras características que cobran importancia a la hora de implantar estas técnicas, como el coste computacional, el tamaño de la base de datos y su procesamiento previo, y las diferentes combinaciones y ajustes de los parámetros internos de los modelos, con el objetivo de maximizar la precisión y calidad de los resultados.

3. Se hará un estudio de la influencia que tienen los diferentes parámetros que configuran los modelos sobre su resultado y se describirá el proceso seguido para su ajuste.

4. Se estructura haciendo un análisis por separado de las dos redes neuronales y su funcionamiento con la radiación difusa y directa, para finalmente hacer una comparación de resultados y evaluar qué causas los influencian.

2 ESTADO DEL ARTE

2.1 Tipo de Radiación y su uso en Centrales Eléctricas

La radiación que llega a la Tierra proveniente del Sol se puede clasificar en dos categorías principales: radiación solar directa y radiación solar difusa, en su conjunto forman la radiación solar global, que es la cantidad total de energía solar que incide sobre la superficie de la Tierra.

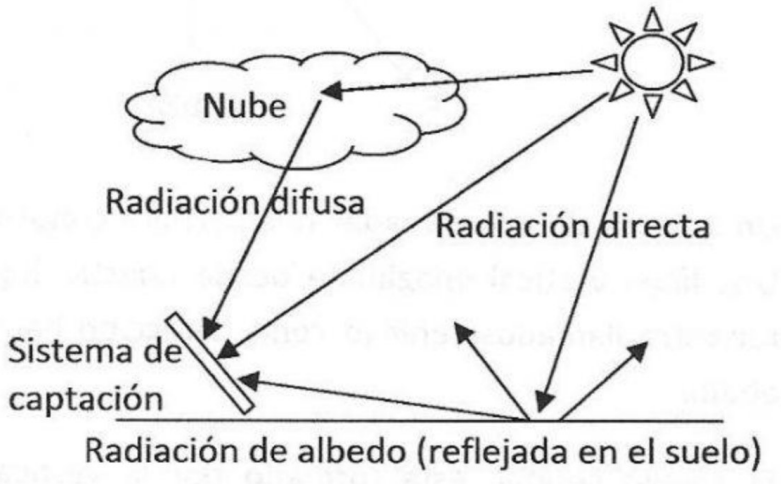


Figura 2.1 Tipos de radiación solar

2.1.1 Radiación Solar Directa

La radiación solar directa es aquella que llega a la superficie terrestre en línea recta directamente desde el Sol, sin ser dispersada significativamente por la atmósfera. Es el tipo de radiación solar más eficiente a la hora de generar electricidad, ya que ofrece una mayor cantidad de energía por unidad de área, y es aprovechada tanto en centrales solares térmicas como fotovoltaicas.

La radiación directa normal se puede medir con un pirheliómetro [6], dispositivo que consta de una cúpula de vidrio con un receptor en su interior que absorbe la radiación solar y proporciona una lectura de la irradiancia solar en vatios por metro cuadrado (W/m^2).

Existen diversos modelos cuyos diseños ofrecen prestaciones que convienen según el tipo de aplicación en la que se usen. Dos de ellos son el Pirheliómetro *Eppley* (Modelo NIP) y Pirheliómetro *Kipp* & *Zonen* (Modelo CHP1).

2.1.1 Radiación Solar Difusa

A diferencia de la radiación directa, la radiación solar difusa proviene tanto de la radiación que se dispersa en la atmósfera y es esparcida en múltiples direcciones, como de la radiación reflejada por objetos atmosféricos como las nubes. Este tipo de radiación es mucho menos intensa que la directa, sin embargo tiene una distribución más uniforme en el cielo y se hace notar incluso en días nublados.

El interés en este tipo de radiación es su aprovechamiento en centrales fotovoltaicas, ya que, aunque su eficiencia sea menor, contribuye a la generación de electricidad en momentos en los que la radiación directa es reducida, como por ejemplo en días nublados o durante el atardecer.

Para medir este tipo de radiación se suele utilizar el piranómetro [6], de diseño y funcionamiento similar al pirheliómetro pero utilizado para captar la radiación global. A este dispositivo se le puede incorporar una banda de sombra opaca que se mueve frente a la cúpula de vidrio bloqueando la radiación solar directa, por lo que únicamente se registrarán medidas de la radiación solar difusa.

2.2 Redes Neuronales

Las redes neuronales han revolucionado el campo del aprendizaje automático en las últimas décadas, demostrando su capacidad para resolver problemas complejos en una amplia variedad de disciplinas, como la medicina, sociología e ingeniería.

Existen diversos enfoques dentro del aprendizaje automático, como el aprendizaje no supervisado [7], cuyos algoritmos son capaces de reconocer patrones o estructuras para aislar conjuntos de datos que guarden relación.

Otro método muy conocido y que podemos ver en [8] es el aprendizaje por refuerzo, donde encontramos un agente que aprende a tomar decisiones en base a la respuesta recibida por sus acciones, en forma de recompensa o castigo. Se han realizado diversos estudios, como [9], en los que se utilizan videojuegos para evaluar la capacidad de aprendizaje que puede llegar a desarrollar este tipo de algoritmo.

Sin embargo, el enfoque más común de aprendizaje automático es el aprendizaje supervisado [9], inspirado en el razonamiento humano, que se basa en el aprendizaje a partir de la experiencia. Este método utiliza un conjunto de datos previamente etiquetados, que constan de una entrada y la salida esperada, para entrenar un modelo el cuál ajustará sus parámetros internos en busca de minimizar la diferencia entre las salidas predichas y las reales ya conocidas.

A pesar de los diferentes algoritmos existentes, se ha demostrado que las redes neuronales ofrecen un rendimiento considerablemente superior sobre los métodos tradicionales [11], especialmente cuando se trata de grandes conjuntos de datos o el modelo resulta ser demasiado complejo.

En trabajos como [12], se detallan los beneficios del uso de redes neuronales en el contexto de una planta solar térmica, donde se reflejan las mejoras en cuanto a resultados y tiempo de cómputo, haciéndolas ideales para aplicaciones de tiempo real.

Existen varios tipos de redes neuronales, cada una con una arquitectura y campo de aplicación específico, como las Redes Neuronales Recurrentes (RNN), que se utilizan ante problemas como el procesamiento del lenguaje natural, donde el contexto temporal y el orden cobran importancia; sin embargo, en este trabajo nos centraremos en dos tipos: MLP y CNN.

2.2.1 Redes Neuronales Artificiales (Artificial Neural Networks, ANN)

Las ANN son un tipo de modelo computacional que tratan de imitar la estructura del cerebro humano. Las unidades fundamentales de procesamiento son las neuronas, que están interconectadas mediante conexiones sujetas a ponderación, y que producen una secuencia de activaciones que se propagan a través de la red.

La neurona artificial, representada en la Figura 2.2, parte de una o varias señales de entrada (x), cuyos valores van asociados a una ponderación conocida como Peso (w). En el núcleo de la neurona se procesan dichas señales, aplicándose una combinación lineal de los pesos de las conexiones y las entradas, más un umbral denominado *Bias* (w_0):

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + w_0 \tag{2.1}$$

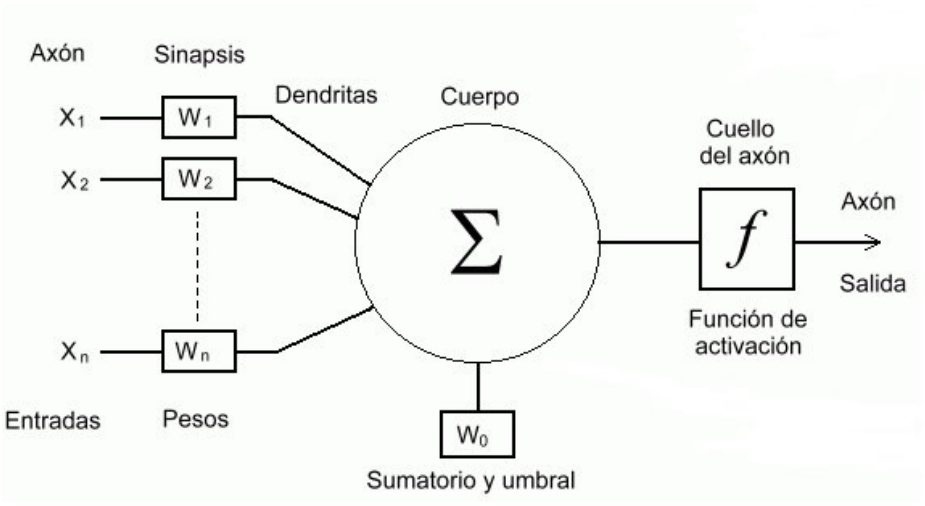


Figura 2.2 Esquema de una Neurona Artificial

Posteriormente se le aplica una función de activación no lineal que será la manera de transmitir esa información por las conexiones de salida. La función de activación usada en este proyecto se conoce como ReLu (Rectified Lineal Unit), cuya gráfica se ve en la Figura 2.3, la cual ofrece buen rendimiento ante problemas de regresión al descartar valores de ponderación negativos y no restringir su resultado a un rango de valores. A parte de esta existen otras funciones de activación, como la función escalón o sigmoidea que serán de gran interés en otras aplicaciones, como problemas de clasificación.

El resultado obtenido será la señal de salida de la neurona, que se utilizará para comparar con los resultados reales en caso de pertenecer a la capa de salida, o se propagará a través de la red si se encuentra en una capa oculta.

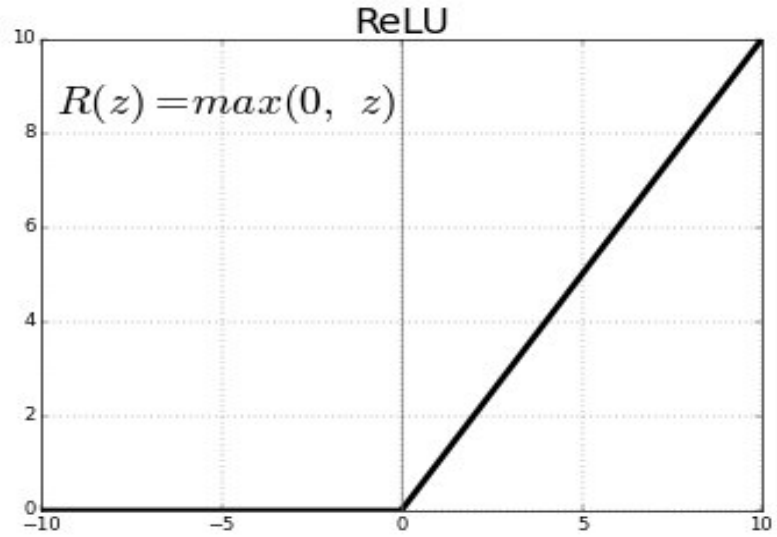


Figura 2.3 Función de Activación Rectificador Lineal Unitario

El proceso iterativo de entrenamiento consiste en minimizar una función de costes (ver Sección 2.3.1) que mide el error o diferencia entre el valor de salida de la red y el valor de salida conocido sujeto a cada entrada. Cada iteración realizada en el entrenamiento se denominará época, o *epoch.*

Seguidamente se ajustan los pesos internos de la red para reducir este error, para ello, el algoritmo calcula un gradiente para cada peso que indica qué tan significativo sería el aumento o disminución del error al aumentar el valor del peso en una cantidad pequeña, y así, ajustar el peso en dirección opuesta a dicho gradiente [13].

Este proceso de ajuste se conoce como 'retropropagación' (*Backpropagation*), y se sirve de un optimizador (ver Sección 2.3.2) para calcular el incremento al que se verá sometido el peso en cuestión. Existen optimizadores básicos, como el SGD (*Stochastic Gradient Descent*), que utilizan una tasa de aprendizaje (*Learning Rate*, α) previamente fijada para actualizar el valor del peso; sin embargo, en trabajos como [13] se exponen las ventajas que ofrecen otro tipo de optimizadores llamados adaptativos, que adaptan una tasa de aprendizaje para cada peso individualmente y según su estado en el proceso iterativo.

Una vez designados los parámetros anteriores se da paso a configurar la arquitectura de la red neuronal:

La estructura más simple con la que se puede modelar una ANN es el 'Perceptrón Simple Bicapa', desarrollada por *Frank Rosenblatt* en 1958 [15]. Esta arquitectura está formada por una capa de entrada con n neuronas y una capa de salida con *m* neuronas, cuyo funcionamiento se ajusta a lo anteriormente explicado. No obstante, resulta ineficaz ante problemas más complejos de naturaleza no lineal, como el del caso en estudio.

El uso actual de ANN se basa en modelos multicapa, que constan de una capa de entrada, un número determinado de capas ocultas y la capa de salida, como se puede observar en la Figura 2.4.

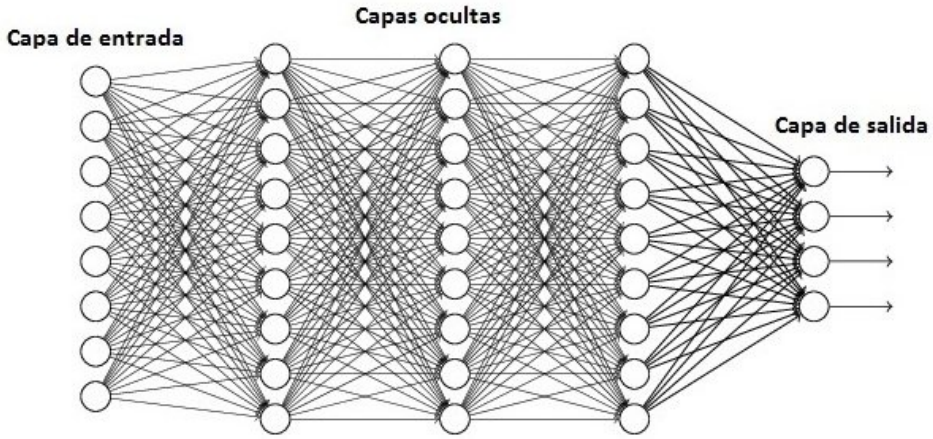


Figura 2.4 Modelo Multicapa de Red Neuronal

La arquitectura de estas redes no sigue una configuración predeterminada, se varían tanto el número de capas como de neuronas según el objetivo del modelo y los resultados ofrecidos, tendiendo presentes algunas consideraciones:

1. El número de neuronas de la capa de entrada será la dimensión de los datos de entrada.

2. La dimensión de la capa de salida dependerá de si se trata de un problema de clasificación, en el que el número de neuronas será el número de clases diferentes, o si tratamos con un problema de regresión, corresponderá al número de variables a predecir.

El diseño de la arquitectura es un paso importante que repercutirá directamente sobre el resultado obtenido, ya que el uso de un pequeño número de capas/neuronas puede provocar un ajuste insuficiente (*Underfitting*) al no ser capaz de detectar adecuadamente las diferentes características que definen a un conjunto de datos extenso. Por lo contrario, el uso de un número excesivo de neuronas da lugar a uno de los principales problemas en el diseño de arquitecturas de ANN, el sobreajuste (*Overfitting*) (ver Sección 2.3.4), que ocurre cuando la capacidad de procesamiento de la red supera a la información contenida en los datos de entrada y el modelo se ajusta de forma perfecta a dicha información, provocando errores en predicciones futuras, como se puede ver en la Figura 2.5.

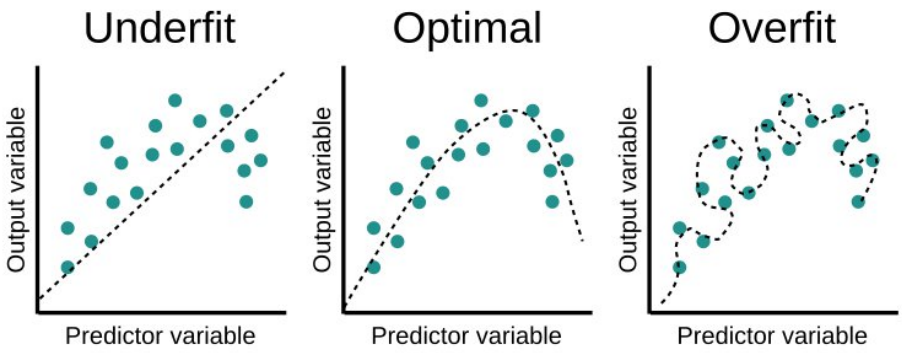


Figura 2.5 Underfit y Overfit

Con la arquitectura y los parámetros de la ANN definidos y ajustados, se utilizará un conjunto de datos de entrenamiento para realizar el proceso iterativo de aprendizaje, donde se ajustarán los pesos del modelo para la obtención de un resultado que difiera lo mínimo posible de las salidas conocidas. Por lo general, las predicciones sobre este conjunto de datos serán excelentes, por ello, para evaluar el rendimiento real del modelo se utilizará un conjunto de datos de prueba que no haya sido utilizado para el entrenamiento.

Una vez finalizado este proceso y, habiendo conseguido una precisión aceptable para el problema en cuestión, se tendrá un modelo capaz de hacer predicciones sobre datos en conjunto o individuales que se le proporcionen en un futuro a la red, sin necesidad de volver a ajustar nuevamente los parámetros anteriormente descritos.

2.2.2 Redes Neuronales Convolucionales (Convolutional Neural Networks, CNN)

Uno de los campos de interés para aplicar este tipo de modelos es la visión por computador y el tratamiento de imágenes, cuyo problema radica en la forma de introducir la información contenida en las imágenes a la red neuronal de forma efectiva.

Una primera forma de hacerlo es transformando la imagen en un único vector que contenga todos sus píxeles y adaptarlo a la capa de entrada de la red, sin embargo, a parte del coste computacional requerido, esta técnica no resulta muy eficiente, como veremos en Sección 3.1.

Es por ello que se ha desarrollado un tipo de arquitectura de red neuronal llamada Convolucional, diseñadas para el procesamiento de datos de entrada en forma de cuadrícula, como son las imágenes [16].

La característica principal de una CNN es su capa de convolución, esta operación matemática, representada en la Figura 2.6, consiste en recorrer la matriz de píxeles que forma la imagen (de izquierda a derecha y de arriba abajo) y realizar un producto escalar con una matriz más pequeña denominada filtro o kernel, obteniendo una matriz de menor dimensión y en la que se representan diferentes características según el filtro, como bordes, texturas o formas.

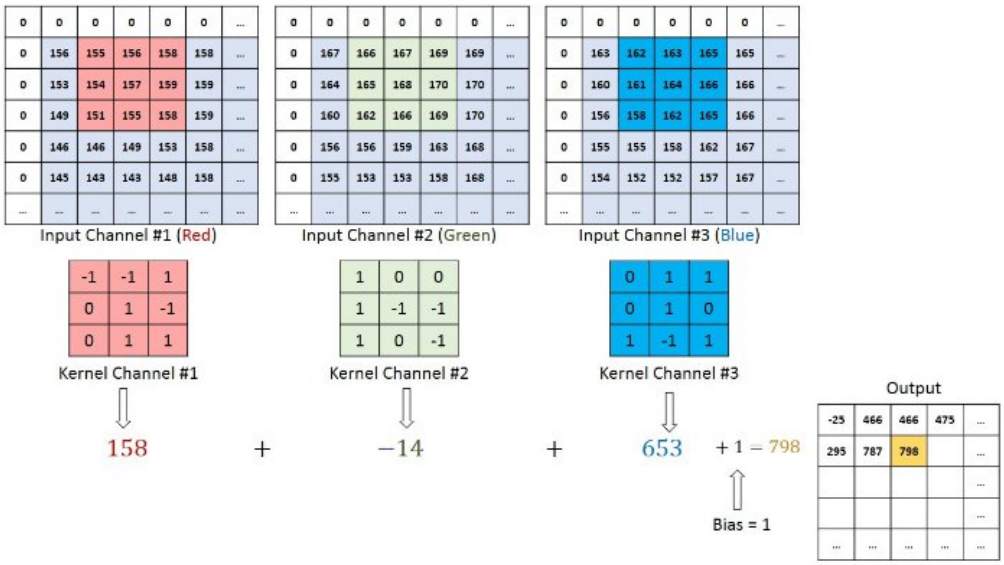


Figura 2.6 Operación de Convolución

En cada capa se realizan numerosas operaciones de convolución aplicando diferentes filtros, por lo que se obtiene un 'mapa de características' que contiene tantas matrices de salida como filtros aplicados, donde cada una de ellas captura una característica diferente de la matriz inicial.

A la salida de cada convolución se le aplica una función de activación, como ReLu, para añadir la no linealidad y obviar valores negativos. Las matrices resultantes se pasan por una capa de agrupación (*Pooling*), en la que se busca reducir la dimensionalidad recorriendo de nuevo la matriz y preservando el valor que más interese. En nuestro caso se aplicará '*Max-Pooling*', Figura 2.7, que mantiene el valor máximo reduciendo así el ruido en la imagen, frente al '*Average-Pooling*', que realiza una media entre los valores de cada submatriz.

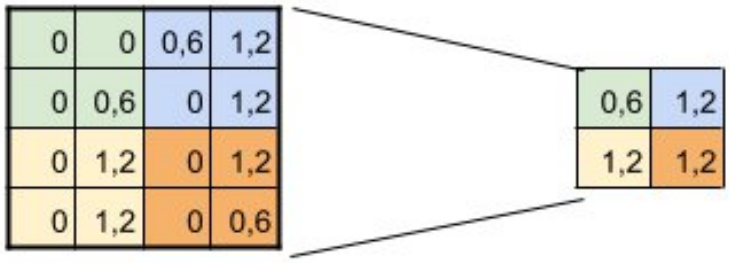


Figura 2.7 Max-Pooling

Este proceso de Convolución -*Pooling* no tiene por qué realizarse una única vez, se pueden añadir tantas combinaciones de estas capas como se requiera hasta formar arquitecturas bastante complejas pero efectivas, ya que la red será capaz de reconocer características más complejas de la imagen. Un ejemplo de ello es la arquitectura *VGGNet* [17], que llega a tener hasta dieciséis capas de convolución y cinco de *Max-Pooling*.

La última etapa consistirá en conectar el resultado de la arquitectura convolucional con un perceptrón multicapa tradicional (MLP), para ello se debe redimensionar o 'aplanar' la matriz resultante de tres dimensiones (ancho x alto x número de mapas) para obtener una capa de entrada al MLP de una única dimensión, como se puede ver en la Figura 2.8.

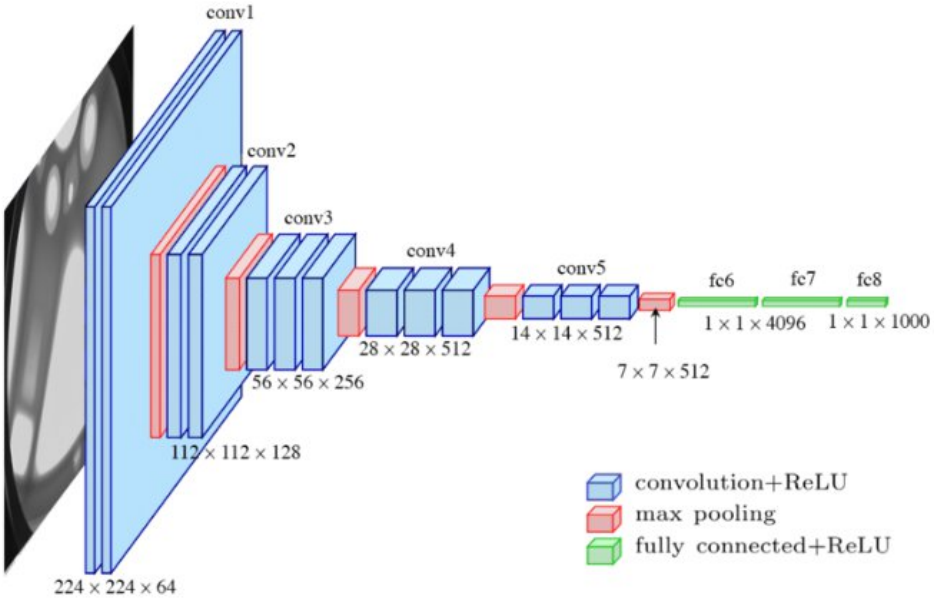


Figura 2.8 CNN completamente conectada (VGGNet)

A pesar de contener un MLP al final de la arquitectura, el proceso de entrenamiento en este tipo de redes es diferente a las tradicionales, ya que tiene como objetivo corregir el valor de los pesos de los filtros (kernels). Esto implica un menor coste computacional al tratar con un número más reducido de parámetros, en comparación a introducir la imagen en bruto directamente en un MLP.

Este tipo de modelo de aprendizaje profundo, que imita el córtex visual del ojo humano, ha demostrado un gran éxito en tareas de visión por computador, y se están realizando numerosas investigaciones en áreas como la medicina [18] para evaluar su eficacia como método para la detección temprana del cáncer [19] o para realizar diagnósticos de enfermedades como el Covid-19 [20].

2.3 Métodos y parámetros utilizados

En esta sección se dará una explicación detallada de los diferentes parámetros usados a lo largo del proyecto, así como de las diferentes formas de evaluar los resultados.

2.3.1 Función de pérdidas

La función de pérdidas, o coste, es la medida que sirve para evaluar qué tan bien se están realizando las predicciones durante el entrenamiento del modelo, cuyo objetivo es minimizar su valor. Según qué tipo de modelo se esté utilizando y su finalidad se podrán utilizar diferentes funciones de coste.

En problemas de clasificación, suelen usarse dos funciones: Entropía Cruzada Binaria (*Binary Cross-Entropy*), cuando se busca etiquetar dos clases diferentes; y Entropía Cruzada Categórica (*Categorical Cross-Entropy*), cuando nos encontramos con más de dos clases diferentes en las que catalogar los resultados del algoritmo.

Ante problemas de regresión, cuyo objetivo es predecir un valor numérico, las dos funciones de coste más utilizadas son:

1. Error Absoluto Medio (MAE)

   Mide el error medio de las diferencias absolutas entre las predicciones y los valores reales. Al tratar con valores absolutos, sólo tiene en cuenta la magnitud del error sin tener en cuenta el sesgo o dirección que tenga la medida. Además, requiere una cierta complejidad computacional a la hora de calcular los gradientes.

   Una ventaja que ofrece es que otorga el mismo peso a todos los errores individuales por igual.

$$MAE = \frac{1}{n}\sum_{j=1}^{n}\left|y_j - y_j\right| \tag{2.2}$$

2. Error Cuadrático Medio (MSE)

   Mide el error promedio al cuadrado, lo que provoca una mayor penalización hacia valores atípicos, es decir, tienen mayor peso las predicciones más alejadas de sus valores reales que las menos desviadas.

   El MSE, por sus propiedades matemáticas, facilita el cálculo de los gradientes, lo que lo hace una buena elección a la hora de evaluar el coste computacional del algoritmo.

$$MSE = \frac{1}{n}\sum_{j=1}^{n}\left(y_j - y_j\right)^2 \tag{2.3}$$

2.3.2 Optimizadores

Como se ha explicado anteriormente, el proceso de aprendizaje de una red neuronal consiste en variar el valor de los pesos de cada neurona con el fin de hallar un mínimo global de la función de coste y cumplir un margen temporal que no demore demasiado este proceso.

El encargado de calcular las variaciones a las que se tienen que someter los pesos es el optimizador, cuyo algoritmo principal es el Gradiente Descendente (GD) [21], que calcula la derivada parcial de la función de costes para cada peso y ajusta su valor en dirección contraria a esta y con una magnitud fijada por una tasa de aprendizaje (α).

$$w_{t+1} = w_t - \alpha \cdot \frac{\partial f}{\partial w} \tag{2.4}$$

Sin embargo, este algoritmo presenta ciertas limitaciones:

Debido a que la función de costes puede llegar a depender de millones de parámetros, existe una cantidad elevada de mínimos locales que hacen que el algoritmo no obtenga su máximo rendimiento, ya que una vez encontrado uno de estos mínimos, el algoritmo deja de avanzar en la búsqueda del mínimo global asumiendo que se encuentra ya en él, se puede ver una representación en la Figura 2.9.

Otro algoritmo también utilizado es el Gradiente Descendente Estocástico (SGD), una variante del GD cuya diferencia principal es que utiliza puntos aleatorios de la curva en cada paso del entrenamiento para calcular los gradientes, lo que puede ayudar a escapar de mínimos locales, pero incrementa el coste computacional al provocar que el gradiente tenga mayor fluctuación.

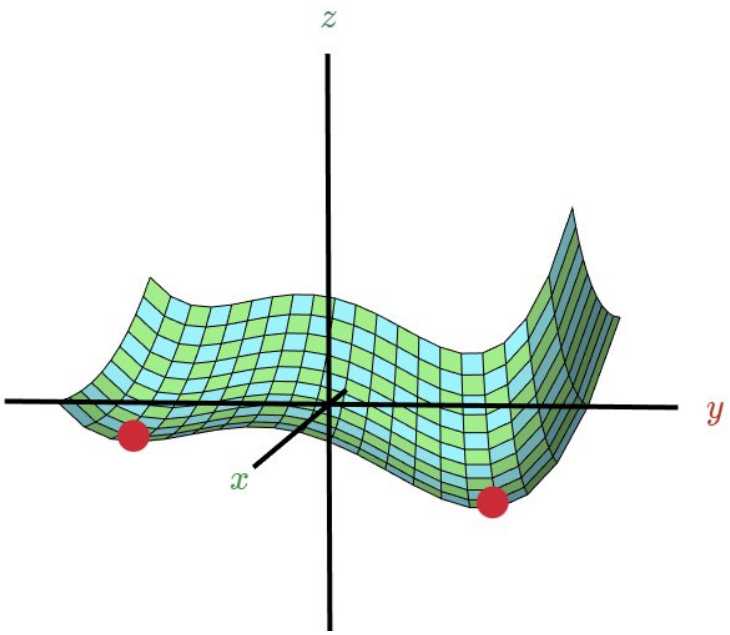


Figura 2.9 Mínimos locales de la función de costes

El segundo problema es la convergencia, ya que en función de la tasa de aprendizaje elegida, el algoritmo puede demorarse demasiado en encontrar un mínimo, o peor aún, no conseguir converger y mantenerse oscilando en torno a este. Se puede ver una representación de estos efectos en la Figura 2.10.

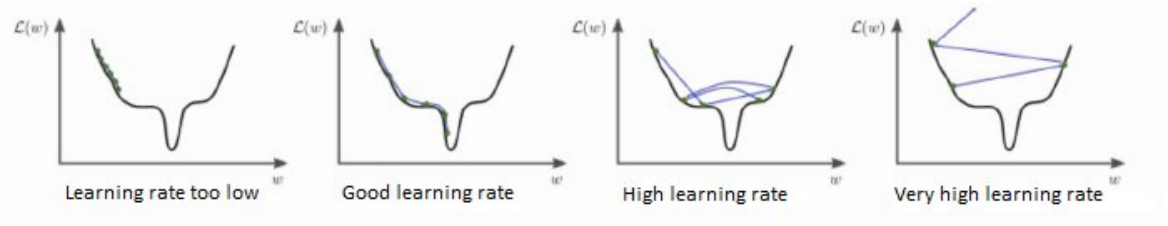


Figura 2.10 Comportamiento según la tasa de aprendizaje

Es por esto que se han desarrollado nuevos algoritmos basados en adaptar de forma automática la tasa de aprendizaje según en la situación que se encuentre el proceso de cálculo, consiguiendo así favorecer la convergencia al mismo tiempo que minimizar las posibilidades de caer en un mínimo local, o detectarlo a tiempo y esquivarlo.

Los optimizadores a continuación descritos, se basan en el uso del *Momentum*, un término que recoge una fracción acumulativa de los gradientes calculados en iteraciones o épocas anteriores, consiguiendo una mayor aceleración en el descenso del gradiente cuando la pendiente de la función de pérdidas es más pronunciada y una deceleración al presentar un perfil más suave:

1. Adam (Adaptive Moment Estimation)

   La idea detrás de *Adam* es actualizar la tasa de aprendizaje usando las medias móviles de primer y segundo orden del gradiente (m_t y u_t), lo que resulta en una rápida convergencia y una alta eficiencia evitando mínimos locales [22].

   Esto es debido a que no se guía únicamente por la mayor pendiente encontrada en el momento, que puede apuntar a un mínimo local, sino que busca una dirección de descenso que apunte al mínimo global, aunque no presente la pendiente de máximo valor.

$$m_t = \beta_1 \cdot m_{t-1} - (1-\beta_1)\cdot g_t \tag{2.5}$$

$$u_t = \beta_2 \cdot u_{t-1} - (1-\beta_2)\cdot {g_t}^2 \tag{2.6}$$

$$w_{t+1} = w_t - \left(\alpha \cdot \frac{m_t}{\sqrt{u_t + \epsilon}}\right)\cdot g_t \tag{2.7}$$

$$g_t = \frac{\partial f}{\partial w_t}; \beta_1, \beta_2, \epsilon = constantes$$

2. AdaMax

   Esta variante mantiene la idea del algoritmo *Adam*, la diferencia es que en lugar de usar las medias móviles de primer y segundo orden, utiliza un momento de orden máximo.

   Este optimizador ofrece ventajas al abordar problemas en los que los pesos tienen valores dispersos o cuando nos encontramos ante grandes conjuntos de datos, debido a su gran capacidad para adaptarse a cada peso de manera individual [23].

$$m_t = \beta_1 \cdot m_{t-1} - (1-\beta_1)\cdot g_t \tag{2.8}$$

$$u_t = max(\beta_2 \cdot u_{t-1}, |g_t|) \tag{2.9}$$

$$w_{t+1} = w_t - \left(\alpha \cdot \frac{m_t}{u_t}\right) \cdot g_t \tag{2.10}$$

En conclusión, los algoritmos de *Adam* y sus derivaciones, como *AdaMax*, son los más idóneos cuando nos encontramos con una red compleja en la que el número de parámetros y datos de entrada son elevados, aunque estos requieran un coste computacional algo mayor que los algoritmos no adaptativos, son los que ofrecen mejores resultados.

2.3.3 Métricas

A la hora de evaluar la red se utilizarán diferentes métricas que muestran la calidad de los resultados y con las que se podrá valorar si el comportamiento que ofrece es el esperado.

El conjunto de datos total se divide en dos bloques, o *sets*, uno de entrenamiento (75% - 85%) que se utilizará para entrenar la red, y otro de validación (25% - 15%), cuyos datos no se han utilizado durante el entrenamiento y servirán para calcular las métricas sobre estos, arrojando así un resultado con mayor validez sobre el desempeño del modelo.

En redes neuronales diseñadas para abordar problemas de clasificación se utilizará, entre otras, la precisión (*Accurate*) para analizar el rendimiento ofrecido. Al arrojar estos modelos resultados en formato binario, es decir, si acierta devolverá un 1 y si falla, un 0, la forma de ver el rendimiento del modelo será:

$$Acc = \frac{nº\ Aciertos}{nº\ Aciertos + nº\ Fallos} \cdot 100 \tag{2.11}$$

Esto será diferente ante problemas de regresión, ya que al trabajar con datos continuos, no es efectivo evaluar el modelo de esta forma.

Se utilizarán en este caso varias métricas anteriormente mencionadas, como el MSE y el MAE, que además de utilizarse como función de pérdidas, son buenos indicadores del rendimiento de la red.

Otra métrica de análisis también utilizada para evaluar la calidad de ajuste de un modelo es el Coeficiente de Determinación (R^2), que mide la proporción de la varianza de la variable dependiente que es predecible a partir de las variables independientes.

$$R^2 = 1 - \frac{\sum_{j=1}^{n} (y_j - \hat{y}_j)^2}{\sum_{j=1}^{n} (y_j - \bar{y}_j)^2} \tag{2.12}$$

$$\bar{y}_j: Valor\ promedio\ ; \quad \hat{y}_j: Valor\ predicho$$

El análisis del comportamiento del modelo se hará utilizando estas tres métricas descritas en conjunto, dando lugar a una imagen completa acerca del rendimiento y los posibles problemas que lo influencien.

2.3.4 Análisis de resultados

De forma complementaria a las métricas, se hace uso de diferentes gráficas que ofrecen un análisis visual tanto de los resultados como del proceso de entrenamiento, pudiendo observar en ellas comportamientos, como el *overfitting,* que resultarían difíciles de percibir de otra forma.

Una primera gráfica será la que representa la evolución de la función de pérdidas con respecto a las épocas. Estarán representadas tanto la curva de pérdidas de los datos de entrenamiento como del conjunto de validación, que se utiliza para hacer una evaluación de la red con datos que no se hayan usado para entrenarla.

En esta gráfica, se pueden observar diferentes fenómenos:

1. Underfitting

   El subajuste o *underfitting* ocurre cuando el modelo no tiene la suficiente capacidad para abordar la complejidad del conjunto de datos.

   Un modelo subajustado presentará una gráfica en la que la función de pérdidas en el conjunto de validación no decrece, lo que indica que el modelo no pudo aprender sobre el conjunto de datos, como podemos observar en la Figura 2.11.

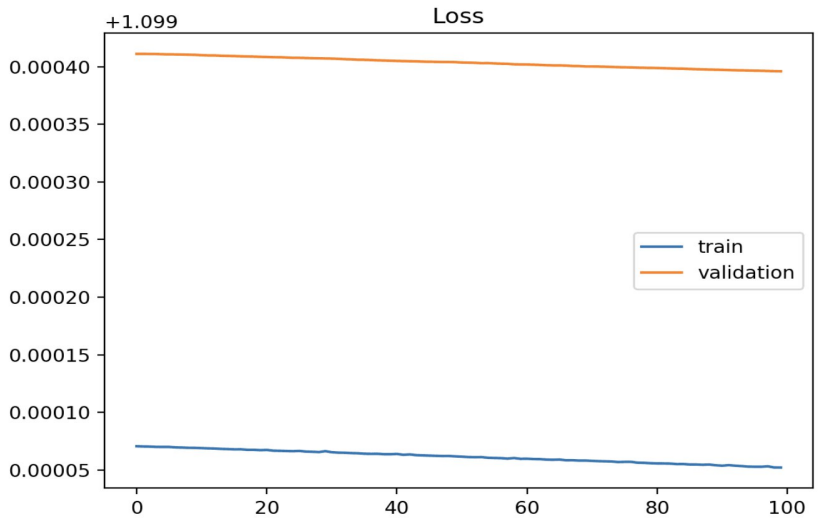


Figura 2.11 Underfitting. Modelo con poca capacidad

Otra forma de reconocer el subajuste es en una gráfica como la Figura 2.12, en la que la pérdida continúa disminuyendo sin llegar a mantenerse constante en torno a un valor, lo que significa que el proceso se detuvo de forma prematura y el modelo es capaz de seguir aprendiendo más.

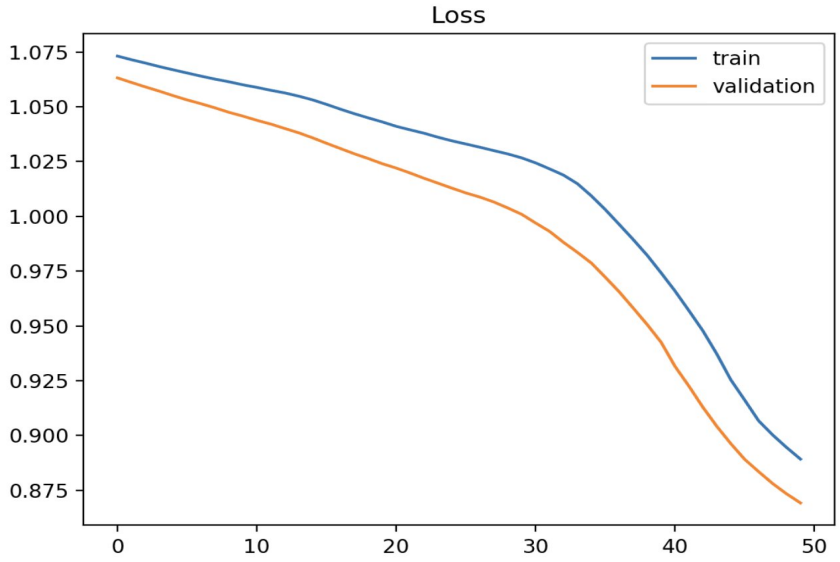


Figura 2.12 Underfitting. Modelo con poco entrenamiento

2. Overfitting

El sobreajuste u *overfitting* es uno de los problemas más comunes que surgen durante el ajuste de una red neuronal, ocurre cuando el modelo aprende de forma casi perfecta el conjunto de datos de entrenamiento, por lo que no es capaz de generalizar sobre nuevos datos, ya sea porque tiene más capacidad de la requerida por el problema o porque se ha entrenado durante demasiado tiempo.

En las gráficas es fácil de observar que se ha producido sobreajuste cuando la curva de pérdidas sobre los datos de entrenamiento continúa decreciendo y puede llegar a un valor muy cercano a 0, mientras que la curva sobre los datos de validación decrece hasta un punto de inflexión a partir del cual empieza a crecer y deja de seguir la misma dinámica de la curva de entrenamiento, como se puede observar en la Figura 2.13.

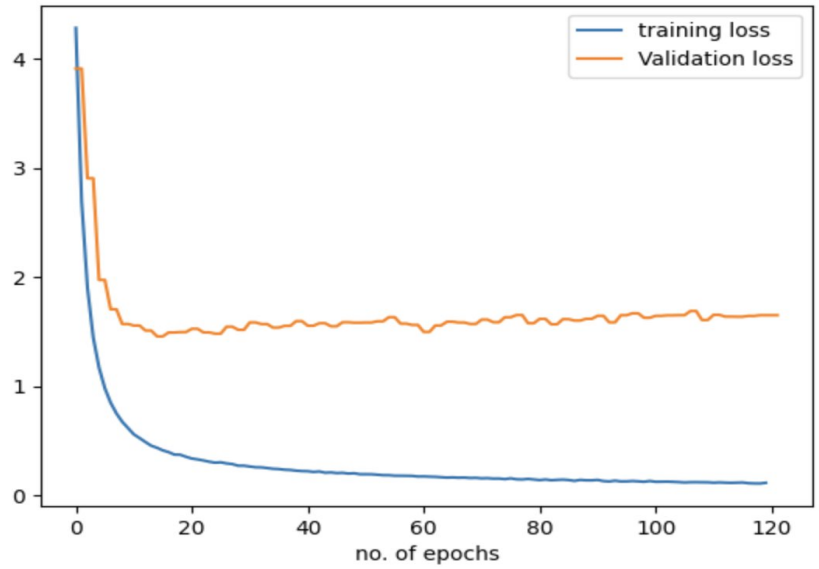


Figura 2.13 Overfitting. Pérdida de generalización

Existen diversos métodos para prevenir el sobreajuste, el más efectivo es denominado *Dropout*, que consiste en desactivar temporalmente las conexiones de un porcentaje de neuronas distribuidas aleatoriamente en una capa de la red en cada iteración. Al forzar al modelo a no depender de ninguna neurona en concreto, se consigue una mayor generalización, ya que la red continúa el entrenamiento con una fracción aleatoria de neuronas, evitando así aprender de forma excesiva alguna característica o patrones del conjunto de datos [24].

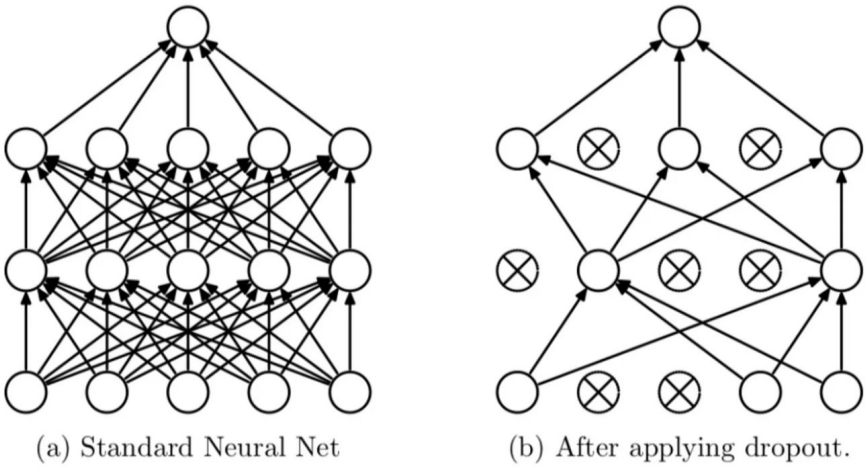


Figura 2.14 Dropout

3. Datos no representativos

En ocasiones se da la situación en la que el conjunto de datos no proporciona la suficiente información para evaluar la capacidad del modelo, esto se puede deber a que el conjunto de validación tiene pocos ejemplos en comparación con el conjunto de entrenamiento, lo cual resultaría en una representación como la Figura 2.15, donde la curva de validación presenta un comportamiento ruidoso alrededor de la curva de entrenamiento.

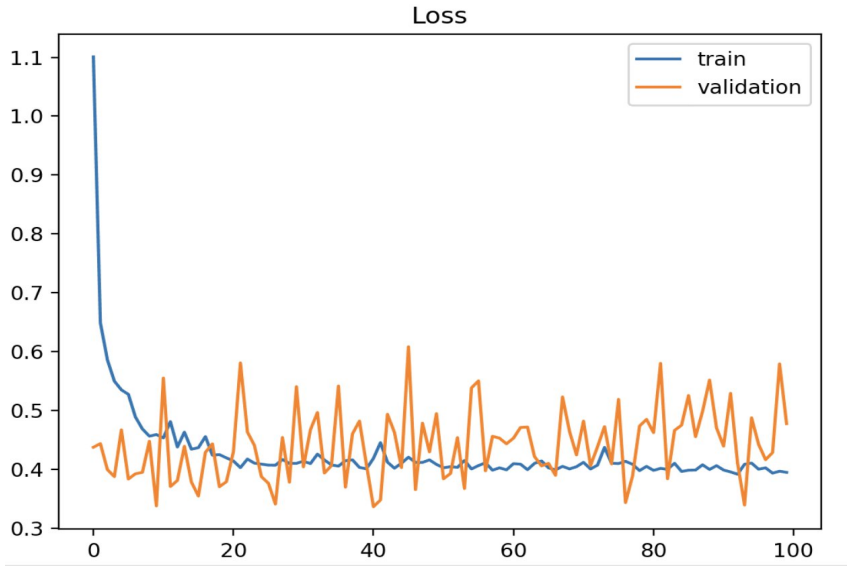


Figura 2.15 Comportamiento con datos de validación insuficientes

Otra forma de manifestarse este fenómeno es en gráficas como la Figura 2.16, en la que la curva de pérdidas de validación es menor que la de entrenamiento, lo que significa que el modelo tiene más facilidad para predecir los datos de validación que los de entrenamiento, por lo que no nos podríamos guiar por estos resultados a la hora de evaluar su rendimiento.

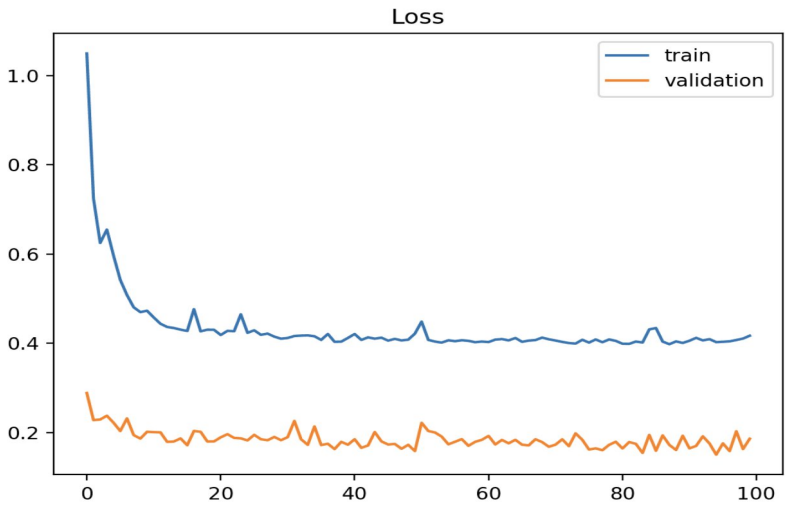


Figura 2.16 Comportamiento errático por no representatividad

El análisis gráfico de los resultados es importante para poder identificar comportamientos que provocan la pérdida de validez del modelo y actuar como se precise para intentar corregirlos. Un buen comportamiento sería uno como el mostrado en la Figura 2.17, donde las curvas de entrenamiento y validación siguen una misma dinámica y no presenta una gran distancia entre sus valores finales.

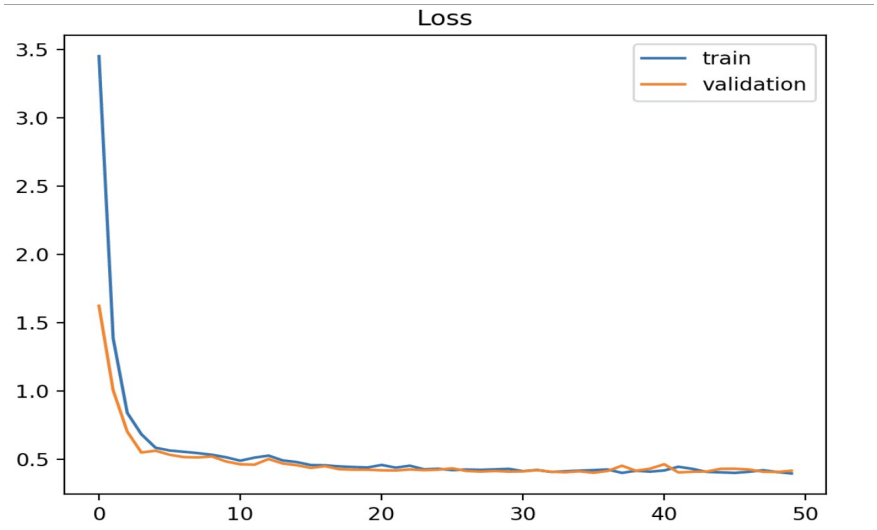


Figura 2.17 Modelo bien ajustado

Otra gráfica que resulta de utilidad para el análisis del modelo es el diagrama residual, que representa la relación entre las salidas predichas y las reales, como la Figura 2.18, donde se pueden observar ciertos detalles de interés.

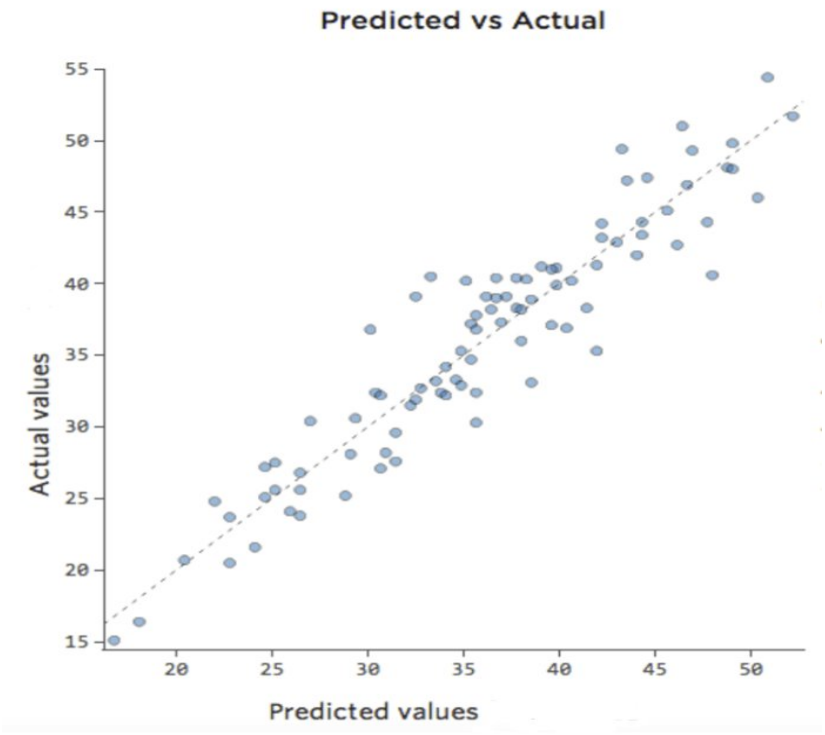


Figura 2.18 Diagrama Residual

La línea discontinua representa la zona en la que las predicciones coinciden con los valores reales, por lo que observando de qué manera se distribuyen estos residuos en torno a la línea se puede ver la eficacia que ha tenido el modelo, así como la cantidad de dispersión que presentan los datos.

2.3.5 Enriquecimiento de los datos

Uno de los problemas vistos que influencia el resultado de los entrenamientos es la falta de datos, en ocasiones no se dispone de un conjunto de datos lo suficientemente grande para que la red sea capaz de reconocer ciertas características y generalizar. Este inconveniente se agrava cuando tratamos con imágenes, ya que la falta de un conjunto de imágenes extenso limita la capacidad del modelo y puede resultar en un funcionamiento deficiente.

Para reducir, en cierta medida, el impacto negativo que provoca la escasez de imágenes, se utilizan métodos como el aumento de datos (*Data Augmentation*) para enriquecer la base de datos. La idea de este método consiste en aplicar transformaciones y manipulaciones a los datos originales, generando así nuevas muestras que conservan las características principales de los datos originales, pero presentan variaciones menores.

Algunas técnicas que se aplican a las imágenes son la rotación en un cierto ángulo, reflexión horizontal o vertical, adición de ruido Gaussiano o cambio de brillo y contraste.

Por lo general, realizar un aumento de datos siempre traerá beneficios al desempeño del modelo, pero dependiendo de la forma que se aplique o si no se realiza de forma adecuada, como por ejemplo aplicar rotaciones a imágenes en las que la orientación cobra un papel importante, puede llegar a ser contraproducente en algunos aspectos.

2.3.6 Uso de GPU *vs* uso de CPU y limitaciones

La investigación en el campo de las redes neuronales ha conllevado diversas mejoras y nuevas tecnologías que facilitan y potencian su uso en problemas de todo tipo.

Un avance muy notorio ha sido el desarrollo de *Frameworks* y librerías de lenguajes de programación especializadas en la creación, ajuste y explotación de redes neuronales. El lenguaje por excelencia para usar en este campo es *Python*, en el que se ha desarrollado un potente *framework* llamado *TensorFlow*, gracias al cuál se pueden construir modelos altamente complejos dejando atrás el cálculo matemático que se requiere para su funcionamiento.

Una ventaja que ofrece es la posibilidad de utilizar la GPU (*Graphics Processing Unit*) para el entrenamiento de la red, gracias a su compatibilidad con arquitecturas CUDA, que permiten el uso de la GPU para realizar cálculos en paralelo de propósito general.

La GPU, a diferencia de la CPU (*Central Processing Unit*), posee un número de núcleos de procesamiento mucho más elevado, lo que las hace idóneas para el cálculo matricial en paralelo, en especial con CNNs, donde el procesamiento de las imágenes requiere una gran potencia computacional para trabajar con matrices extensas.

En trabajos como [25] se realizan comparativas de diferentes redes neuronales y el rendimiento que ofrecen al ser entrenadas con CPU y GPU, la diferencia en cuanto al tiempo consumido es notable, y se hace más pronunciada conforme crecen en tamaño las matrices, como se puede observar en la Figura 2.19.

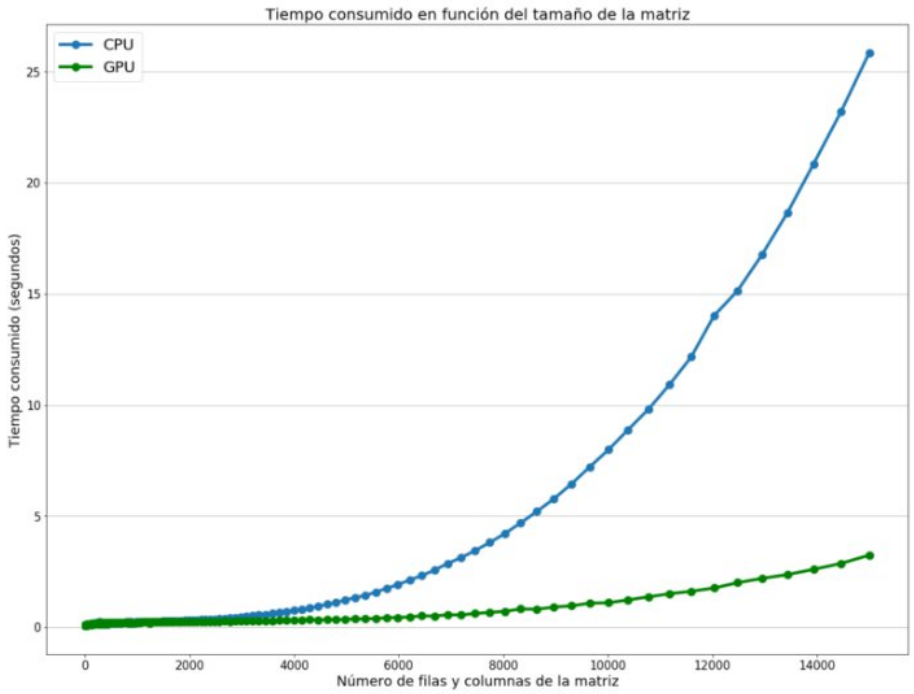


Figura 2.19 Comparativa del tiempo consumido frente al tamaño de la matriz usando CPU y GPU

3 DESARROLLO DEL PROYECTO

En este capítulo se aplicarán los métodos explicados anteriormente y se valorarán sus resultados haciendo una comparación entre las diferentes configuraciones de parámetros elegidas.

Se estructurará probando los dos tipos de red con la radiación directa, un MLP en la sección 3.1 y una RNC en la sección 3.2, seguidamente se valorará la eficacia de utilizar el *dataset* enriquecido en la sección 3.3 y finalmente se probaran los distintos modelos obtenidos con la radiación difusa en la sección 3.4. Se hará una comparación de todos los resultados obtenidos y se analizará la configuración de parámetros que ha llevado a los mejores resultados en la sección 3.5.

El lenguaje utilizado es *Python*, en el que se desarrollará la red neuronal apoyándose en el framework *TensorFlow* y con la ayuda de las librerías: *Keras*, para el diseño de la red, *Scikit-Learn* para el tratamiento de los datos, *OpenCv* para procesar las imágenes, *Numpy* y *Pandas* para el manejo de datos numéricos y su organización en tablas relacionales, *MatplotLib* para la creación de gráficas, y una base de datos de *SQLite* para almacenar los datos y facilitar su portabilidad.

El código desarrollado consta de dos *scripts*:

1. BaseDatosImagenes.py

   Servirá para crear la base de datos que relaciona cada imagen con sus medidas de la radiación. Las imágenes son leídas de un repositorio de carpetas y tratadas de manera que se eliminen imperfecciones y elementos sobrantes. Una vez tratadas serán guardadas en formato binaro (*.bob*) en una base de datos de *SQLite* que se divide en tablas donde se catalogan las imágenes en diferentes escalas, a color y en gris.

2. RedNeuronal.ipynb

   Es el *script* en el que se construye toda la estructura de la red neuronal. Está desarrollado en un cuaderno de *Jupyter*, idóneo para ejecutar el código por secciones y realizar pruebas de secciones específicas sin tener que ejecutar el *script* entero en cada prueba. En él se leerán los datos de las imágenes en formato binario y se traducirán a formato matricial, adaptando los datos al tipo de formato numérico de entrada de la red.

   Una vez entrenada la red y ajustados sus parámetros, se realizan predicciones sobre el *dataset* de pruebas para graficar los resultados y volcarlos en una nueva base de datos junto a la configuración de parámetros escogida.

La base de datos utilizada para la realización de las pruebas pertenece al Grupo de Investigación de Termodinámica y Energías Renovables de la Universidad de Sevilla. Consta de imágenes del cielo tomadas por una cámara *All-Sky* cada cinco minutos, como podemos ver en la Figura 3.1, desde las 6:30 a.m hasta las 21:00 p.m de los 23 primeros días del mes de Marzo de 2022, y de mediciones de diferentes tipos de radiación tomadas cada cinco segundos durante el mismo periodo temporal. La Estación Meteorológica de la Escuela de Ingeniería de la US está ubicada en Sevilla, a 34.71º Norte, 6.01º Oeste y 12 metros de elevación.

Las mediciones son recogidas por dos pirheliómetros (CHP1 y NIP) y un piranómetro (Difusa CMP10), una cámara *All-Sky* que graba las imágenes y diversos sensores que recogen otras variables (temperatura, humedad relativa, velocidad del viento, altura de las nubes, presión atmosférica y hora solar).Aunque estas otras mediciones no se han usado en este proyecto, se podrían utilizar como entradas adicionales para mejorar las estimaciones obtenidas con las redes neuronales.

Figura 3.1 Imagen del cielo tomada con cámara All-Sky

Esta base de datos se dividirá en tres bloques ordenados aleatoriamente: un 60% correspondiente a datos de entrenamiento, un 20% para datos de validación utilizados para realizar una valoración del rendimiento interno, y otro 20% para datos de prueba, con los que se harán las predicciones y la evaluación visual de los resultados.

3.1 Perceptrón Multicapa

La primera consideración a tener en cuenta para este tipo de red es que el formato de entrada de los datos en la red debe ser un vector plano de tantas componentes como píxeles tengan las imágenes, por lo tanto el primer paso del procesado de las imágenes será convertirlas en un vector de una única dimensión.

Por lo tanto, se creará una matriz compuesta de tantas filas como imágenes se quieran introducir a la red y tantas columnas como píxeles contengan dichas imágenes. La red leerá los datos de esta matriz por lotes de tamaño definido por un parámetro denominado *Batch Size*, el cuál se verá su influencia más adelante.

Este factor influye en el tiempo de cómputo y en la capacidad que tiene la GPU de almacenar dicha información. Es por eso que se realizarán primeramente pruebas con una configuración estándar de los parámetros, definida en la Sección 3.1.1, y se utilizarán distintas escalas para las imágenes, en concreto 16x16, 32x32, 64x64 y 128x128 píxeles.

3.1.1 Desarrollo

Para realizar las primeras pruebas se elige una configuración proveniente de un método llamado *Grid Search*, que realiza un barrido por todas las combinaciones posibles de un listado de parámetros proporcionado.

Se elige un tamaño de la red formado por dos capas de 3072 y 768 neuronas en sentido descendente y, tras realizar *Grid Search*, se obtiene la siguiente configuración como la que arroja mejores resultados:

Optimizador Adamax, Batch Size de 128 y la función de pérdidas MSE.

Tras varias pruebas realizadas variando la escala de las imágenes, se concluye lo siguiente:

1. Con un tamaño de 16 x 16 píxeles, la imagen no arroja suficiente información, por lo que el resultado de las predicciones es deficiente.

2. Usando un tamaño de 128 x 128 píxeles, la GPU no es capaz de asignar la suficiente memoria para completar el entrenamiento, dando un fallo de sobrecarga de VRAM y sin arrojar ningún resultado.

3. Los dos tamaños que muestran buen rendimiento son imágenes de 32 y 64 píxeles, siendo las que ofrecen mejores resultados las de 64 píxeles. Por ello se realizarán las siguientes pruebas usando únicamente imágenes de este tamaño.

3.1.1.1 Tamaño de la Red

La siguiente batería de pruebas se hará para ver el tamaño óptimo de la red y cómo responden diferentes estructuras.

Se prueba inicialmente la red más sencilla posible, formada únicamente por la capa de entrada y la capa de salida (Perceptrón Simple), con la que vemos los siguientes resultados sobre los datos de validación:

Coeficiente R^2 (%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto NIP $(\frac{W}{m^2})$	Error Absoluto CHP1 $(\frac{W}{m^2})$
0	169.707,2	231,24	231,42

Tabla 3.1 Resultados del Perceptrón Simple

Estos resultados hacen descartar esta estructura de red, que se ve incapaz de aprender sobre ningún dato ni ajustar sus pesos a un valor determinado, como se puede ver en la siguiente gráfica que representa la evolución del coeficiente R^2 a lo largo de las épocas de entrenamiento.

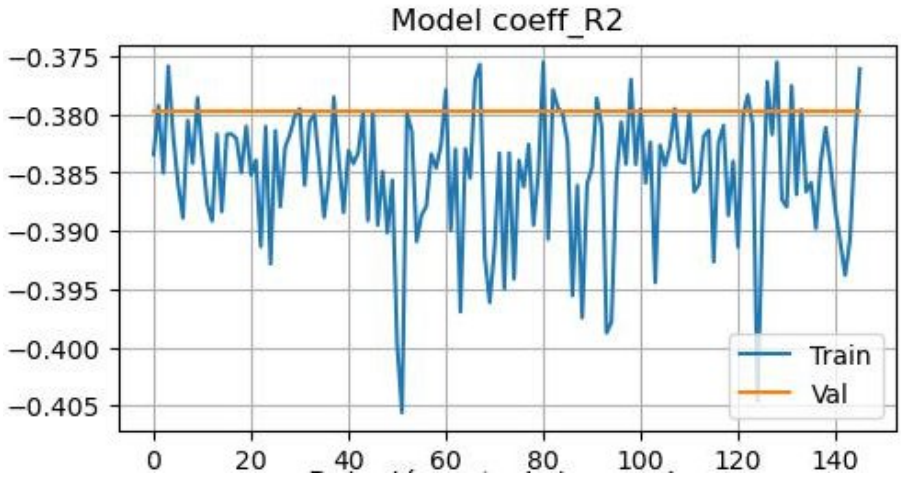


Figura 3.2 Coeficiente R2 en el Perceptrón Simple

Las siguientes pruebas se harán variando la estructura desde una hasta tres capas ocultas, con diferentes números de neuronas cada una de ellas. Por limitaciones de memoria de la GPU, la estructura de mayor tamaño posible será de tres capas con 3072, 3072 y 1536 neuronas respectivamente, lo que suma un total de 103.827.458 parámetros de entrenamiento. Los resultados obtenidos son los siguientes:

Tamaño de la Red	Coeficiente R^2(%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto NIP $(\frac{W}{m^2})$	Error Absoluto CHP1 $(\frac{W}{m^2})$
768	74	29.471,22	88,05	88,41
1536	17	96.701,59	231,24	76,9
3072	8	22.693,18	75,15	76,17
768 x 768	18	95.401,21	231,22	66,02
768 x 1536	17	96.846,3	76,42	229,36
1536 x 768	75	28.945,98	85,56	85,85
1536 x 1536	80	22.162,28	68,07	69,37
1536 x 3072	81	21.178,86	64,3	64,67
3072 x 768	78	25.938,4	77,59	77,4
3072 x 1536	82	20.651,68	64,57	64,85
3072 x 3072	81	22.118,0	68,85	68,69
768 x 1536 x 3072	82	20.667,98	61,38	61,96
1536 x 1536 x 768	14	99.656,66	231,24	88,16
1536 x 1536 x 1536	82	20.065,36	60,11	60,98
1536 x 1536 x 3072	19	94.799,95	60,8	231,42
1536 x 3072 x 1536	83	19.303,28	58,04	58,64

1536 x 3072 x 3072	80	23.154,14	75,92	75,12
3072 x 1536 x 768	79	24.127,84	72,33	71,73
3072 x 1536 x1536	83	19.625,81	59,79	60,13
3072 x 1536 x 3072	82	20.975,91	62,31	62,58
3072 x 3072 x 1536	0	169.707,2	232,56	231,29

Tabla 3.2 Resultados variando la estructura interna

Para resumir estos datos, podemos ver como influye el tamaño de las redes de una, dos y tres capas en la siguiente gráfica:

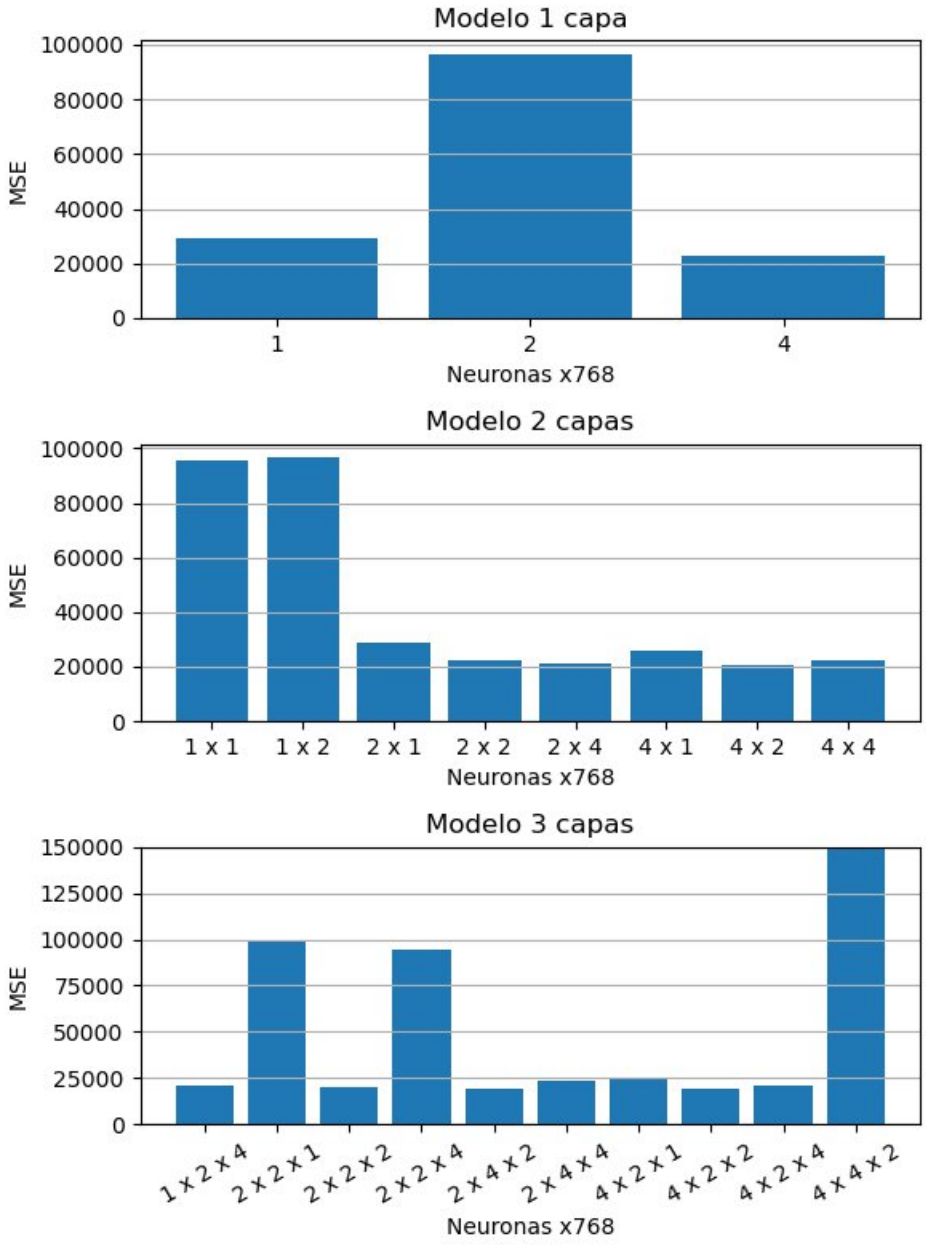


Figura 3.3 Error cuadrático medio según la estructura de la red

A la vista de los resultados, se observa que tanto un pequeño número de neuronas como un número elevado de estas no producen resultados beneficiosos, ya que el error interesa que sea el menor posible.

Por ello, se escogen dos de las estructuras que arrojan mejores resultados, denominadas:

1. Modelo rombo (R): 1536 x 3072 x 1536 neuronas
2. Modelo delta (D): 3072 x 1536 x 1536 neuronas

A partir de estos dos modelos, cuyo coeficiente de determinación llega a alcanzar el 83%, se realizarán las pruebas sobre los demás parámetros que intervienen en el ajuste.

3.1.1.2 Optimizador

El siguiente parámetro será el optimizador, cuya influencia es muy notable en el resultado final, pero cobra también una gran importancia a la hora de tener en cuenta el tiempo total de computación, ya que según el optimizador y su algoritmo interno, será capaz de alcanzar más o menos rápido la convergencia del modelo.

Se probarán los siguientes optimizadores: Adam, Adamax, Adadelta, Stochastic Gradient Descendent y RMSprop.

Los resultados para cada uno de los modelos son los siguientes:

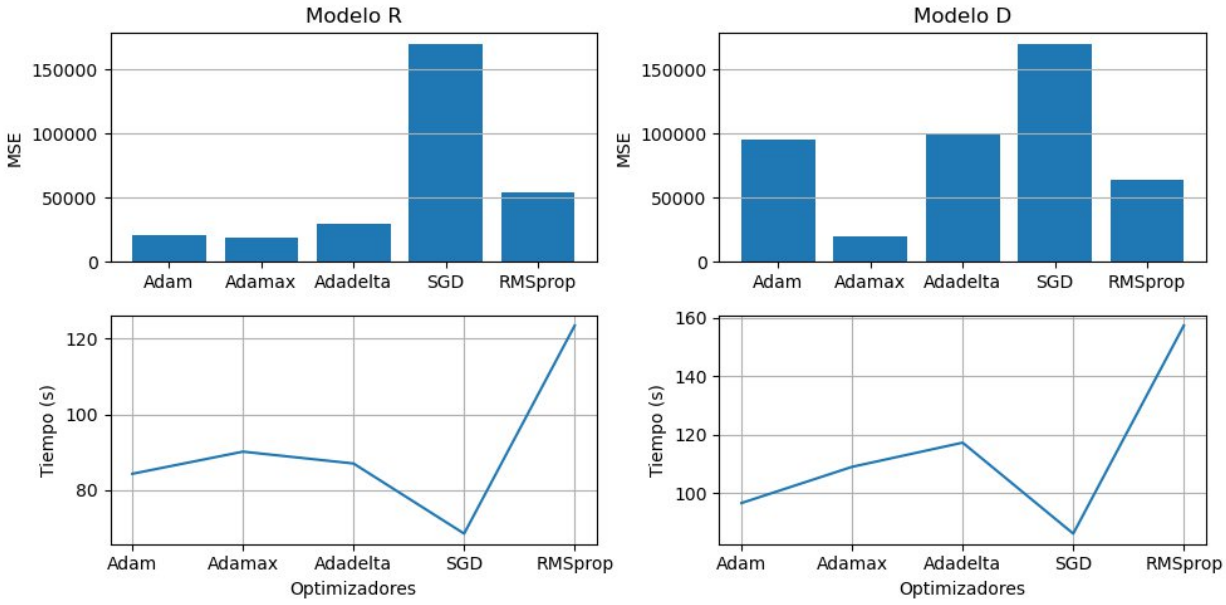


Figura 3.4 Error cuadrático medio y tiempo de entrenamiento según optimizador

Observando las gráficas podemos concluir lo siguiente:

1. El optimizador SGD, al ofrecer un MSE bastante alto y un tiempo de entrenamiento mucho menor, sugiere que es propicio a caer en mínimos locales, lo cuál tiene sentido al tratarse de un optimizador sin tasa de aprendizaje adaptativa.

2. RMSprop, a pesar de no ofrecer un mal resultado, es el que mayor tiempo de entrenamiento requiere.

Los tres optimizadores adaptativos, a pesar de observarse claramente en el Modelo D su diferencia, tienen un comportamiento similar en el Modelo R. Para valorar adecuadamente su rendimiento, habrá que guiarse de las gráficas que se recogen durante el entrenamiento de este modelo.

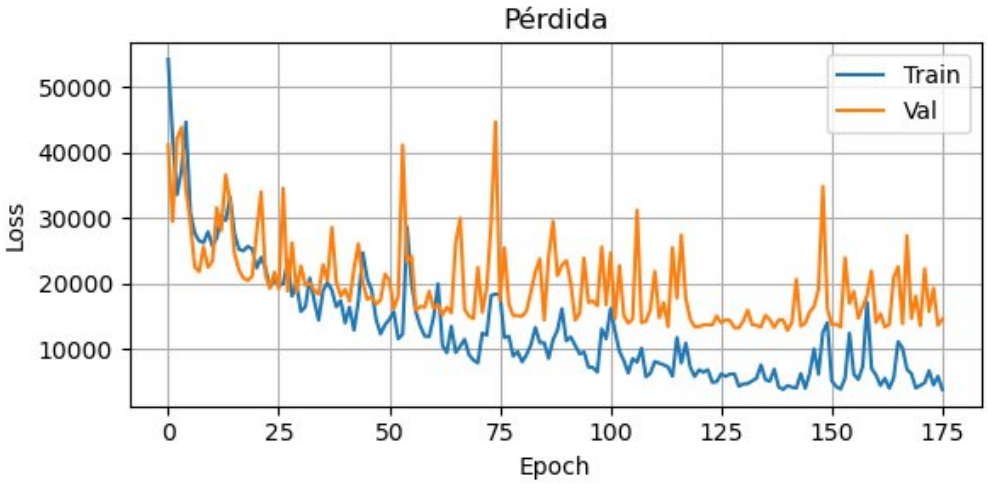


Figura 3.5 Función de pérdidas durante el entrenamiento con optimizador Adam

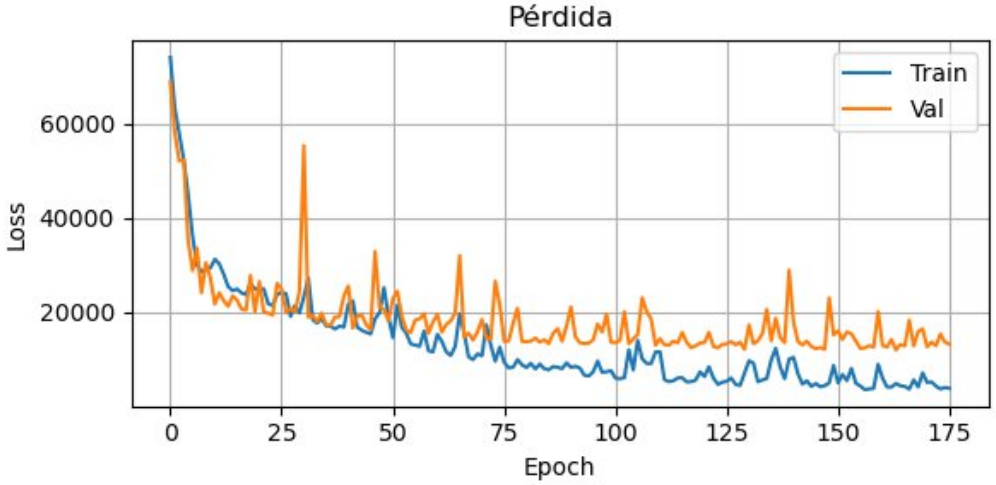


Figura 3.6 Función de pérdidas durante el entrenamiento con optimizador Adamax

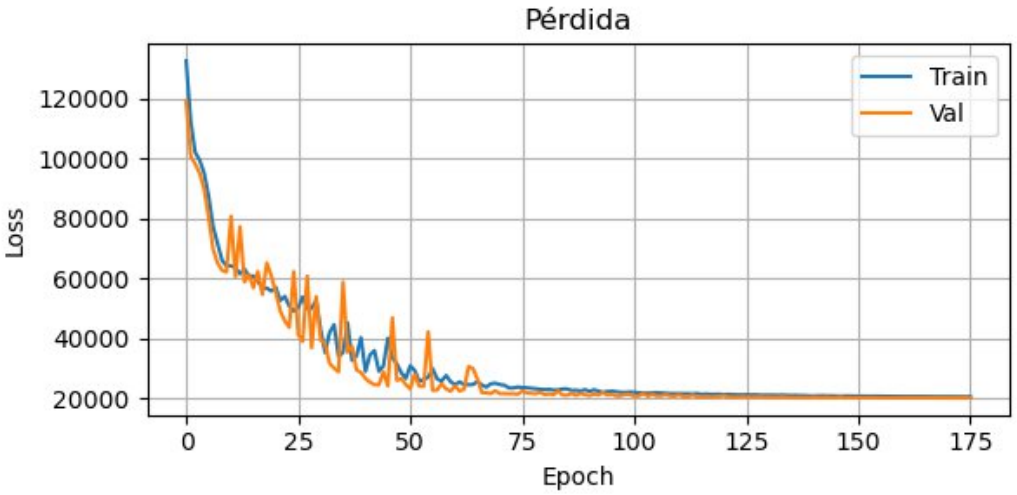


Figura 3.7 Función de pérdidas durante el entrenamiento con optimizador Adadelta

Se observa lo siguiente:

1. El optimizador Adadelta ofrece una convergencia limpia que se mantiene constante sin hacer notar señales de *overfitting*, sin embargo, su función de pérdidas no es capaz de alcanzar un valor inferior a 20.000.

2. Entre Adam y Adamax, ambos muestran señales de *overfitting* al notarse una separación entre sus curvas de entrenamiento y validación. Sus resultados finales de error son similares, aunque el optimizador Adam presenta una mayor oscilación, lo que indica una mayor incapacidad para encontrar un intervalo de adaptación de la tasa de aprendizaje.

Estos resultados hacen decantarse definitivamente por el optimizador Adamax, el cuál ofrece buen rendimiento para los dos modelos en cuestión.

3.1.1.3 Batch Size

Las siguientes pruebas se harán para ver la influencia que tiene el tamaño del *Batch*. El ajuste de este parámetro se hará en búsqueda de un equilibrio entre el tiempo de entrenamiento, la estabilidad y precisión de los resultados.

Tras una nueva batería de pruebas con los modelos *R* y *D*, se obtienen estos resultados:

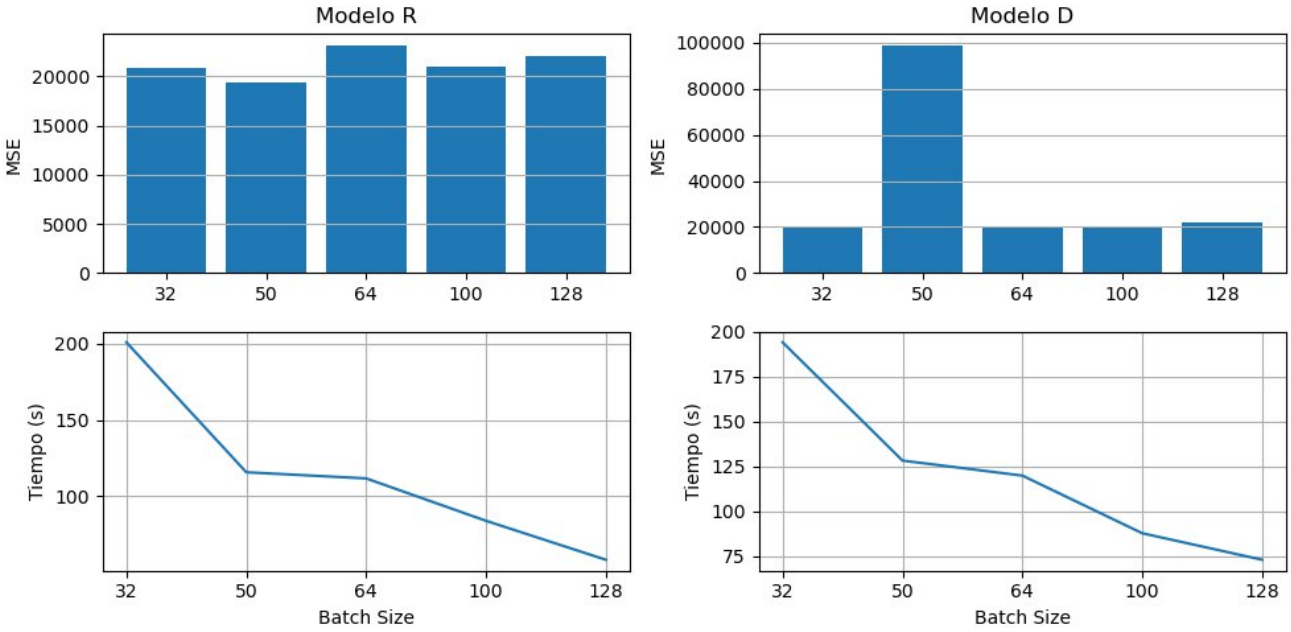


Figura 3.8 Error cuadrático medio y tiempo de entrenamiento según tamaño de Batch

Observando los resultados vemos que este parámetro no tiene gran influencia en la medida final del error, a excepción de un tamaño de 50 para el modelo D; sin embargo, tiene una influencia muy notoria en el tiempo de entrenamiento, que hace que cobre importancia para tener en cuenta su ajuste.

El motivo de este cambio en los tiempos de ejecución se debe a que el *Batch Size* determina el tamaño de los lotes de datos que se le irán pasando a la red para el ajuste de sus pesos, por lo que a mayor tamaño, menor número de iteraciones son necesarias en cada época. Asimismo, se deberá buscar un equilibrio con la capacidad de memoria disponible de la GPU, ya que los lotes de datos de grandes tamaños ocupan más memoria. En nuestro caso, un *Batch Size* superior a 128 provocaba fallos de ocupación de VRAM, por lo que se restringe el rango de tamaños a este valor.

Puesto que el interés radica en minimizar la función de pérdidas, se elegirán los siguientes valores para los modelos dejando un poco atrás el tiempo de ejecución, que al tratarse en este caso de segundos, no cobra gran importancia:

- *Batch Size* de 50 para el Modelo R
- *Batch Size* de 100 para el Modelo D

3.1.1.4 Dropout

Con todos los parámetros estudiados se decide optar por la siguiente configuración, que ha arrojado los mejores resultados en las pruebas:

- 64 x 64 píxeles
- *Batch Size* de 100
- Optimizador Adamax
- Función de pérdidas MSE

Esta misma configuración será utilizada por dos modelos con estructura diferente:

1. Modelo R: Tres capas ocultas, un tamaño de 1536 x 3072 x 1536 neuronas con un total de 28.320.770 parámetros de entrenamiento y *Batch Size* de 50.

2. Modelo D: Tres capas ocultas, un tamaño de 3072 x 1536 x 1536 neuronas con un total de 44.835.842 parámetros de entrenamiento y *Batch Size* de 100.

Se realiza un entrenamiento con cada modelo para ver, tanto de forma numérica como gráfica, el resultado que arrojan en las predicciones:

Modelo	Coeficiente R^2 (%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
Modelo R	83	19.168,21	57,04	98,415
Modelo D	82	19.761,9	59,35	82,386

Tabla 3.3 Resultados de los Modelos R y D

Se muestran las gráficas de entrenamiento, donde se representan la función de pérdidas y el coeficiente de determinación:

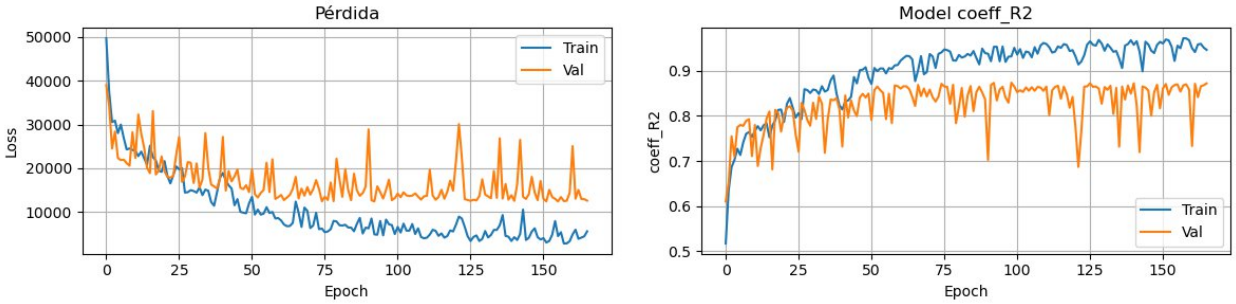


Figura 3.9 Comportamiento durante el entrenamiento del Modelo R

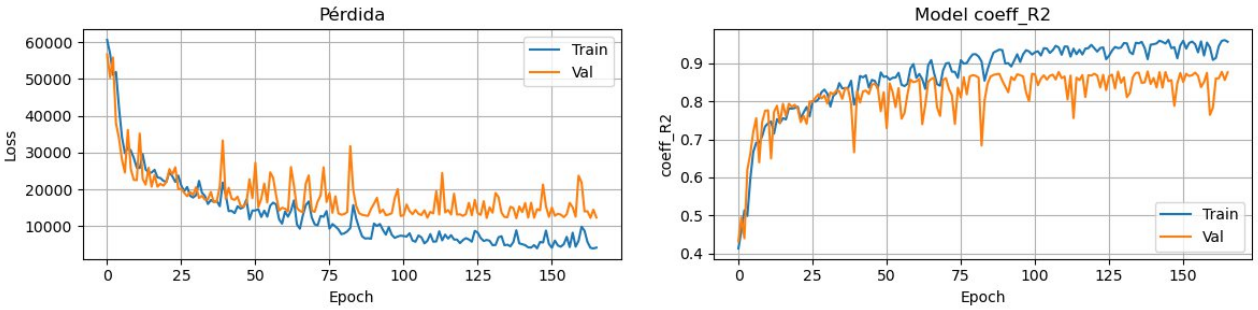


Figura 3.10 Comportamiento durante el entrenamiento del Modelo D

En estas gráficas podemos observar como, a pesar de no ser malos resultados, es notable la presencia de *overfitting*. Esto se puede respaldar viendo los diagramas residuales, en los que observamos que las predicciones sobre los datos de entrenamiento se ajustan mejor a la línea de regresión que las predicciones sobre los datos de prueba:

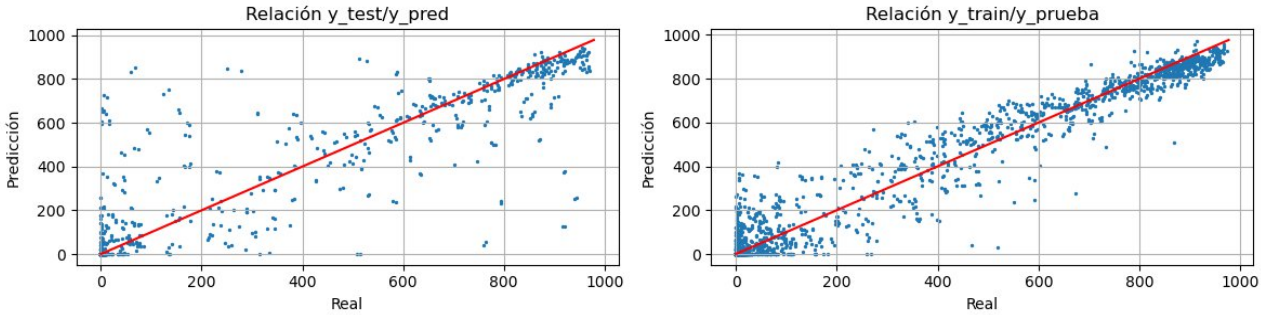


Figura 3.11 Diagrama residual del Modelo R

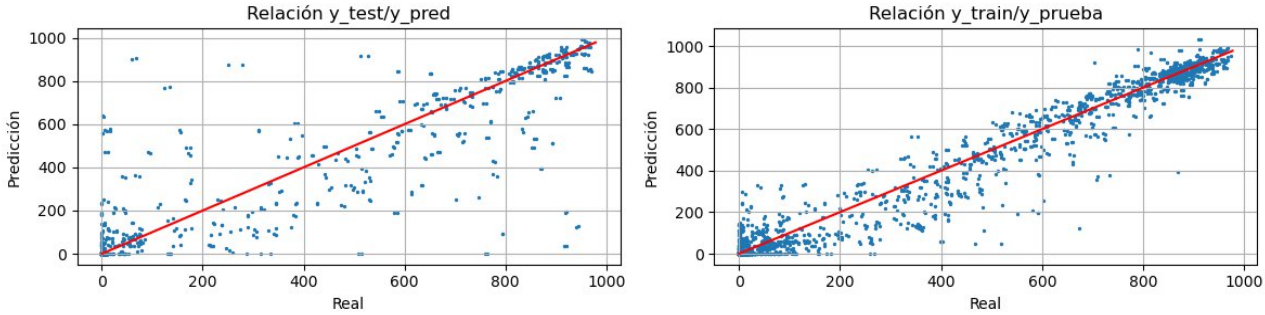


Figura 3.12 Diagrama residual del Modelo D

Este efecto indeseado se puede intentar corregir añadiendo una capa de *Dropout* a la salida del bloque de capas ocultas de la red. Se probarán distintos valores para valorar su efecto:

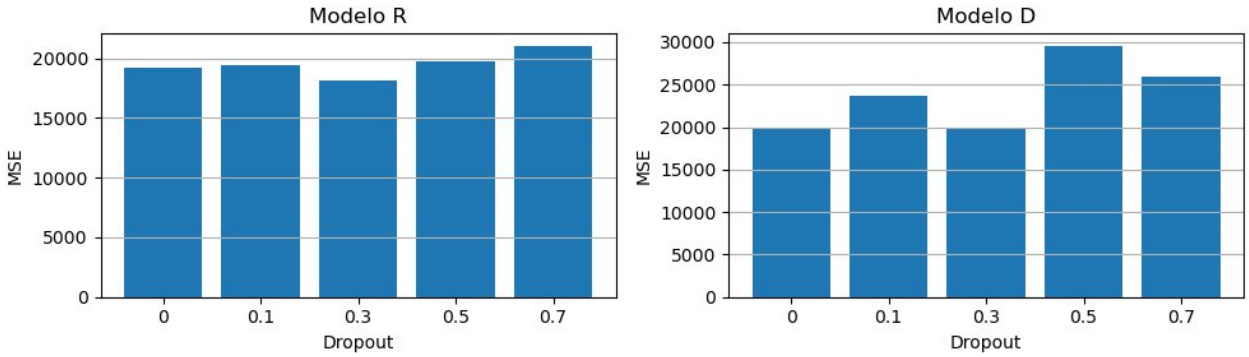


Figura 3.13 Efecto del Dropout en el error cuadrático

Se observa como un valor alto de *Dropout* puede hacer que la red pierda rendimiento al inutilizar demasiadas neuronas en cada época. Este método debe ser utilizado con cierto cuidado para notar sus beneficios, siendo un valor del 30% de neuronas inutilizadas la mejor opción para reducir en parte el efecto del del *overfitting* sin perder eficacia a la hora de minimizar el error en la predicción.

Tras este estudio realizado se concluye con que el modelo R, con una estructura en forma de rombo en el interior de sus capas ocultas, presenta una mejor eficiencia a la hora de predecir la radiación directa.

Al aplicarle un *Dropout* del 30% se puede observar como reduce el efecto del sobreajuste:

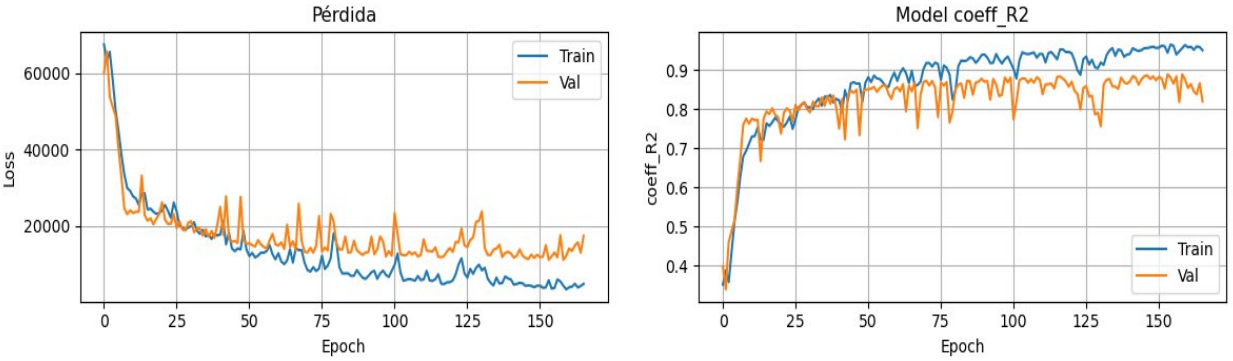


Figura 3.14 Comportamiento del Modelo R tras aplicar Dropout del 30%

3.1.2 Resultados

Para la evaluación final del modelo se realiza una predicción utilizando el modelo escogido, ya entrenado, sobre los cuatro días completos separados al inicio, lo que da una idea de la eficacia real de la aplicación de este modelo en el caso de estudio.

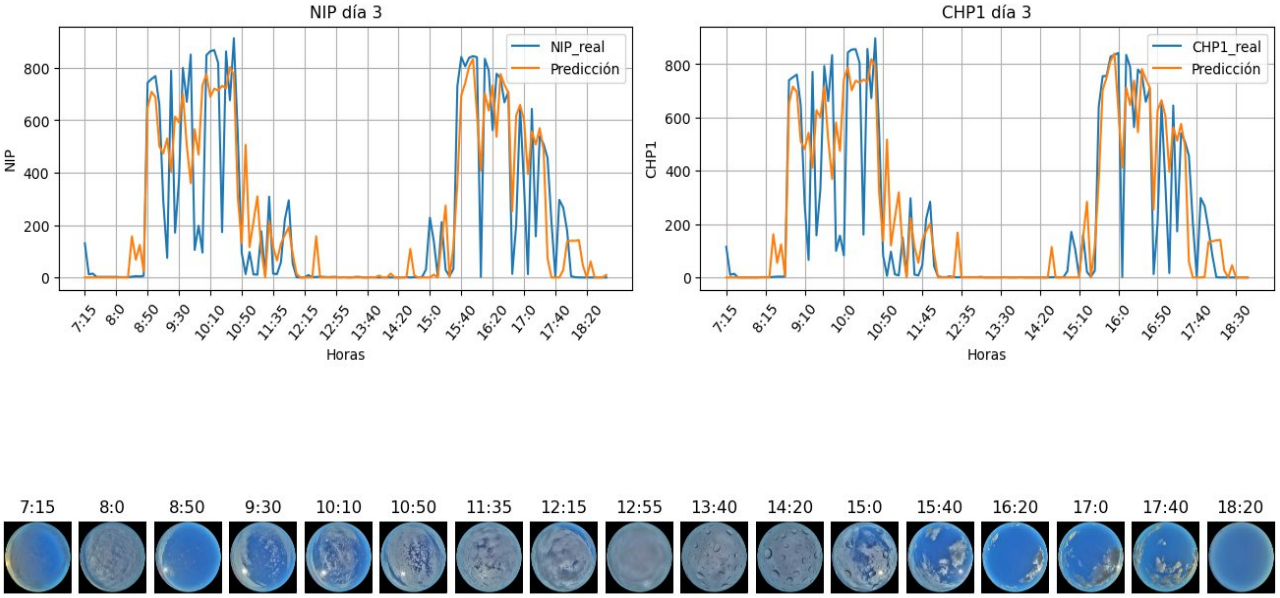


Figura 3.15 Evolución temporal del día 03/03/2022 con MLP

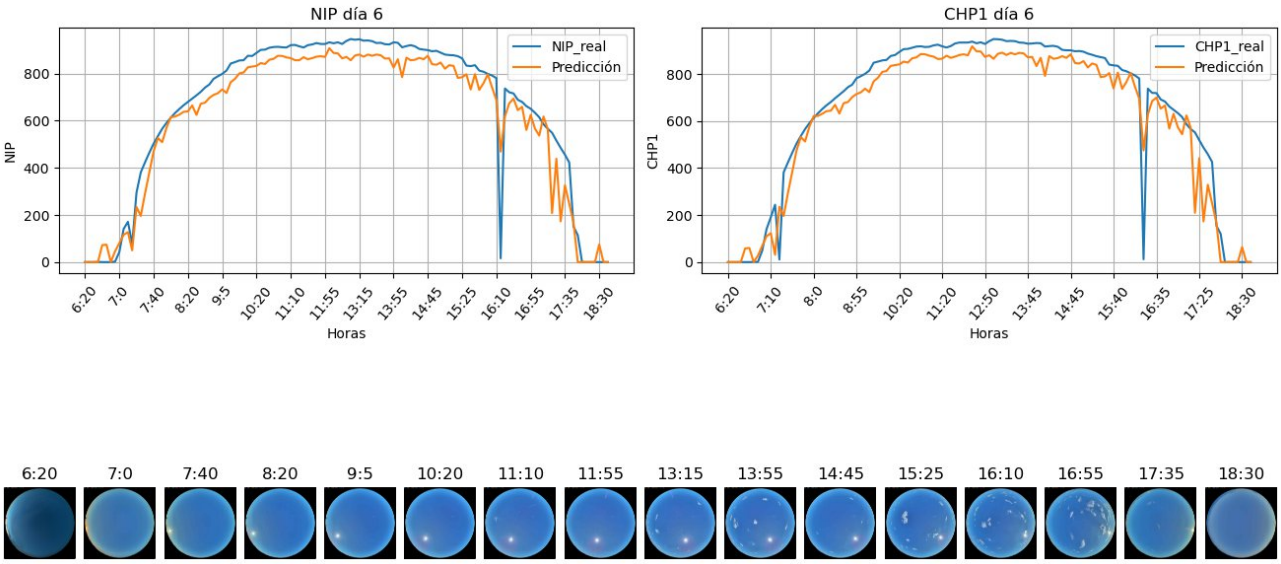
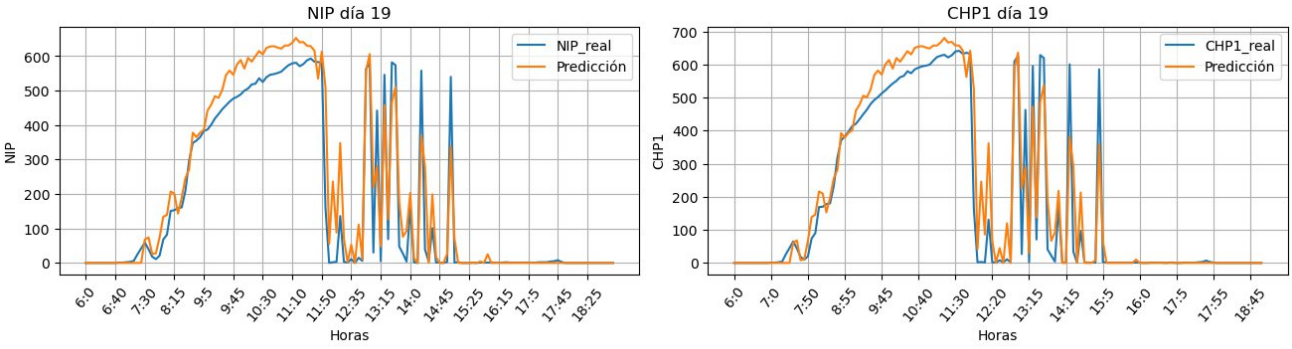


Figura 3.16 Evolución temporal del día 06/03/2022 con MLP



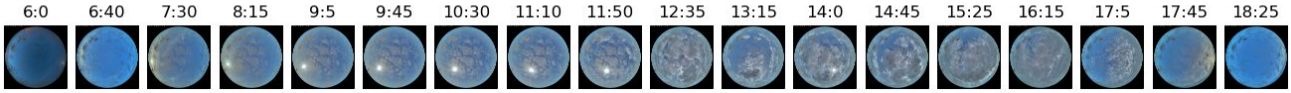
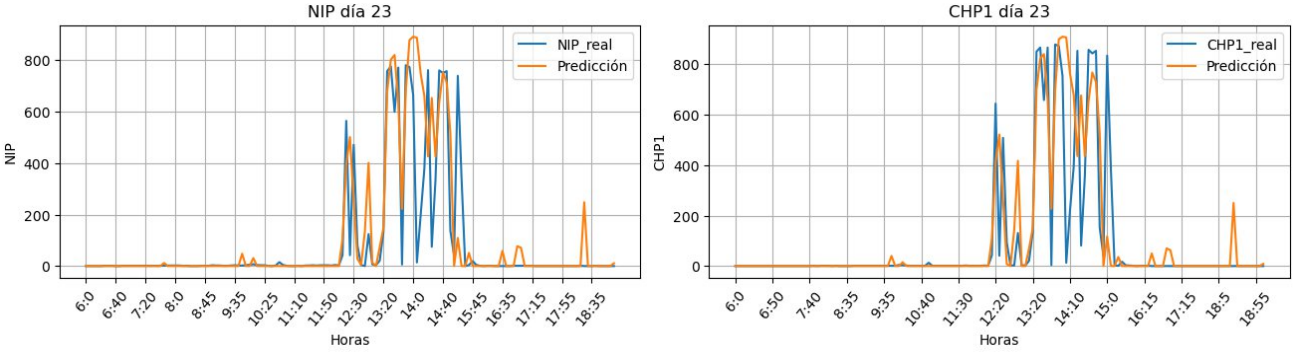


Figura 3.17 Evolución temporal del día 19/03/2022 con MLP



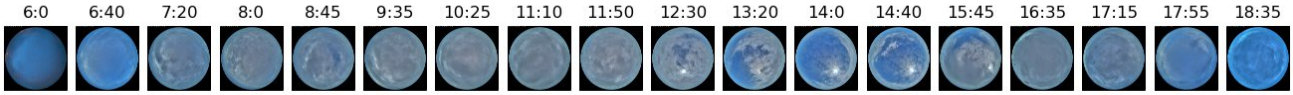


Figura 3.18 Evolución temporal del día 23/03/2022 con MLP

Se observa como el MLP es capaz de seguir la dinámica de la evolución de la radiación a lo largo del día, sin embargo, al tratarse de datos reales que no están preparados para un uso académico, el modelo no llega a ofrecer un resultado aceptable para el caso de aplicación, ya que el error cometido por esta es considerable para un uso real.

Esto se debe en parte a la incapacidad de un MLP de relacionar características en datos provenientes de un dataset formado únicamente por imágenes, donde se ve limitada la capacidad de la red a un coeficiente de determinación máximo del 83% y un valor mínimo del error medio de 57 W/m^2 en las mediciones.

Por ello en el siguiente capítulo se probará la eficacia de una RNC, que ofrece un mayor rendimiento al tratar con imágenes.

3.2 Red Neuronal Convolucional

En esta sección se estudiará el rendimiento que tiene una RNC, modelo apto para enfrentarse a problemas en los que están involucradas imágenes.

3.2.1 Desarrollo

En primer lugar, se realiza una batería de pruebas con diferentes combinaciones de parámetros para escoger una configuración base sobre la que hacer las pruebas, teniendo en cuenta las siguientes consideraciones:

1. Se aplicará un *Dropout* del 30% durante todas las pruebas para reducir el efecto del sobreajuste.

2. La escala de las imágenes será de 64 x 64 píxeles.

3. Debido a la estructura de este tipo de modelo de red neuronal y la forma que tiene de realizar los cálculos matriciales, hay que mirar muy de cerca la ocupación de memoria de la GPU.

Los resultados arrojados son los siguientes:

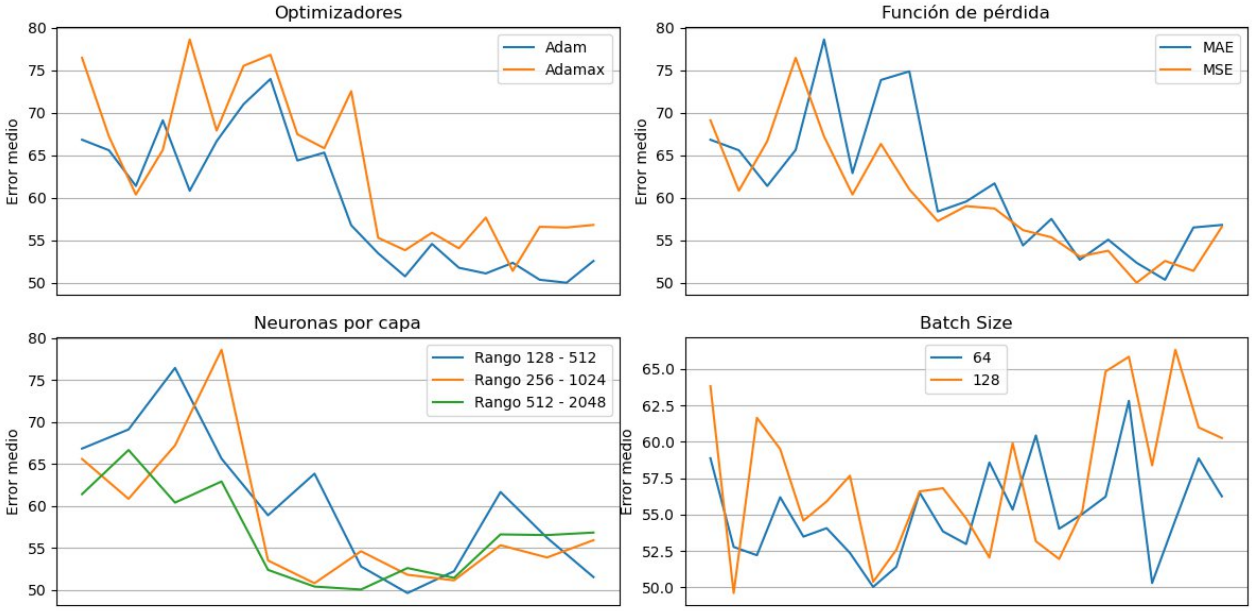


Figura 3.19 Error medio en batería de pruebas

A la vista de estos resultados podemos concluir lo siguiente:

1. En RNC, el optimizador que ofrece mejor rendimiento es Adam.

2. Se seguirá usando como función de pérdida el error cuadrático medio.

3. Las capas con mayor número de neuronas, limitado por la capacidad de la GPU, presentan un mejor comportamiento.

4. La influencia del *Batch Size* no se ve tan clara en este caso, por lo que se escogerá un tamaño de 128, por lo general, para priorizar el tiempo de ejecución; no obstante habrá casos en los que sea necesario escoger un valor más pequeño para no saturar la memoria de GPU.

Una vez configurado el modelo base, se realiza el análisis de la estructura convolucional que define a este tipo de red. Este se hará iniciando con una única capa de convolución de tamaño igual a la escala de la imagen, y a partir de esta se irán añadiendo más capas. Las combinaciones se harán en conjunto con las capas ocultas de la red, probándose estructuras de dos y tres capas de diferente número de neuronas.

Se obtienen los siguientes resultados:

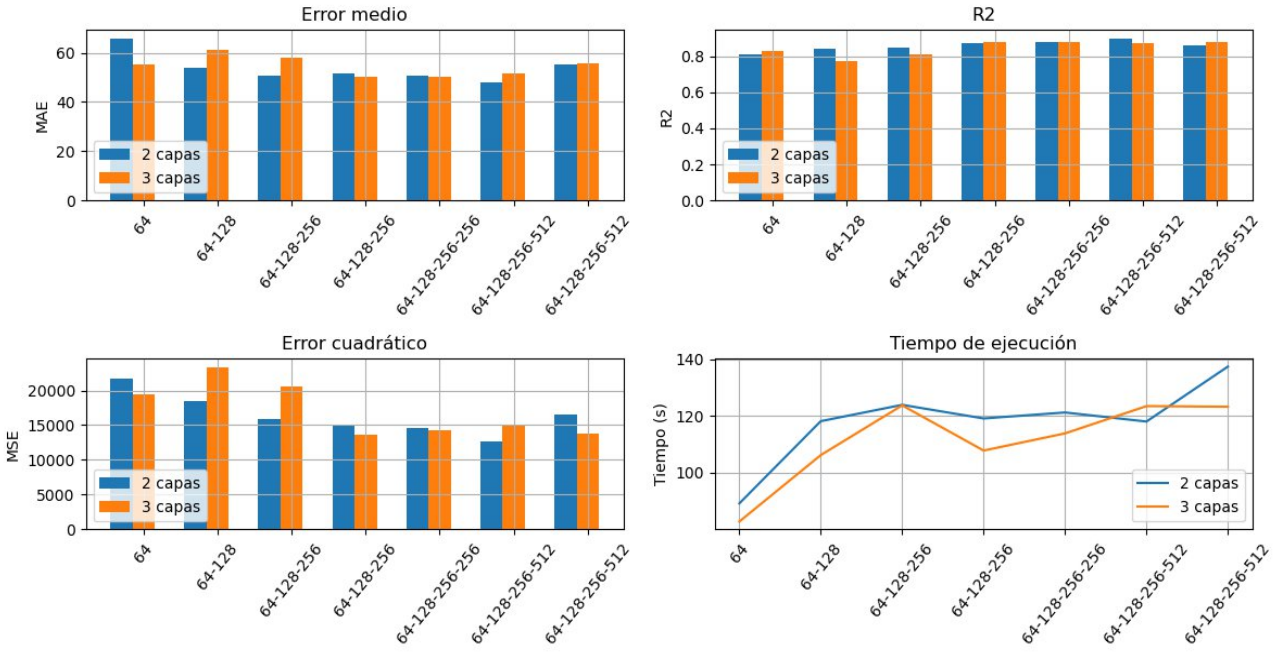


Figura 3.20 Resultados de las pruebas frente a la estructura convolucional

En estos resultados se puede observar que a mayor número de capas convolucionales se mejora el rendimiento del modelo. El tamaño de la estructura se ve limitado por el tamaño de la imagen, ya que tras cada capa de convolución y *Max-Pooling* por la que pasa la imagen, esta se reduce en tamaño, como se puede ver en la Figura 3.1, donde viene detallada la estructura de la RNC. Se ve que a la última capa de convolución llegan 512 filtros de tamaño 2 x 2, por lo que no existe la posibilidad de aplicar más capas.

Esto se solucionaría utilizando imágenes de mayor tamaño en la entrada, lo que aumenta la ocupación de memoria y reduce su viabilidad usando tarjetas gráficas comunes.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 64)        1792

 max_pooling2d (MaxPooling2D  (None, 31, 31, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 29, 29, 128)       73856

 max_pooling2d_1 (MaxPooling  (None, 14, 14, 128)      0
 2D)

 conv2d_2 (Conv2D)           (None, 12, 12, 256)       295168

 max_pooling2d_2 (MaxPooling  (None, 6, 6, 256)        0
 2D)

 conv2d_3 (Conv2D)           (None, 4, 4, 512)         1180160

 max_pooling2d_3 (MaxPooling  (None, 2, 2, 512)        0
 2D)

 dropout (Dropout)           (None, 2, 2, 512)         0

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 2048)              4196352

 dense_1 (Dense)             (None, 1024)              2098176

 dropout_1 (Dropout)         (None, 1024)              0

 output (Dense)              (None, 2)                 2050

=================================================================
Total params: 7,847,554
Trainable params: 7,847,554
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.21 Estructura de la RNC

Una vez determinado el tamaño de la capa de convolución, formado por una estructura de la siguiente forma: 64 - 128 - 256 - 512, se realizan unas últimas pruebas para ajustar el tamaño de las capas ocultas de la red. Estas se realizarán partiendo de la estructura de dos capas, que se ha visto que ofrece los mejores resultados, y se irá incrementando tanto el número de capas como de neuronas.

En la Gráfica 3.3 se puede ver el resultado de estas pruebas, donde queda reflejado que para esta configuración de RNC, el aumento de capas ocultas y su tamaño empobrecen el rendimiento.

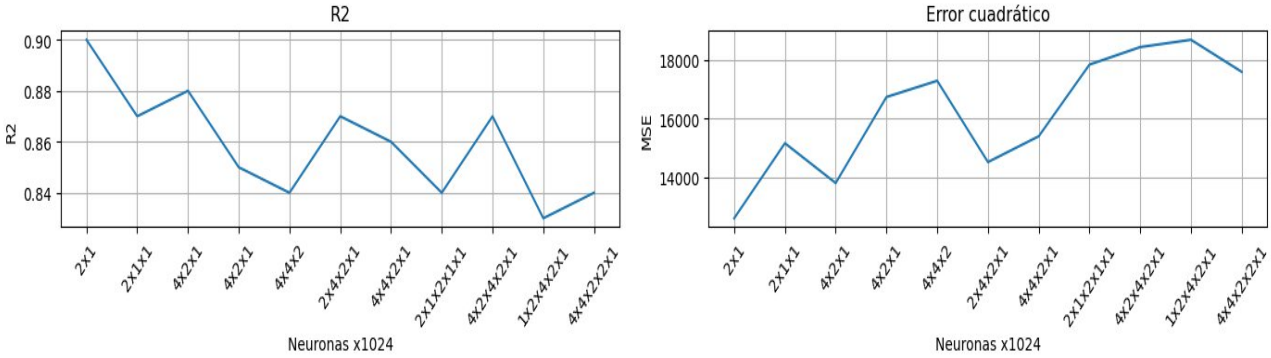


Figura 3.22 Resultados de la variación del tamaño de las capas ocultas

3.2.2 Resultados

Para realizar el análisis final de este tipo de red neuronal, se utilizará el modelo escogido para realizar predicciones sobre los cuatro días separados para evaluación real.

El modelo está definido con la siguiente configuración:

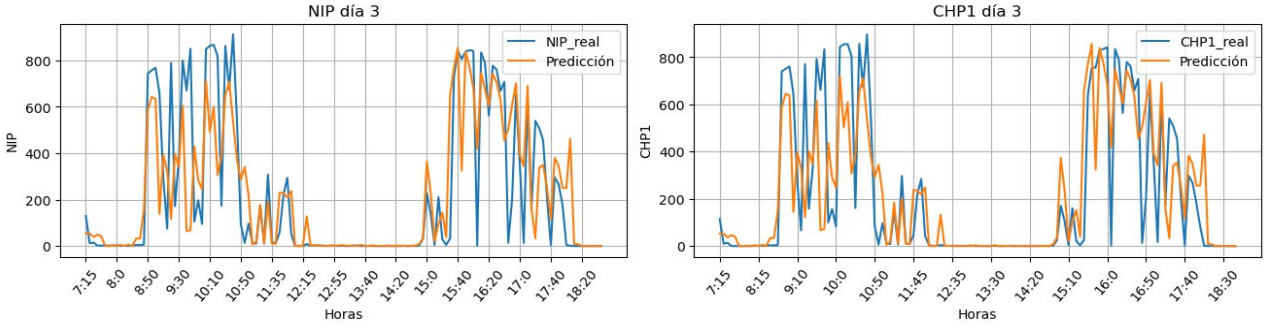
- Optimizador: Adam
- Función de pérdida: Error cuadrático medio
- Batch Size: 128
- Estructura convolucional: 64 - 128 - 256 - 512 filtros
- Estructura de capas ocultas: 2048 x 1024 neuronas

Y arroja el siguiente resultado:

Coeficiente R^2(%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
90	12.590,78	48,13	118,0815

Tabla 3.4 Resultados del modelo con RNC

Su desempeño sobre un caso real sería el siguiente:



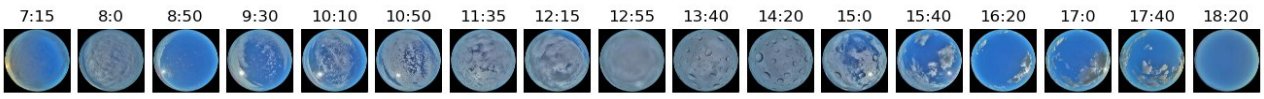


Figura 3.23 Evolución temporal del día 03/03/2022 con RNC

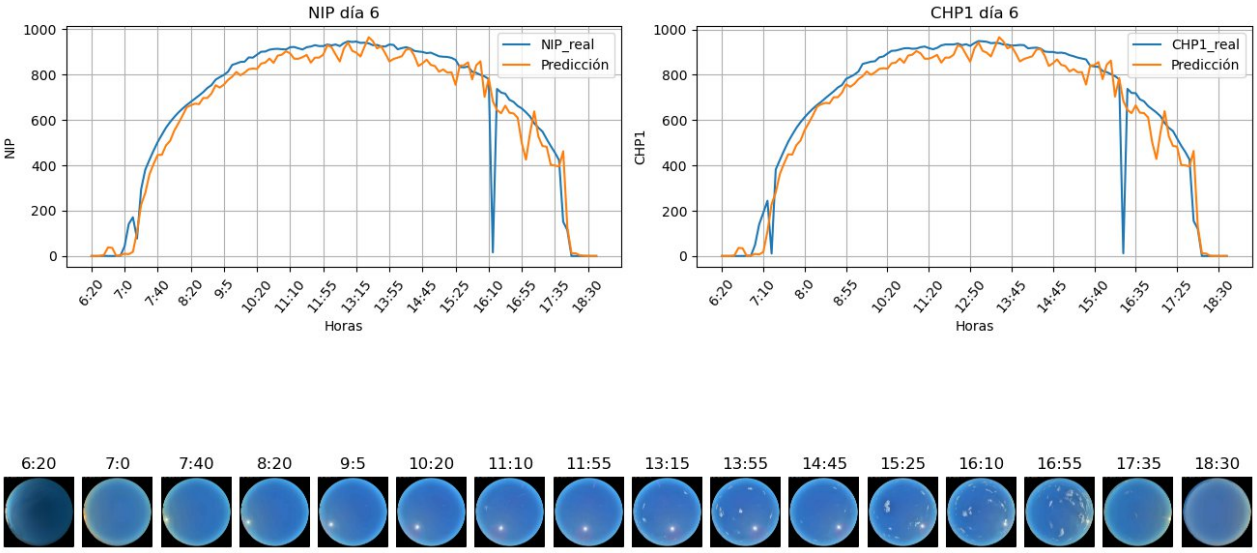


Figura 3.24 Evolución temporal del día 06/03/2022 con RNC

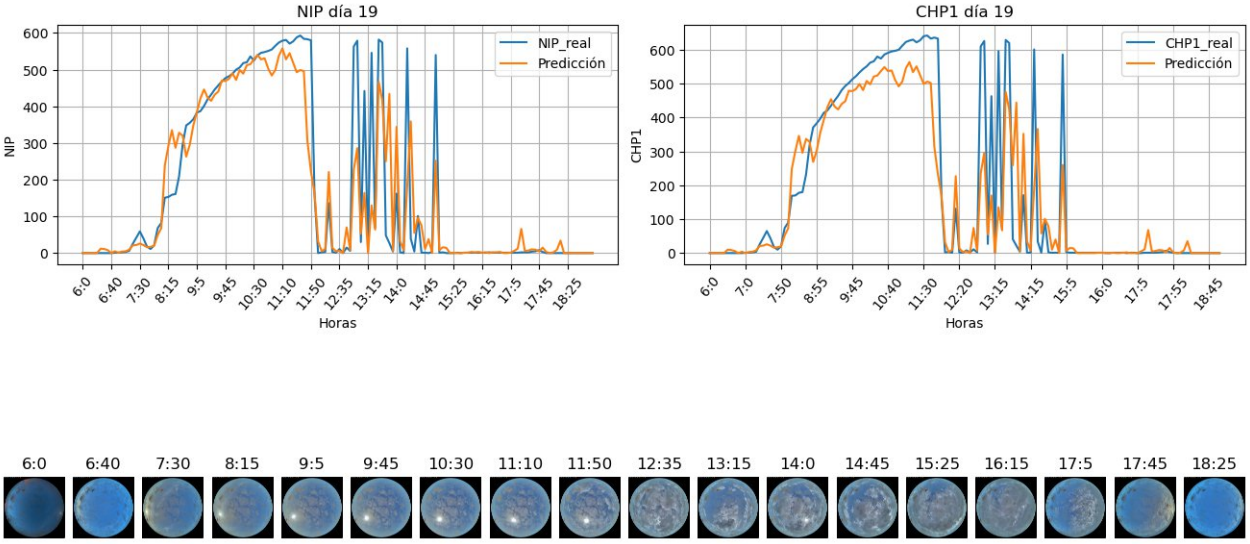


Figura 3.25 Evolución temporal del día 19/03/2022 con RNC

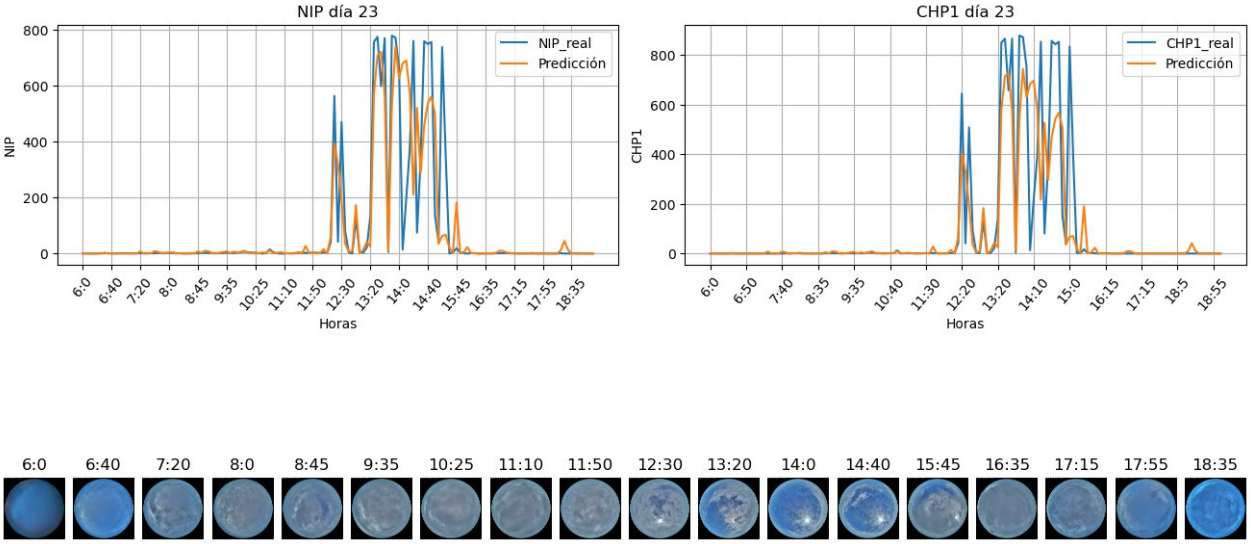


Figura 3.26 Evolución temporal del día 23/03/2022 con RNC

Con los resultados obtenidos se ha demostrado la eficacia que tiene una RNC al tratar con imágenes, arrojando unos resultados mucho mejores que el MLP. Sin embargo, como se comentó anteriormente, sigue sin ofrecer un resultado válido para un caso de aplicación.

Uno de los problemas que puede darse es la falta de capacidad para catalogar todos los datos, esto se puede ver en el diagrama residual del entrenamiento del modelo:

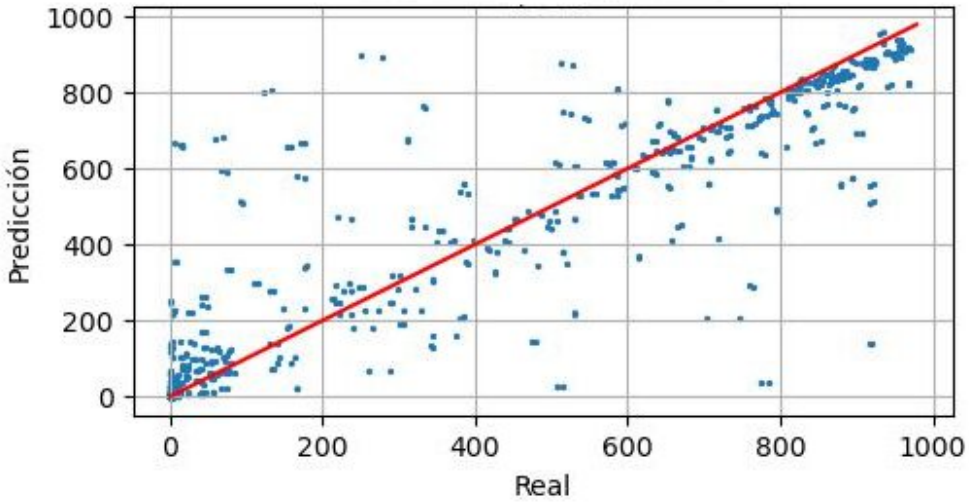


Figura 3.27 Diagrama residual RNC

En este diagrama se puede observar como el modelo ajusta las predicciones a la línea de regresión, por lo general adecuadamente; sin embargo, existen valores atípicos que influencian negativamente en la capacidad de generalizar de la red, de ahí el resultado visto a la hora de aplicarla en días cuyos datos no han pasado nunca por esta.

Una solución a este impedimento es el uso de una mayor cantidad de datos para entrenamiento, por ello en la siguiente sección se verá el rendimiento ofrecido por este modelo utilizando un *dataset* enriquecido.

3.3 Enriquecimiento del dataset

En esta sección se verá la eficacia que tiene el uso de un *dataset* enriquecido con nuevos datos, para ello se realizan una serie de transformaciones a cada imagen del *dataset* original, en concreto:

- Rotaciones de 90º, -90º y 180º.
- Invertido de imagen en los ejes vertical, horizontal y diagonal.

Estas seis transformaciones convierten al *dataset* inicial de 1713 imágenes en uno nuevo de 11995 imágenes. Tanto la imagen original como sus seis derivadas mantienen el mismo valor de radiación, y al tratarse de transformaciones que no alteran la relación entre la imagen y la radiación, permite que la red aumente su capacidad de generalización.

3.3.1 Desarrollo

En primer lugar se probará el nuevo *dataset* con el último modelo optimizado de RNC, con el que se obtienen estos resultados:

Coeficiente R^2(%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
91	8.996,45	34,47	672,7815

Tabla 3.5 Resultados del modelo con RNC usando dataset aumentado

Como es de esperar, el tiempo de ejecución se incrementa proporcionalmente al aumento del número de imágenes.

El modelo presenta una gran mejora en el valor de los errores cometidos y gana un 1% en el coeficiente de determinación. Para intentar mejorar el modelo, se añadirán más capas ocultas a la estructura de la red, sin embargo, al tratar con tal cantidad de datos las modificaciones son limitadas por la capacidad de la GPU. La estructura final tendrá un tamaño de 2048 x 1024 x 512 neuronas, la cuál arroja el mejor resultado obtenido hasta ahora:

Coeficiente R^2(%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
92	8.614,8	33,51	741,2047

Tabla 3.6 Resultados del modelo con RNC usando dataset aumentado y mayor estructura neuronal

Es notable que el enriquecimiento de datos mejora el desempeño del modelo, pero uno de sus inconvenientes es que, al tratar con mayor número de datos, el efecto del sobreajuste se hace más notable. Esto se puede observar en la evolución de la función de pérdida a lo largo del entrenamiento, Figura 3.28:

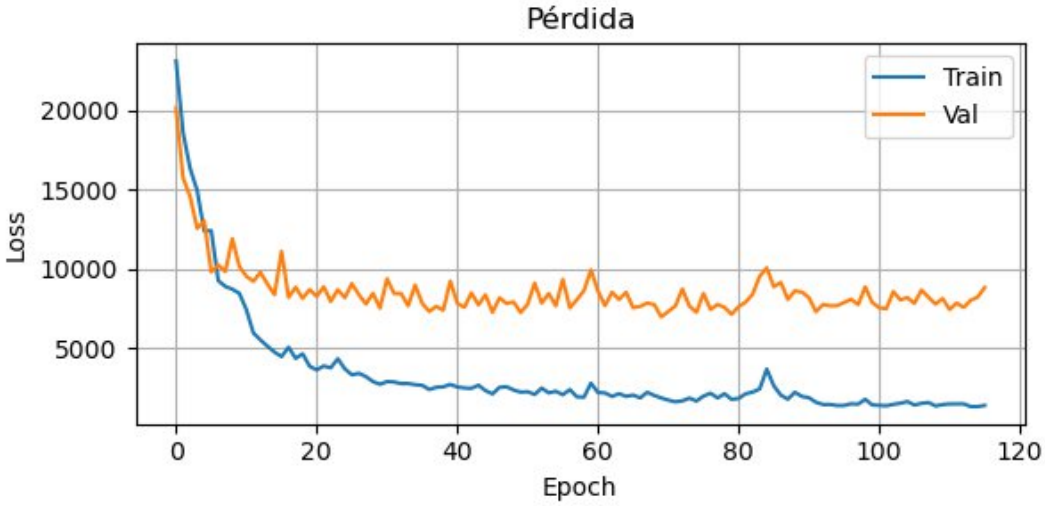


Figura 3.28 Función de pérdidas durante entrenamiento con dataset enriquecido

Este efecto se corrobora con el diagrama residual de los datos de prueba y de entrenamiento, respectivamente:

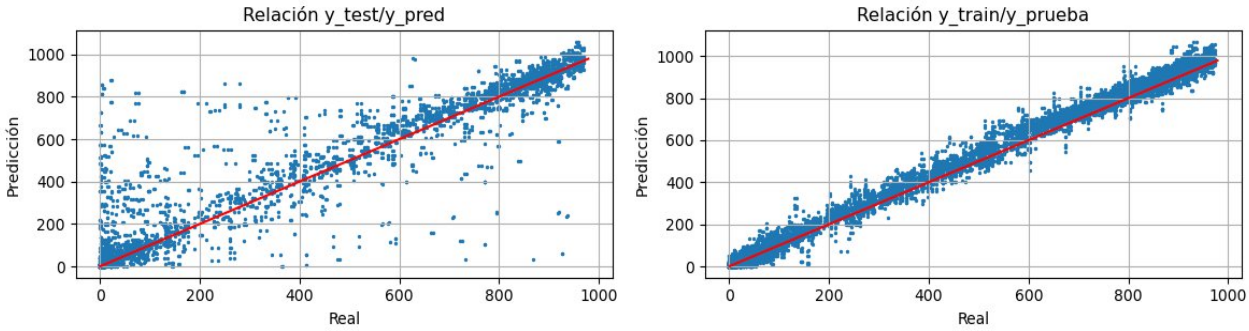


Figura 3.29 Diagrama residual de RNC con dataset enriquecido

En él se observa como la red aprende casi a la perfección sobre los datos de entrenamiento, lo que provoca pérdida de generalización a la hora de hacer predicciones sobre datos nuevos. Por ello se realizan otras pruebas aumentando el porcentaje de *dropout* al 50% y 60%:

Dropout	Coeficiente R^2	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
50%	93 %	6.628,01	29,42	832,693
60%	95 %	5.521,27	26,97	825,712

Tabla 3.7 Resultados del modelo con RNC usando dataset aumentado variando Dropout

Como vemos en estos resultados, el mismo modelo con un *dropout* del 60% ofrece el mejor resultado obtenido hasta ahora, pudiéndose ver además su efecto en la gráfica de la función de pérdida:

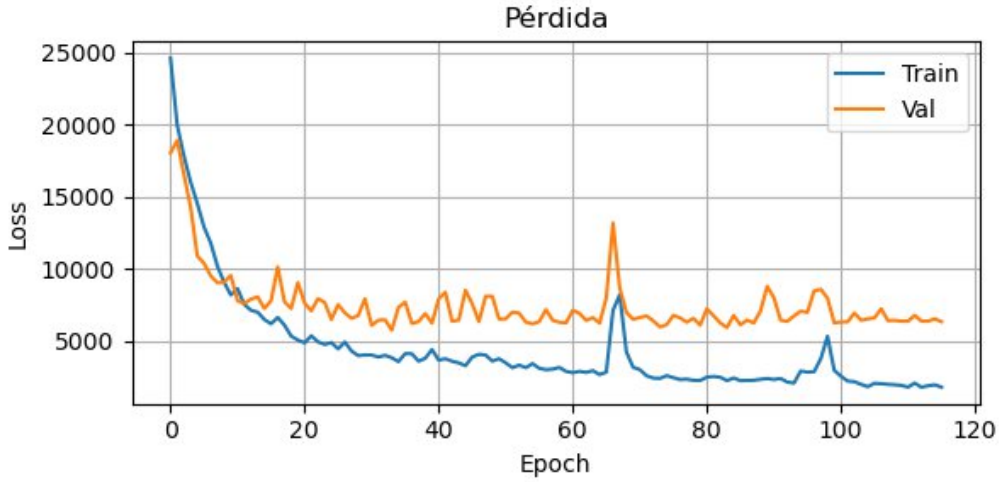


Figura 3.30 Función de pérdidas durante entrenamiento con dataset enriquecido y dropout del 60%

3.3.2 Resultados

Este modelo ha demostrado ofrecer los mejores resultados de todas las pruebas, demostrando la eficacia de utilizar una mayor cantidad de datos para entrenar la red.

Para validar estos resultados se probará el modelo en los cuatro días de prueba:

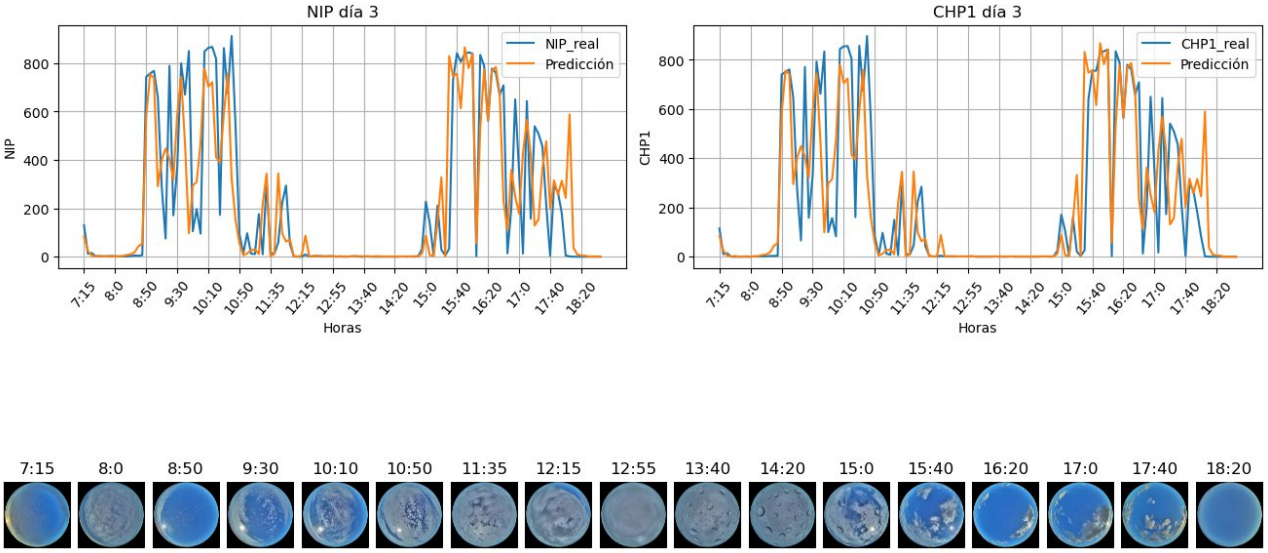


Figura 3.31 Evolución temporal del día 03/03/2022 con dataset enriquecido

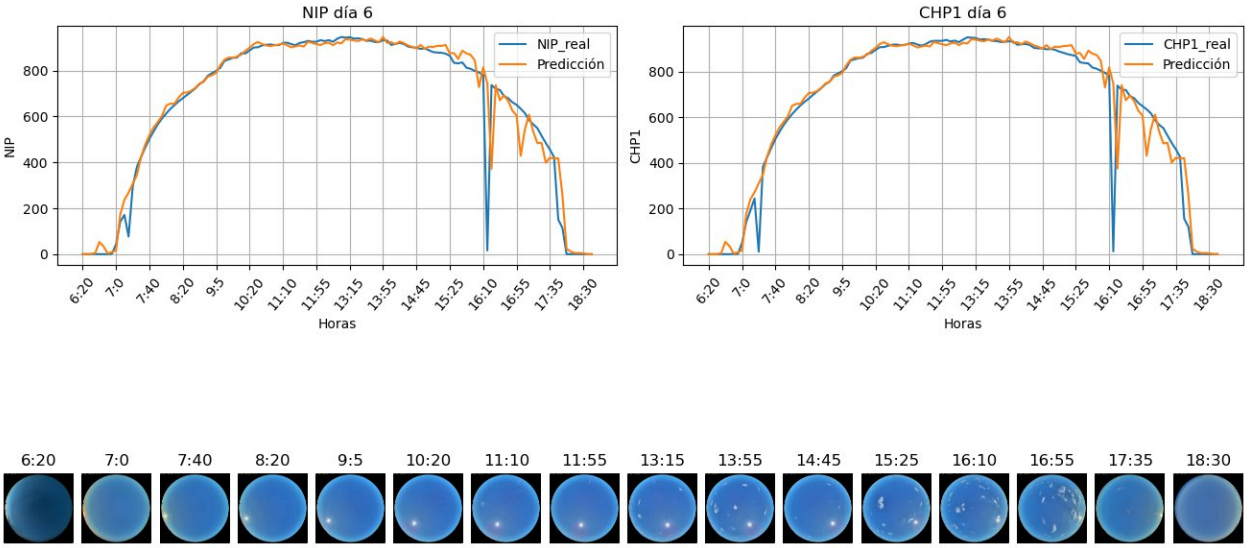


Figura 3.32 Evolución temporal del día 06/03/2022 con dataset enriquecido

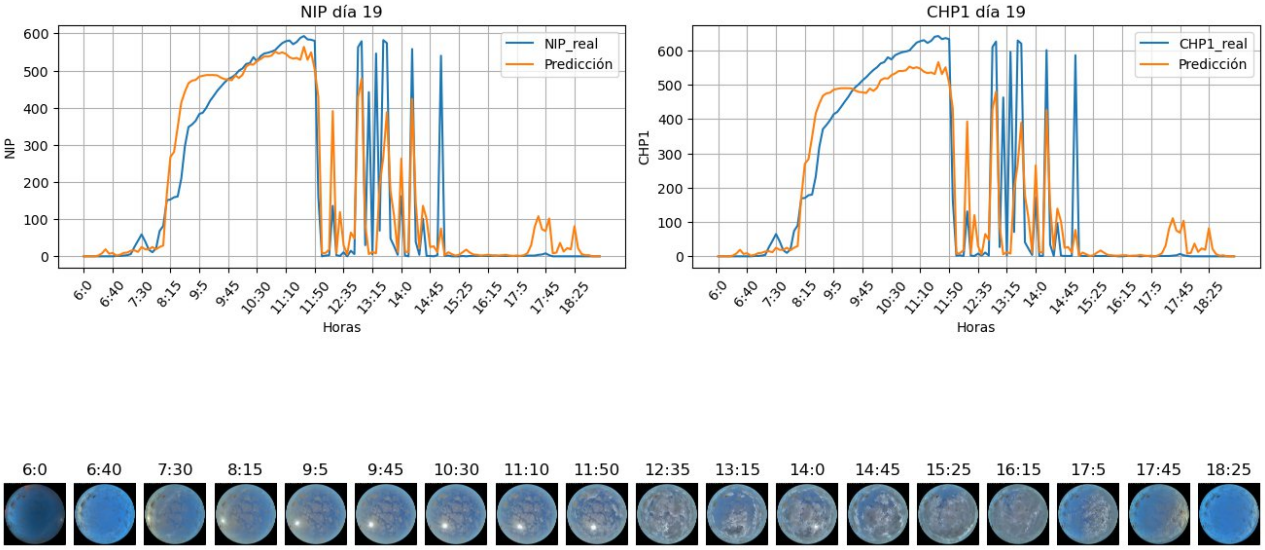


Figura 3.33 Evolución temporal del día 19/03/2022 con dataset enriquecido

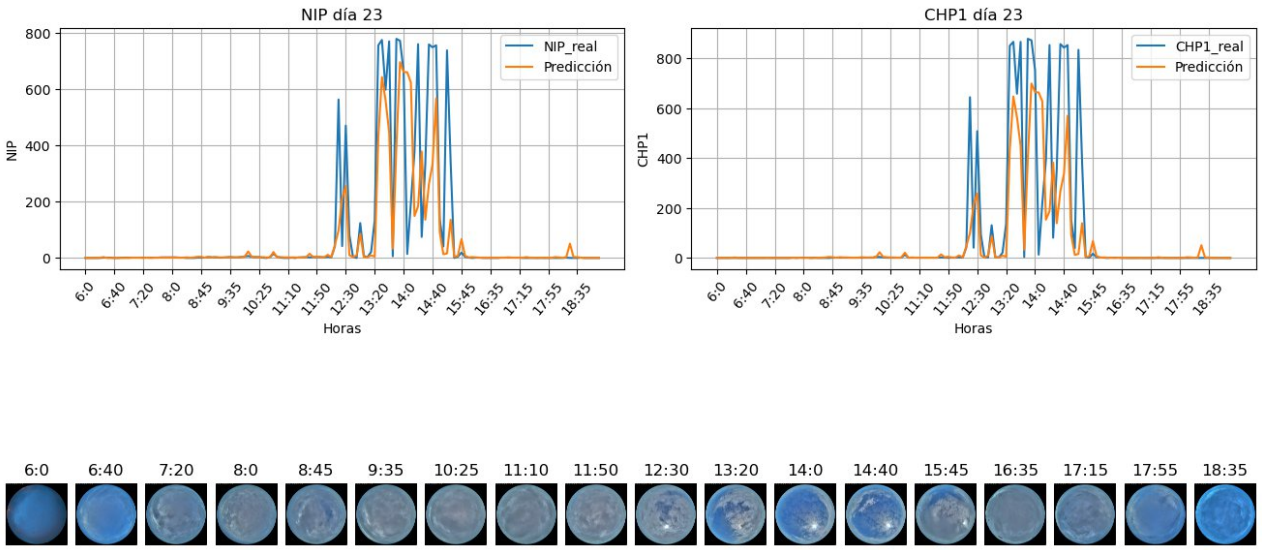


Figura 3.34 Evolución temporal del día 23/03/2022 con dataset enriquecido

Se puede ver que en este caso el modelo presenta una mayor capacidad para realizar predicciones en un caso real, en especial en días como el 6 de marzo de 2022 en los que no existe una gran cantidad de nubes que alteren los valores de radiación, donde el modelo realiza una estimación casi perfecta.

3.4 Radiación Difusa

En este capítulo se usarán los modelos anteriormente descritos para predecir la radiación difusa, lo que requiere adaptar la estructura de la red a una capa de salida de una única neurona.

La ventaja que ofrece este tipo de radiación es que presenta valores que se mantienen más robustos al paso de nubes o lluvia, por lo que no generarán gran cantidad de valores atípicos que empobrezcan el desempeño de la red.

3.4.1 Resultados

Tras adaptar los modelos para el caso en estudio, se prueban los dos tipos de redes neuronales para ver su eficacia, junto al modelo de RNC entrenado con *dataset* aumentado.

Para evaluar los resultados se cuenta tanto con las medidas empíricas como de los diagramas de residuos.

Modelo	Coeficiente R^2 (%)	Error Cuadrático Medio $(\frac{W}{m^2})^2$	Error Absoluto Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
MLP	86	2.064,04	30,67	78,95
RNC	95	739,84	18,44	117,03
RNC *Augmented*	97	581,27	14,25	801,21

Tabla 3.8 Resultados de los tres tipos de Red utilizados con Radiación Difusa

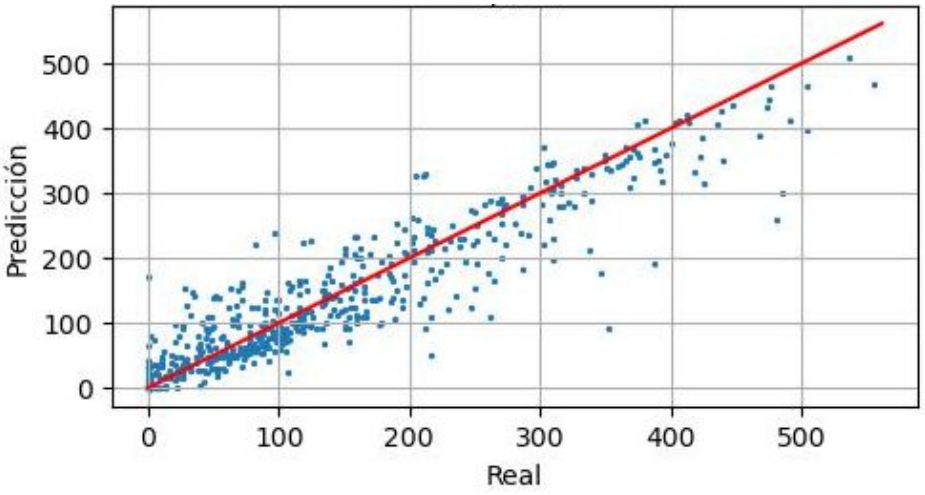


Figura 3.35 Diagrama residual para radiación difusa con MLP

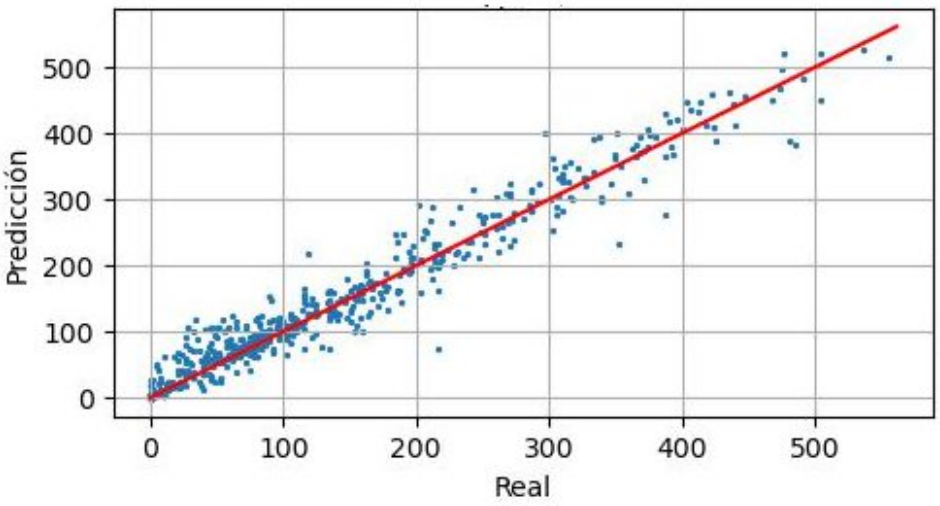


Figura 3.36 Diagrama residual para radiación difusa con RNC

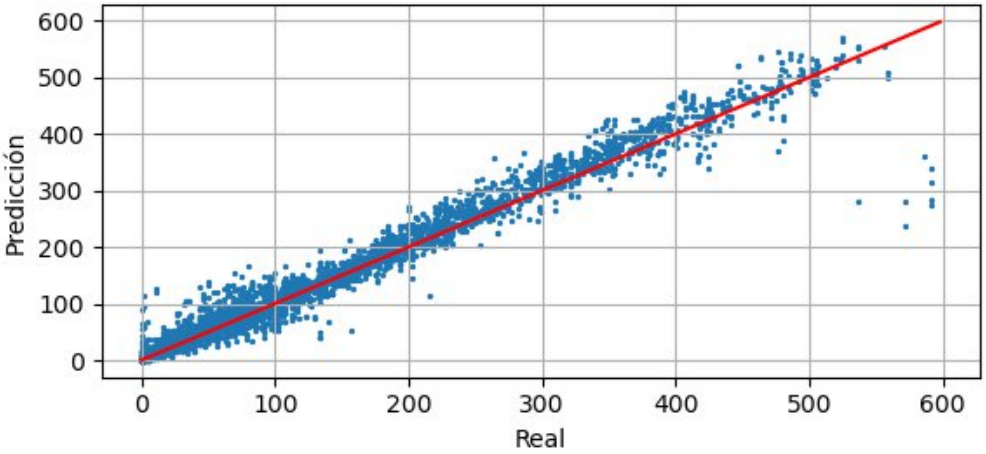


Figura 3.37 Diagrama residual para radiación difusa con RNC y dataset enriquecido

Los resultados muestran que el comportamiento de los diferentes modelos sigue el mismo patrón que con la radiación directa, ofreciendo la RNC mejor rendimiento que el MLP, y a su vez siendo el aumento de datos el que hace que el modelo saque su mayor rendimiento.

Se verá a continuación el resultado de aplicar estos tres modelos al caso real:

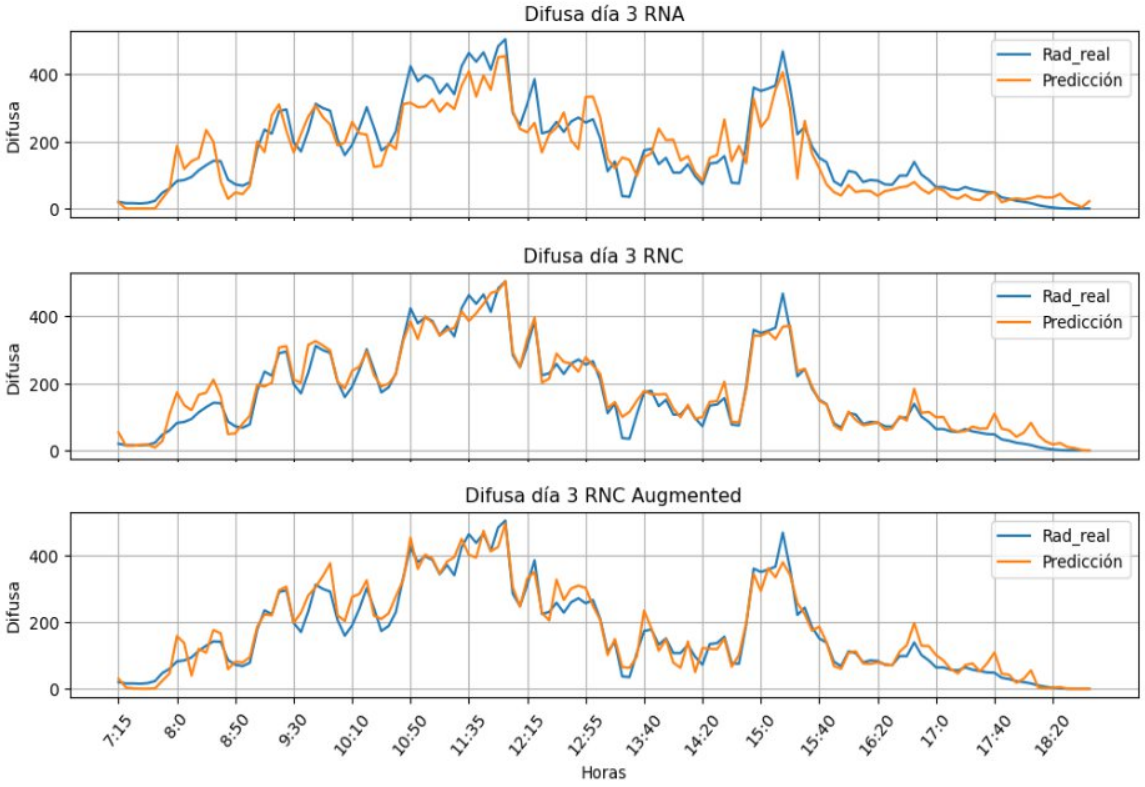


Figura 3.38 Difusa día 03/03/2022 comparación

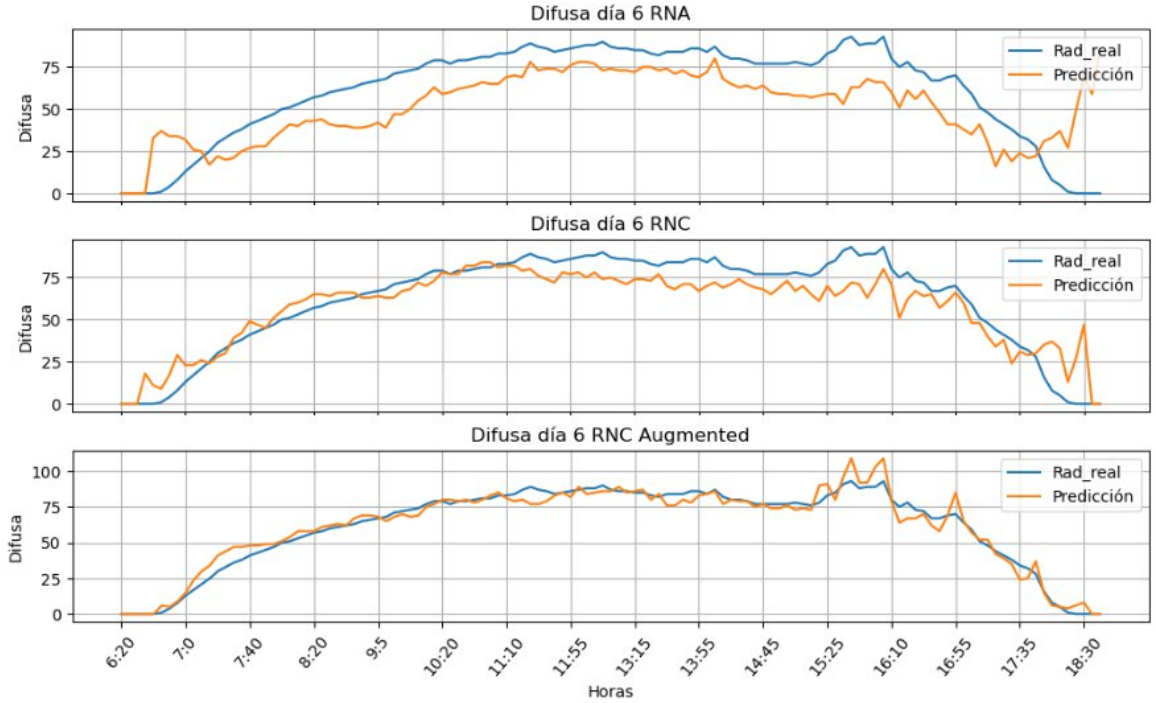


Figura 3.39 Difusa día 06/03/2022 comparación

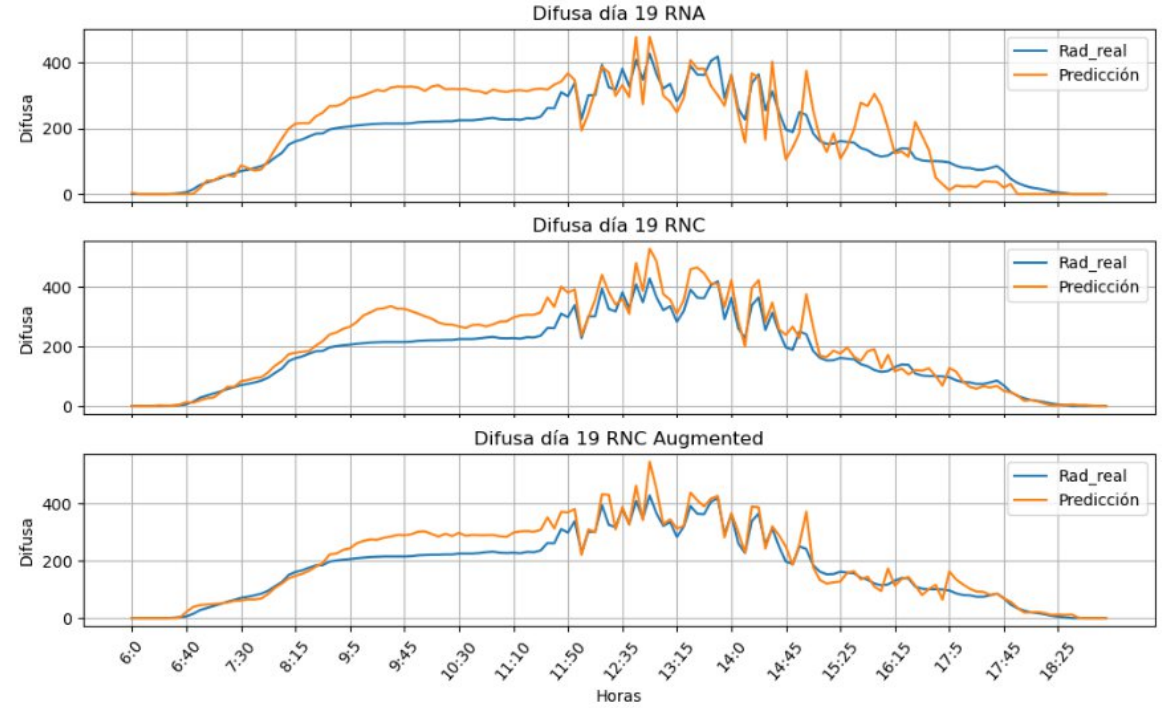


Figura 3.40 Difusa día 19/03/2022 comparación

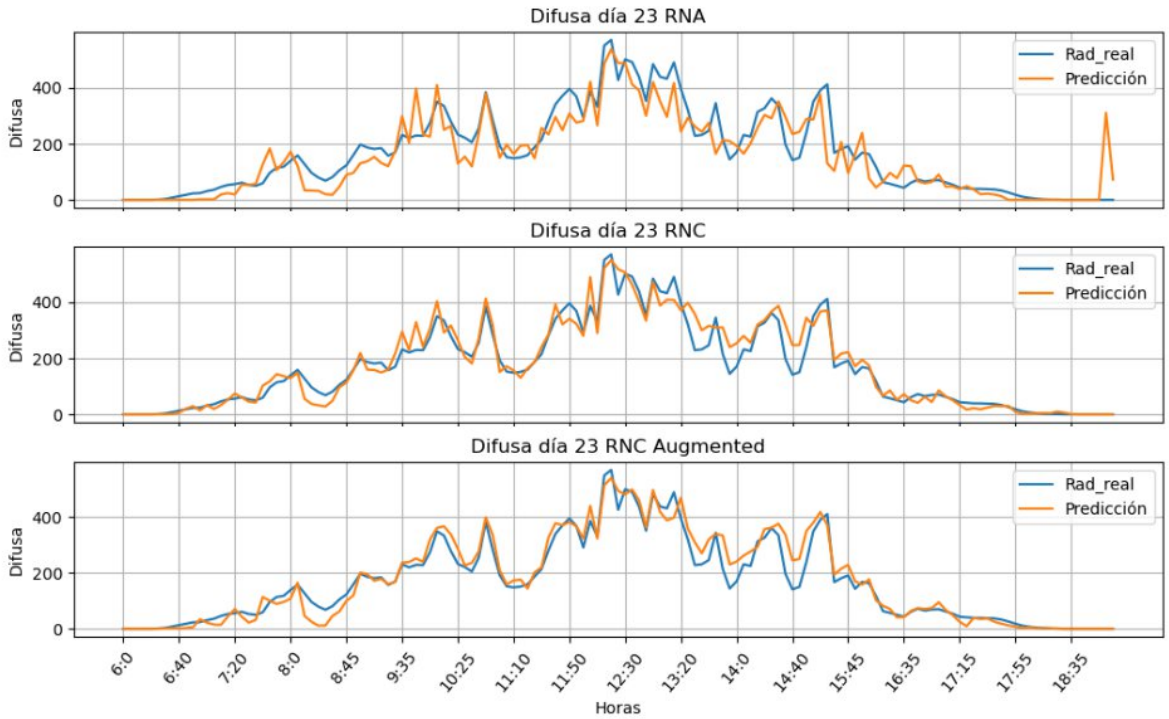


Figura 3.41 Difusa día 23/03/2022 comparación

Queda demostrada la gran eficacia que presenta el modelo de RNC con datos aumentados, el cuál ofrece predicciones que se ajustan con una alta fidelidad al resultado real.

3.5 Resultados

Una vez realizadas todas las pruebas y habiendo analizado sus resultados, se sacan las siguientes conclusiones:

1. El ajuste de los parámetros precisa de la experimentación y el análisis de una extensa batería de pruebas, ya que aún conociendo el marco teórico sobre el que se apoyan, el comportamiento real de cada parámetro se ve influenciado por la combinación de los demás o el tipo de red con el que se trata. Es el ejemplo del optimizador Adam, cuyo funcionamiento en el MLP no ofrece resultados satisfactorios, al contrario que en la RNC, donde se obtiene un buen rendimiento.

2. El tamaño óptimo de las redes neuronales depende de la cantidad de datos que hay a la entrada de estas. Es por eso que en la RNC, donde entran vectorizadas matrices de tamaño 2 x 2, se obtiene un buen rendimiento con únicamente dos capas ocultas, a diferencia del MLP, que presenta un mejor comportamiento con mayores tamaños de red, al entrar en este matrices de 64 x 64.

3. Los modelos son muy sensibles a la presencia de valores atípicos y que no presentan una correlación tan evidente. Esto se hace notar en la diferencia de resultados al tratar con la radiación directa, cuyos valores fluctúan bruscamente influidos por fenómenos meteorológicos, o la radiación difusa, que mantiene una dinámica más suave.

4. Los errores de medición influyen en la capacidad del modelo de buscar relaciones entre datos de entrada y salida. Es el ejemplo de días lluviosos, en los que imágenes similares donde sólo se aprecian nubes grises presentan valores muy distintos de radiación.

3.5.1 Comparación resultados de las pruebas

A continuación se realiza un resumen de los resultados registrados utilizando como objetivo predecir la radiación directa, ya que resulta el caso más desfavorable y se han realizado mayor número de pruebas con este.

En la siguiente tabla comparativa se detalla el rango de valores por los que se mueven los resultados de cada modelo, haciéndose notar la mejora de rendimiento de un modelo convolucional frente al MLP:

Modelo	Rango R^2 (%)	Rango Error Cuadrático $(\frac{W}{m^2})^2$	Rango Error Medio $(\frac{W}{m^2})$
MLP	52 - 83	18.100 - 54.600	57 - 230
RNC	77 - 90	12.500 - 29.200	48 - 80

Tabla 3.9 Rango de valores de los resultados para cada tipo de red

Otro resultado interesante de analizar es el tiempo de ejecución medio de las pruebas con cada modelo, como se puede ver en la siguiente tabla:

Modelo	Tiempo medio de ejecución (s)
MLP	83,9
RNC	112,4

Tabla 3.10 Tiempo de ejecución medio para cada tipo de red

Este resulta ser un 34% mayor en la RNC que en el MLP, debido a que la estructura de la RNC es más compleja y tiene un tamaño superior, ya que partimos de la idea de que una RNC, a *grosso modo*, es un MLP al que se le añade una etapa convolucional en su entrada.

No obstante, esta diferencia de tiempo es asumible teniendo en cuenta la mejora en rendimiento que se obtiene con el uso de una RNC, detallada en la siguiente tabla, donde se realiza una comparativa de los resultados ofrecidos por la mejor configuración encontrada para cada modelo, junto con los resultados obtenidos con el enriquecimiento de datos:

Modelo	Coeficiente R^2 (%)	Error Cuadrático $(\frac{W}{m^2})^2$	Error Medio $(\frac{W}{m^2})$	Tiempo de ejecución (s)
MLP	83	19.168,21	57,05	98,415
MLP Aum	87	13.251,04	51,43	237,83
RNC	90	12.590,78	48,13	118,0815
RNC Aum	95	5.521,27	26,97	825,712

Tabla 3.11 Resultados finales de los mejores modelos obtenidos

Con esto se concluye el análisis del uso de modelos de redes neuronales ante el problema de estimar la radiación solar mediante imágenes del cielo. En el siguiente capítulo se detallarán las conclusiones sacadas a lo largo del proyecto.

4 CONCLUSIONES

Este proyecto ha demostrado que el uso de redes neuronales, en especial lasa redes neuronales convolucionales, es altamente efectivo para predecir la radiación solar a partir de imágenes.

La red neuronal convolucional utilizada en este proyecto ha sido capaz de capturar patrones complejos en los datos, lo que permite una predicción precisa de la radiación solar. Al aprovechar las características propias de las redes neuronales convolucionales, como la detección automática de características relevantes en imágenes, se han podido obtener resultados superiores en comparación con otros enfoques más tradicionales de predicción, como el perceptrón multicapa.

Además de los aspectos positivos mencionados anteriormente, es importante destacar algunas limitaciones y desafíos que han afectado negativamente el rendimiento del proyecto.

4.1 Problemas y limitaciones

En primer lugar, la falta de un dataset más grande ha limitado la capacidad de generalización de la red neuronal. Aunque el enriquecimiento del dataset mejoró los resultados, una base de datos más amplia y diversa habría permitido capturar una mayor variedad de características y patrones, lo que habría mejorado aún más la precisión de las predicciones.

En segundo lugar, los errores en las medidas de radiación solar también pueden haber influenciado negativamente el rendimiento del modelo. Las mediciones inexactas o inconsistentes pueden introducir ruido en el dataset y dificultar la capacidad de la red neuronal para aprender patrones fiables. La calidad y precisión de los datos de entrada son fundamentales para obtener resultados óptimos, por lo que es importante realizar un riguroso control de calidad en las medidas utilizadas.

Además, la capacidad de la GPU ha limitado el tamaño y la complejidad del modelo de red neuronal utilizado. Las redes neuronales, y en especial las convolucionales requieren una gran cantidad de memoria y capacidad de cómputo, especialmente cuando se trabaja con conjuntos de datos extensos o se utilizan arquitecturas de red más profundas. La falta de recursos suficientes en la GPU puede limitar el tamaño del modelo o requerir estrategias de optimización que podrían afectar el rendimiento.

4.2 Trabajo futuro

Como trabajo futuro, en búsqueda de mejorar los resultados y diseñar un modelo con aplicación real, se plantea el estudio de diversas mejoras, como son:

1. La adquisición de un dataset más amplio, que contenga muestras de las cuatro estaciones del año, además de incorporar como entrada datos históricos de radiación solar para capturar patrones meteorológicos de larga duración.

2. Explorar técnicas de transferencia de aprendizaje, basadas en utilizar modelos pre-entrenados en un hardware de mayor capacidad y adaptarlos al problema en cuestión.

3. Utilizar dos redes combinadas: una RNC que clasifique las imágenes en categorías meteorológicas diferentes, por ejemplo catalogar días soleados, nublados y lluviosos, para después introducir sus resultados en un MLP que prediga la radiación entre un rango de valores definido por categoría.

4. Como continuación de estudio, se plantea utilizar un modelo optimizado de red neuronal para predecir radiaciones futuras, utilizando una RNC como predictor principal en conjunto con una Red Neuronal Recurrente, a la cuál se incorporan imágenes de instantes anteriores además de la actual, junto a otras variables, como la radiación típica diaria y la temperatura.

En conclusión, las mejoras mencionadas anteriormente son solo algunas de las muchas oportunidades que existen para mejorar la tecnología de predicción utilizando redes neuronales. Este proyecto ha demostrado que hay un potencial prometedor en el uso de estas técnicas avanzadas de aprendizaje automático para abordar problemas complejos y relevantes en el campo de la energía solar.

Por último, a nivel personal, realizar este proyecto ha sido una experiencia de aprendizaje enriquecedora que me ha generado un gran interés por el conocimiento de este campo de la tecnología, y me siento satisfecho de los resultados obtenidos, a pesar de las limitaciones encontradas.

REFERENCIAS

[1]. Salazar-Peralta, A., Pichardo-S, A., & Pichardo-S, U. (2016). La energía solar, una alternativa para la generación de energía renovable. Revista de Investigación y Desarrollo, 2(5), 11-20.

[2]. Camacho, E. F., & Berenguel, M. (2012). Control of solar energy systems. IFAC proceedings volumes, 45(15), 848-855.

[3]. Kuhn, P., Nouri, B., Wilbert, S., Hanrieder, N., Prahl, C., Ramirez, L., ... & Pitz-Paal, R. (2019). Determination of the optimal camera distance for cloud height measurements with two all-sky imagers. *Solar Energy*, *179*, 74-88.

[4]. Kuhn, P., Wilbert, S., Prahl, C., Schüler, D., Haase, T., Hirsch, T., ... & Pitz-Paal, R. (2017). Shadow camera system for the generation of solar irradiance maps. *Solar Energy*, *157*, 157-170.

[5]. Huertas-Tato, J., Galván, I. M., Aler, R., Rodríguez-Benítez, F. J., & Pozo-Vázquez, D. (2021). Using a multi-view convolutional neural network to monitor solar irradiance. *Neural Computing and Applications*, 1-13.

[6]. Bueis Bellota, C. D. L. (2020). Análisis y corrección de la base de datos meteorológicos del GTER. Aplicación al periodo 2016-2019.

[7]. Caron, M., Bojanowski, P., Joulin, A., & Douze, M. (2018). Deep clustering for unsupervised learning of visual features. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 132-149).

[8]. Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26-38.

[9]. Shao, K., Tang, Z., Zhu, Y., Li, N., & Zhao, D. (2019). A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*.

[10]. Singh, A., Thakur, N., & Sharma, A. (2016, March). A review of supervised machine learning algorithms. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 1310-1315). Ieee.

[11]. Sarker, I. H. (2021). Machine learning: Algorithms, real-world applications and research directions. *SN computer science*, *2*(3), 160.

[12]. Ruiz-Moreno, S., Frejo, J. R. D., & Camacho, E. F. (2021). Model predictive control based on deep learning for solar parabolic-trough plants. *Renewable Energy*, *180*, 193-202.

[13]. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436-444.

[14]. Desai, C. (2020). Comparative analysis of optimizers in deep neural networks. *International Journal of Innovative Science and Research Technology*, *5*(10), 959-962.

[15]. Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

[16]. O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.

[17]. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[18]. Sarvamangala, D. R., & Kulkarni, R. V. (2022). Convolutional neural networks in medical image understanding: a survey. *Evolutionary intelligence*, *15*(1), 1-22.

[19]. Mostavi, M., Chiu, Y. C., Huang, Y., & Chen, Y. (2020). Convolutional neural network models for cancer type prediction based on gene expression. *BMC medical genomics*, *13*, 1-13.

[20]. Kugunavar, S., & Prabhakar, C. J. (2021). Convolutional neural networks for the diagnosis and prognosis of the coronavirus disease pandemic. *Visual computing for industry, biomedicine, and art*, *4*(1), 12.

[21]. Haji, S. H., & Abdulazeez, A. M. (2021). Comparison of optimization techniques based on gradient descent algorithm: A review. *PalArch's Journal of Archaeology of Egypt/Egyptology*, *18*(4), 2715-2743.

[22]. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[23]. Ariff, N. A. M., & Ismail, A. R. (2023, January). Study of adam and adamax optimizers on alexnet architecture for voice biometric authentication system. In *2023 17th International Conference on Ubiquitous Information Management and Communication (IMCOM)* (pp. 1-4). IEEE.

[24]. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.

[25]. Wang, Y. E., Wei, G. Y., & Brooks, D. (2019). Benchmarking TPU, GPU, and CPU platforms for deep learning. *arXiv preprint arXiv:1907.10701*.

ANEXO I. BaseDatosImagenes.py

```
import cv2
import numpy as np
import pytesseract
import math
import pandas as pd
import os
import sqlite3
from time import time


class funciones:
    def __init__(self, img, nombre, df, escala, radio, gray, augmented):
        self.img = img
        self.nombre = nombre
        self.escala = escala
        self.radio = radio
        self.gray = gray
        self.df = df
        self.augmented = augmented
        self.datos = self.fechado()
        self.img_proc = self.procesado()
        self.valores = self.radiaciones()
        self.difusa = self.valores[0]
        self.nip = self.valores[1]
        self.chp1 = self.valores[2]
        self.rad = self.valores[3]
        self.grupo = self.valores[4]
        self.imagenes = self.augmentation()
```

```
    def fechado(self):
        name = list(self.nombre)
        dia = int(name[6]+name[7])
        hora = int(name[8]+name[9])
        minuto = int(name[10]+name[11])
        return [dia, hora, minuto]

    def radiaciones(self):
        difusa, nip, chp1 = self.df[self.df.hora.eq(self.datos[1]) &
self.df.minuto.eq(self.datos[2]) & self.df.segundo.eq(0)].iloc[0,0:3]
        med = int((nip+chp1)/2)
        if med<=5:
            gru=0
        elif med>5 and med<=250:
            gru=1
        elif med>250 and med<=500:
            gru=2
        elif med>500 and med<=750:
            gru=3
        elif med>750:
            gru=4
        return difusa, nip, chp1, med, gru

    def procesado(self):
        #Contornos
        img2 = cv2.cvtColor(self.img, cv2.COLOR_BGR2GRAY)
        ret, th = cv2.threshold(img2, 10, 255, cv2.THRESH_BINARY)
        contornos, jerarquia = cv2.findContours(th,cv2.RETR_TREE,cv2.CHAIN_APPROX_NONE)
        lista_areas = []
        for c in contornos:
            area = cv2.contourArea(c)
            lista_areas.append(area)
        cnt = contornos[lista_areas.index(max(lista_areas))]
        cv2.drawContours(self.img, cnt, -1, (0,0,255), 1)

        #Calcular centroide del contorno circular
```

```
        (rx, ry), radius = cv2.minEnclosingCircle (cnt)
        radio = radius*self.radio
        theta =  np.linspace(0, 2*np.pi, 3*self.img.shape[0])
        x = rx + radio*np.cos(theta)
        y = ry + radio*np.sin(theta)


        img3 = np.zeros((self.img.shape[0],self.img.shape[1], self.img.shape[2]),
          np.uint8)
        for i in range(0,3*self.img.shape[0]):
            img3[int(2*rx-x[i]):int(x[i]),int(y[i])]=self.img[int(2*rx-
            x[i]):int(x[i]),int(y[i])]


        img3=img3[int(2*rx-x[0]):int(x[0]),int(2*rx-x[0]):int(x[0])]
        #Escalado
        if self.escala <= 1:
            ancho = int(img3.shape[1]*self.escala)
            alto = int(img3.shape[0]*self.escala)
            dsize = (ancho, alto)
        else:
            dsize = (self.escala, self.escala)
        img3 = cv2.resize(img3, dsize)
        if self.gray:
            img3 = cv2.cvtColor(img3, cv2.COLOR_BGR2GRAY)
        return img3


    def augmentation(self):
        img1 = cv2.rotate(self.img_proc, cv2.ROTATE_180)
        img2 = cv2.flip(self.img_proc, 0)
        img3 = cv2.rotate(self.img_proc, cv2.ROTATE_90_CLOCKWISE)
        img4 = cv2.flip(self.img_proc, 1)
        img5 = cv2.rotate(self.img_proc, cv2.ROTATE_90_COUNTERCLOCKWISE)
        img6 = cv2.flip(self.img_proc, -1)
        if self.augmented:
            return [self.img_proc, img1, img2, img3, img4, img5, img6]
        else:
            return [self.img_proc]
```

```
############################
#VARIABLES DE CONTROL
############################
#escala = 32         # n<=1: porcentaje / n>1: píxeles
radio = 0.93
gray = False
augmented = False
############################
#DIRECCIONES
############################
path = "C:/Users/Jose/Desktop/"
ruta_imagenes = path+"TFG_documentos/imagenes"
ruta_datos = path+"TFG_documentos/Estación meteorológica/meteo/"
if augmented:
    Base = path+"TFG/BasesDatos/BaseDatosTFG_augmented.sqlite"
else:
    Base = path+"TFG/BasesDatos/BaseDatosTFG.sqlite"
Base_dias = path+"TFG/BasesDatos/Datos_Día.sqlite"

############################
#CREACIÓN BASES DE DATOS
############################
for escala in [16, 32, 64, 128]:
    image_list = []
    data = pd.DataFrame(columns=["nombre", "día", "hora", "minuto", "difusa",
     "NIP", "CHP1", "radiación", "grupo", "imagen"])
    X = []
    y = []
    label = os.listdir(ruta_imagenes)
    j=0
    for carpeta in os.listdir(ruta_imagenes):
        dia = int(carpeta[-2:])
        txt = open(ruta_datos+f"meteo_2022_0{dia+59}.txt", "r")
        df = pd.read_csv(txt, sep="\t",
          header=None).iloc[:,[0,4,10,11]].rename(columns={0:"fecha",4:"difusa",
        10:"nip",11:"chp1"})
```

```
    df=df.join(df["fecha"].str.split(":",expand=True)).iloc[:,1:].rename(
      columns={0:"hora",1:"minuto",2:"segundo"}).astype(int)

    for name in os.listdir(ruta_imagenes+"/"+carpeta):
        if(name.endswith(".jpg")):
            image_list.append([carpeta, name])
            ig = cv2.imread(ruta_imagenes+"/"+carpeta+"/"+name,1)
            fun = funciones(ig, name, df, escala, radio, gray, augmented)
            for i, imagen in enumerate(fun.imagenes):
                indice = len(image_list)-1+i+j
                data.loc[indice,"nombre"]=name
                data.iloc[indice, 1:4] = fun.datos
                data.iloc[indice, 4:7] = [fun.difusa, fun.nip, fun.chp1]
                data.iloc[indice, 7] = fun.rad
                data.iloc[indice, 8] = fun.grupo
                data.iloc[indice, -1] = imagen
            j+=len(fun.imagenes)-1
            print(f"{escala}: {len(image_list)}")
data = data.sort_values(["día","hora", "minuto"], ascending=[True, True, True])
con = sqlite3.connect(Base)
if gray:
    data.to_sql(f'Datos{escala}x{escala}_gris', con, if_exists='replace')
else:
    data.to_sql(f'Datos{escala}x{escala}_color', con, if_exists='replace')
con.close()
```

ANEXO II. RedNeuronal.ipynb

```
import cv2

import numpy as np

import pandas as pd

import os

import sqlite3

from time import time

import time

from io import StringIO

import codecs

import tensorflow as tf

import gc

import keras

from keras import backend as K

from tensorflow import keras

from keras.models import Sequential

from keras.models import model_from_json

from keras.utils import plot_model

from keras.layers import Dense, Normalization, BatchNormalization, Dropout,
Flatten, Conv2D, Input

from keras.layers import RandomRotation, RandomFlip, RandomContrast

from keras.layers.convolutional import MaxPooling2D

from sklearn.model_selection import train_test_split

from keras.utils import to_categorical

from keras.constraints import maxnorm

from sklearn.model_selection import GridSearchCV

from keras.wrappers.scikit_learn import KerasClassifier

from keras.wrappers.scikit_learn import KerasRegressor

from sklearn.multioutput import MultiOutputRegressor

import matplotlib.pyplot as plt

import matplotlib.dates as mdates

from sklearn.preprocessing import StandardScaler

from matplotlib.backends.backend_pdf import PdfPages

import matplotlib.gridspec as gridspec
```

```
def reset_memory(objeto, bool):
    print(tf.config.experimental.get_memory_info('GPU:0'))
    tf.config.experimental.reset_memory_stats("GPU:0")
    K.clear_session()
    gc.collect()
    if bool:
        del objeto.Modelo
        del objeto
    print(tf.config.experimental.get_memory_info('GPU:0'))

def coeff_R2(y_true, y_pred):
    ss_res = K.sum(K.square(y_true - y_pred))
    ss_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return(1 - ss_res/(ss_tot + K.epsilon()))

class Procesado(object):
    def __init__(self, escala, gray, y_norm, rn, augmented):
        if augmented:
            self.Base="C:/Users/Jose/Desktop/TFG/BasesDatos/BaseDatosTFG_augmented.sqlite"
        else:
            self.Base = "C:/Users/Jose/Desktop/TFG/BasesDatos/BaseDatosTFG.sqlite"
        self.BaseRes ="C:/Users/Jose/Desktop/TFG/BasesDatos/ResultadoPruebas.sqlite"
        self.gray = gray
        self.escala = escala
        self.y_normalizado = y_norm
        self.rn = rn[0]
        self.X = []
        self.y = []
        self.X_train = []
        self.y_train = []
        self.X_val = []
        self.y_val = []
        self.X_test = []
        self.y_test = []
        self.media = 0
```

```
        self.stad = 0

    def resha(self, dato):
        if self.gray:
            nun = np.frombuffer(dato, np.uint8).reshape((self.escala,self.escala))
        else:
            nun = np.frombuffer(dato,np.uint8).reshape((self.escala,self.escala,3))
        return nun.astype("float32")

    def procesado_dataset(self):
        con = sqlite3.connect(self.Base)
        if self.gray:
          dataset=pd.read_sql(f'SELECT * FROM Datos{self.escala}x{self.escala}_gris',con)
        else:
          dataset=pd.read_sql(f'SELECT * FROM Datos{self.escala}x{self.escala}_color',con)
        con.close
        self.X = dataset.loc[:,"imagen"]
        #Extrae datos en blob
        self.X = np.array(self.X.apply(self.resha))/255.0
        self.X = np.array([np.array(val) for val in self.X])

        if self.rn=="RNA":
            self.X = self.X.reshape(self.X.shape[0], -1)
        #Convierte blob a vector
        self.y = dataset.loc[:,["difusa","NIP","CHP1"]]
        self.y = self.y*(self.y>=0).values.astype("float32")
        if self.y_normalizado:
            self.y.loc[:,["NIP","CHP1"]] = self.y.loc[:,
                ["NIP","CHP1"]].apply(lambda x: (x-x.mean())/ x.std(),
                axis=0).values.astype("float32")
            self.media = self.y.loc[:,["NIP","CHP1"]].mean()
            self.stad = self.y.loc[:,["NIP","CHP1"]].std()
        self.y = np.array(self.y)
        self.X_train, X_prob, self.y_train, y_prob = train_test_split(self.X,self.y,
          test_size=0.40,random_state=10)
        self.X_val, self.X_test, self.y_val,
```

```
            self.y_test = train_test_split(X_prob,y_prob,
            test_size=0.50,random_state=10)
        print(f"X_train: {self.X_train.shape}")

class RedNeuronal(object):
    def __init__(self, param, neu, datos, tipo, filtros, batch_norm, dropout):
        self.ytip = tipo[1]
        self.resfun = self.tipo_out(datos)
        self.batch_size = param[0]
        self.epochs = param[1]
        self.optimizer = param[2]
        self.loss = param[3]
        self.kernel = param[4]
        self.metrics = [coeff_R2, "mse", "mae"]
        self.activation = "relu"
        self.X_train = datos.X_train
        self.y_train = self.resfun[0]
        self.X_val = datos.X_val
        self.y_val = self.resfun[1]
        self.X_test = datos.X_test
        self.y_test = self.resfun[2]
        self.Modelo = 0
        self.dimin = int(self.X_train.shape[1])
        self.dimout = int(self.y_train.shape[1])
        self.neuronas = [int(self.dimin*n) for n in neu]
        self.entrenamiento = 0
        self.y_pred = []
        self.y_prueba = []
        self.norm = datos.y_normalizado
        self.gray = datos.gray
        self.tipo = tipo[0]
        self.filtros = filtros
        self.batch_norm = batch_norm
        self.dropout = dropout
        self.stan = [datos.stad, datos.media]
```

```
def tipo_out(self, datos):
    if self.ytip == "D":
        a = (0,1)
    elif self.ytip == "R":
        a = (1,3)
    return datos.y_train[:,a[0]:a[1]], datos.y_val[:,a[0]:a[1]], datos.y_test[:,a[0]:a[1]]
def construir(self):
    if self.tipo == "RNA":
        self.build_RNA()
    if self.tipo == "RNC":
        self.build_RNC()
def build_RNC(self):
    a = 3
    if self.gray:
        a = 1

    self.Modelo = Sequential()
    #Capa de entrada
    self.Modelo.add(Input(shape=(self.dimin,self.dimin,a)))

    #Capas de convolución
    for fil in self.filtros:
        self.Modelo.add(Conv2D(filters = fil, kernel_size=(3, 3), activation="relu"))
        if self.batch_norm:
            self.Modelo.add(BatchNormalization())
        self.Modelo.add(MaxPooling2D(pool_size=(2,2)))
    if self.dropout>0:
        self.Modelo.add(Dropout(self.dropout))
    self.Modelo.add(Flatten())
    #Capas ocultas
    for cap in self.neuronas:
        self.Modelo.add(Dense(units = cap, kernel_initializer=self.kernel,
                              activation="relu"))
    if self.dropout>0:
        self.Modelo.add(Dropout(self.dropout))
```

```
        #Capa de salida
        self.Modelo.add(Dense(units=self.dimout, activation=self.activation,
          name="output"))
        self.Modelo.compile(optimizer=self.optimizer,
                            loss=self.loss, metrics=self.metrics)
    def build_RNA(self):
        self.Modelo = Sequential()
        #Capa de entrada
        self.Modelo.add(Input(shape=self.dimin))
        #Capas ocultas
        for cap in self.neuronas:
            self.Modelo.add(Dense(units = cap, kernel_initializer=self.kernel,
                                  activation="relu"))
            if self.batch_norm:
                self.Modelo.add(BatchNormalization())
        if self.dropout>0:
            self.Modelo.add(Dropout(self.dropout))
        #Capa de salida
        self.Modelo.add(Dense(units = self.dimout, activation=self.activation,
          name="output"))

        #Compilación
        self.Modelo.compile(optimizer=self.optimizer,
                    loss=self.loss,
                    metrics=self.metrics)
    def entrenar_RN(self):
        self.entrenamiento = self.Modelo.fit(self.X_train, self.y_train,
          batch_size=self.batch_size,epochs=self.epochs, validation_data=(self.X_val,
          self.y_val), verbose=1)
    def predicción(self):
        y_pred = self.Modelo.predict(self.X_test)
        y_prueba = self.Modelo.predict(self.X_train)
        if self.norm:
            y_pred = pd.DataFrame(y_pred).apply(lambda x:
                x*self.stan[0]+self.stan[1], axis=0).values
            y_prueba = pd.DataFrame(y_prueba).apply(lambda x:
```

```
                x*self.stan[0]+self.stan[1], axis=0).values

        self.y_pred = np.array(y_pred, dtype=int)
        self.y_prueba = np.array(y_prueba, dtype=int)
    def evaluar(self):
        return self.Modelo.evaluate(self.X_test, self.y_test,
                batch_size=self.batch_size)

class resultados(object):
    def __init__(self, red, escala, guardar, t):
        self.BaseRes = "C:/Users/Jose/Desktop/TFG/BasesDatos/ResultadoPruebas.sqlite"
        self.carpeta_graf = "C:/Users/Jose/Desktop/TFG/Gráficas/"
        self.carpeta_model = "C:/Users/Jose/Desktop/TFG/Modelos/"
        self.red = red
        self.tiempo = t
        self.escala = escala
        self.guarda = guardar
        self.resultados = self.calcula_res()
        self.f_perd = self.resultados[1]
        self.valor_res = self.resultados[2]
        self.errornip = self.resultados[3]
        self.errorchp1 = self.resultados[4]
        self.registros = self.registros_fun()
        self.query = self.querys()

    def querys(self):
        conv = ""; neur = ""; cad_crear = ""; cad_inser1 = ""; cad_inser2 = ""
        for i in self.red.filtros:
            conv += f"-{int(i)}"
        for i in self.red.neuronas:
            neur += f"x{int(i)}"
        parametros_cad = [('pixel TEXT', f'"{self.escala}x{self.escala}"'),
                ('gris BOOL', f'{self.red.gray}'),
                ('dataset INTEGER', f'{self.red.X_train.shape[0]}'),
                ('batch_size INTEGER', f'{self.red.batch_size}'),
```

```
        ('epochs INTEGER', f'{self.red.epochs}'),
        #('y_norm BOOL', f'{self.red.norm}'),
        #('batch_norm BOOL', f'{self.red.batch_norm}'),
        ('optimizador TEXT', f'"{self.red.optimizer}"'),
        ('kernel TEXT', f'"{self.red.kernel}"'),
        ('dropout REAL', f'"{self.red.dropout}"')]
convolucion = ('convolucion TEXT', f'"{conv[1:]}"')
res_cad = [('neuronas TEXT', f'"{neur[1:]}"'),
        ('loss TEXT', f'"{self.red.loss}: {self.f_perd}"'),
        #('resultado TEXT', f'"{self.resultados[5]}: {self.valor_res}"')]
        ('coeff_R2 REAL', f'{self.valor_res}')]
errores = ('errorNIP REAL', 'errorCHP1 REAL', 'error_medio REAL',
   f'"{self.errornip}"',f'"{self.errorchp1}"', f'"{self.errornip}"')
tiempo = ('tiempo REAL', f'{self.tiempo}')
for cad in parametros_cad:
    cad_crear += ", " + cad[0]
    cad_inser1 += ", " + cad[0].split(" ")[0]
    cad_inser2 += ", " + cad[1]
if self.red.tipo == "RNC":
    cad_crear += ", " + convolucion[0]
    cad_inser1 += ", " + convolucion[0].split(" ")[0]
    cad_inser2 += ", " + convolucion[1]
for cad in res_cad:
    cad_crear += ", " + cad[0]
    cad_inser1 += ", " + cad[0].split(" ")[0]
    cad_inser2 += ", " + cad[1]
if self.red.ytip == "R":
    cad_crear += ", " + errores[0] + ", " + errores[1]
    cad_inser1 += ", " + errores[0].split(" ")[0] + ", " + errores[1].split(" ")[0]
    cad_inser2 += ", " + errores[3] + ", " + errores[4]
elif self.red.ytip == "D":
    cad_crear += ", " + errores[2]
    cad_inser1 += ", " + errores[2].split(" ")[0]
    cad_inser2 += ", " + errores[5]
cad_crear += ", " + tiempo[0]
```

```
        cad_inser1 += ", " + tiempo[0].split(" ")[0]
        cad_inser2 += ", " + tiempo[1]
        q_crear = f'CREATE TABLE IF NOT EXISTS "{self.red.tipo}_{self.red.ytip}"
          ({cad_crear[2:]})'
        q_insertar = f'INSERT INTO "{self.red.tipo}_{self.red.ytip}" ({cad_inser1[2:]})
          VALUES({cad_inser2[2:]})'
        return q_crear, q_insertar
    def registros_fun(self):
        con = sqlite3.connect(self.BaseRes)
        cur = con.cursor()
        cur.execute(self.querys()[0])
        cur.execute(f"SELECT count(*) FROM {self.red.tipo}_{self.red.ytip}")
        con.commit()
        return cur.fetchall()[0][0]

    def calcula_res(self):
        compara = list(map(lambda x,y: abs(x-y), self.red.y_test, self.red.y_pred))
        error_medio = sum(compara)/len(compara)
        enip = float(error_medio[0])
        if self.red.ytip != "D":
            echp1 = float(error_medio[1])
        else:
            echp1 = 0
        f_per_t = self.red.entrenamiento.history["loss"][-1]
        eval = self.red.evaluar()
        med = self.red.metrics[0]
        if self.red.ytip != "C":
            med = med.__name__
        re = 2
        if self.red.norm:
            re = 5
        return round(f_per_t, re), round(eval[0], re), round(eval[1], re),
          round(enip, re),round(echp1, re), med
```

```
def grafica(self):
    print(f"Loss_train: {self.resultados[0]}\nError NIP: {self.errornip},
      Error CHP1: {self.errorchp1}")
    fig, ax = plt.subplots(2,2,figsize=(12, 6), sharex=False)
    ax[0,0].plot(self.red.entrenamiento.history["loss"][4:], label="Loss_train")
    ax[0,0].plot(self.red.entrenamiento.history["val_loss"][4:], label="Loss_val")
    ax[0,0].set_title("Pérdida")
    ax[0,0].set_ylabel("Loss")
    ax[0,0].set_xlabel("Epoch")
    ax[0,0].legend(["Train", "Val"], loc="upper right")
    ax[0,0].grid()

    ax[0,1].plot(self.red.entrenamiento.history[f"{self.resultados[5]}"][4:],
      label="Accuracy_train")
    ax[0,1].plot(self.red.entrenamiento.history[f"val_{self.resultados[5]}"][4:],
      label="Accuracy_val")
    ax[0,1].set_title(f"Model {self.resultados[5]}")
    ax[0,1].set_ylabel(f"{self.resultados[5]}")
    ax[0,1].set_xlabel("Epoch")
    ax[0,1].legend(["Train", "Val"], loc="lower right")
    ax[0,1].grid()

    ax[1,0].scatter(self.red.y_test, self.red.y_pred, s=2)
    ax[1,0].set_title("Relación y_test/y_pred")
    ax[1,0].set_xlabel("Real")
    ax[1,0].set_ylabel("Predicción")
    ax[1,0].plot([min(self.red.y_test[:,0])*0.99, max(self.red.y_test[:,0])*1.01],
      [min(self.red.y_test[:,0])*0.99,max(self.red.y_test[:,0])*1.01], color="red")
    ax[1,0].grid()

    ax[1,1].scatter(self.red.y_train, self.red.y_prueba, s=2)
    ax[1,1].set_title("Relación y_train/y_prueba")
    ax[1,1].set_xlabel("Real")
    ax[1,1].set_ylabel("Predicción")
    ax[1,1].plot([min(self.red.y_train[:,0])*0.99, max(self.red.y_train[:,0])*1.01],
      [min(self.red.y_train[:,0])*0.99,max(self.red.y_train[:,0])*1.01],color="red")
    ax[1,1].grid()
```

```
        if self.guarda:
            plt.savefig(self.carpeta_graf +
                f"Prueba_{self.red.tipo}_{self.red.ytip}_{self.registros+1}.jpg",
              bbox_inches='tight')
            self.crear_tabla()
            self.guardar_modelo()
        plt.tight_layout()
        plt.show()
    def crear_tabla(self):
        con = sqlite3.connect(self.BaseRes)
        cur = con.cursor()
        cur.execute(self.query[0])
        cur.execute(self.query[1])
        con.commit()
    def guardar_modelo(self):
        model_json = self.red.Modelo.to_json()
        with
     open(self.carpeta_model+f"model_{self.red.tipo}_{self.red.ytip}_{self.registros+1}.json",
          "w") as json_file:
            json_file.write(model_json)
        pesos = self.red.Modelo.get_weights()
        pesos = np.asanyarray(pesos, dtype=object)
        np.save(self.carpeta_model+f"pesos_{self.red.tipo}_{self.red.ytip}_{self.registros+1}",
          pesos)


class evolucion_temporal(object):
    def __init__(self, dia, rn, registro):
        self.ruta_imagenes = "C:/Users/Jose/Desktop/TFG_documentos/imagenes_test"
        self.Base = "C:/Users/Jose/Desktop/TFG/BasesDatos/BaseDatos_Pruebas.sqlite"
        self.BaseRes = "C:/Users/Jose/Desktop/TFG/BasesDatos/ResultadoPruebas.sqlite"
        self.carpeta_model = "C:/Users/Jose/Desktop/TFG/Modelos/"
        self.dia = dia
        self.Modelo = 0
        self.tipo = rn[0]
        self.ytip = rn[1]
        self.dataset = []
        self.registro = self.registros_fun(registro)
```

```
        self.parametros = self.extracc_datos()
        self.escala = self.parametros[0]
        self.optimizer = self.parametros[1]
        self.loss = self.parametros[2]
        self.datos = self.procesado()
        self.X = self.datos[0]
        self.y = self.datos[1]
        self.x_ax = self.datos[2]
        self.y_pred = self.prediccion()

    def resha(self, dato):
        nun = np.frombuffer(dato, np.uint8).reshape((self.escala,self.escala,3))
        return nun.astype("float32")
    def procesado(self):
        con = sqlite3.connect(self.Base)
        dataset = pd.read_sql(f'SELECT * FROM Datos{self.escala}x{self.escala}_color WHERE
          (día={self.dia})', con)
        con.close
        self.dataset = dataset.iloc[:,:-1]
        X = dataset.loc[:,"imagen"]
        hora = dataset.loc[:,"hora"]
        minuto = dataset.loc[:, "minuto"]
        x_ax = hora.astype(str)+":"+minuto.astype(str)
        #Extrae datos en blob
        X = np.array(X.apply(self.resha))/255.0
        X = np.array([np.array(val) for val in X])
        if self.tipo=="RNA":
            X = X.reshape(X.shape[0], -1)
        if self.ytip=="D":
            y = dataset.loc[:,"difusa"]
        else:
            y = dataset.loc[:,["NIP","CHP1"]]
        y = y*(y>=0).values.astype("float32")
        return X, np.array(y), x_ax
```

```
def registros_fun(self, registro):
    reg = registro
    if registro==-1:
        con = sqlite3.connect(self.BaseRes)
        cur = con.cursor()
        cur.execute(f"SELECT count(*) FROM {self.tipo}_{self.ytip}")
        con.commit()
        reg = cur.fetchall()[0][0]
    return reg
def extracc_datos(self):
    con = sqlite3.connect(self.BaseRes)
    cur = con.cursor()
    cur.execute(f"SELECT pixel, optimizador,
       loss FROM {self.tipo}_{self.ytip} WHERE rowid = {self.registro}")
    con.commit()
    res = cur.fetchall()[0]
    return int(res[0].split("x")[0]), res[1], res[2].split(" ")[1][:-1]


def prediccion(self):
    with open(self.carpeta_model+f"model_{self.tipo}_{self.ytip}_{self.registro}.json",
      'r') as json_file:
        model_json = json_file.read()
    self.Modelo = model_from_json(model_json)
    pesos =np.load(self.carpeta_model+f"pesos_{self.tipo}_{self.ytip}_{self.registro}.npy",
      allow_pickle=True)
    self.Modelo.set_weights(pesos)
    self.Modelo.compile(loss=self.loss, optimizer=self.optimizer,
      metrics=["mse"],run_eagerly=True)
    y_pred = self.Modelo.predict(self.X)
    y_pred = np.array(y_pred, dtype=int)
    return y_pred


def grafica(self, mostrar):
    x_ax = self.x_ax
    fig = plt.figure(figsize=(13,7), constrained_layout = True, dpi=100)
    GridSpec = gridspec.GridSpec(ncols=2, nrows=2, figure= fig, hspace=None)
    subfig1 = fig.add_subfigure(GridSpec[0,:],)
```

```
if self.ytip!="D":
    ax1 = subfig1.subplots(1,2, sharex=False)
    ax1[0].plot(x_ax, self.y[:,0], label="NIP_real")
    ax1[0].plot(x_ax, self.y_pred[:,0], label="NIP_pred")
    ax1[0].set_title(f"NIP día {self.dia}")
    ax1[0].set_ylabel("NIP")
    ax1[0].set_xlabel("Horas")
    ax1[0].legend(["NIP_real", "Predicción"], loc="upper right")
    ax1[0].set_xticks(x_ax[0:len(x_ax):8])
    ax1[0].set_xticklabels(x_ax[0:len(x_ax):8], rotation = 50)
    ax1[0].grid()

    ax1[1].plot(x_ax, self.y[:,1], label="CHP1")
    ax1[1].plot(x_ax, self.y_pred[:,1], label="CHP1_pred")
    ax1[1].set_title(f"CHP1 día {self.dia}")
    ax1[1].set_ylabel("CHP1")
    ax1[1].set_xlabel("Horas")
    ax1[1].legend(["CHP1_real", "Predicción"], loc="upper right")
    ax1[1].set_xticks(x_ax[0:len(x_ax):8])
    ax1[1].set_xticklabels(x_ax[0:len(x_ax):8], rotation = 50)
    ax1[1].grid()
else:
    ax1 = subfig1.subplots(1,1, sharex=False)
    ax1.plot(x_ax, self.y[:], label="Rad_real")
    ax1.plot(x_ax, self.y_pred[:], label="Rad_pred")
    ax1.set_title(f"Difusa día {self.dia}")
    ax1.set_ylabel("Difusa")
    ax1.set_xlabel("Horas")
    ax1.legend(["Rad_real", "Predicción"], loc="upper right")
    ax1.set_xticks(x_ax[0:len(x_ax):8])
    ax1.set_xticklabels(x_ax[0:len(x_ax):8], rotation = 50)
    ax1.grid()
lista, l = self.enumera_imagen()
subfig2 = fig.add_subfigure(GridSpec[1,:])
ax2 = subfig2.subplots(1,l)
```

```
    for i, ind in enumerate(range(0, len(lista), 8)):
        img = plt.imread(self.ruta_imagenes + f"/202203"+f"0{self.dia}"[-2:] +
            "/"+lista[ind],1)
        ax2[i].imshow(img)
        ax2[i].axis("off")
        ax2[i].set_title(f"{x_ax[ind]}")
    if mostrar:
        plt.show()
    return fig
def enumera_imagen(self):
    lista = []
    for imagen in os.listdir(self.ruta_imagenes + f"/202203"+f"0{self.dia}"[-2:]):
        lista.append(imagen)
    for i, ind in enumerate(range(0, len(lista), 8)):
        None
    return lista, i+1
def escoger_modelo(self, parametro):
    con = sqlite3.connect(self.BaseRes)
    cur = con.cursor()
    if parametro=="resultado":
        op = "MAX"
    else:
        op = "MIN"
    cur.execute(f"SELECT ROWID, {parametro} FROM {self.tipo}_{self.ytip} WHERE
      {parametro} = (SELECT {op}({parametro}) FROM {self.tipo}_{self.ytip})")
    con.commit()
    reg = cur.fetchall()
    print(reg)
```

```
##########################################################################################

#PARAMETROS DE PROCESADO DE DATOS
escala = 64
gray = False
y_normalizado = False
augmented = True
#PARAMETROS DE COMPILACIÓN
rn = ("RNC", "D")  # R: Regresión, D: Difusa
batch_normalization = False
batch_size = 128
epochs = 120
optimizer = "adam"
kernel = "glorot_uniform"
loss = "mse"
dropout = 0.6
#CAPAS DENSAS
mul = 8
neuronas = [4,2,1]
neuronas = [mul*i for i in neuronas]
#CAPAS DE CONVOLUCION
filtros = [64, 128, 256, 512]
data = Procesado(escala, gray, y_normalizado, rn, augmented)
data.procesado_dataset()
red = RedNeuronal((batch_size,epochs,optimizer,loss,kernel), neuronas, data, rn,
filtros, batch_normalization, dropout)
red.construir()
modelo = red.Modelo
print(f"Neuronas: {red.neuronas}")
modelo.summary()
```

```
#############################################################################################

t1 = time.time()
entrenamiento = red.entrenar_RN()
t2 = time.time()
t = round(t2 - t1, 4)
red.predicción()
guardar_datos = True
res = resultados(red, escala, guardar_datos, t)
res.grafica()
reset_memory(red, True)

#############################################################################################

modelo = 3
rnt = ("RNC", "D")
pp = PdfPages(f"modelo_{rnt[0]}_{rnt[1]}_{modelo}.pdf")
for dia in [3, 6, 19, 23]:
    evolucion = evolucion_temporal(dia, rnt, modelo)
    fig = evolucion.grafica(False)
    pp.savefig(fig)
    reset_memory(evolucion, True)
pp.close()
```