Detección automatizada de meteoros mediante Deep Learning

David Regordosa Avellana
Máster en Inteligencia de Negocio y Big Data Analytics
Data Analytics

Consultor: **Jordi Gironés Roig**

Entrega TFM: Julio del 2021

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Detección automatizada de meteoros mediante Deep Learning*
Nombre del autor:	*David Regordosa Avellana*
Nombre del consultor/a:	*Jordi Gironés Roig*
Nombre del PRA:	Albert Solé Ribalta
Fecha de entrega (mm/aaaa):	*07/2021*
Titulación:	*Máster en Inteligencia de Negocio y Big Data Analytics*
Área del Trabajo Final:	*Data Analytics*
Idioma del trabajo:	*Español*
Palabras clave	*Meteoros, clasificación, fast.ai, CNN*

Resumen del Trabajo

La finalidad de este trabajo es la de desarrollar un *pipeline* automatizado que sea capaz de detectar meteoros mediante el procesado de imágenes y usando técnicas de *Deep Learning*, como las redes neuronales convolucionales (CNN), testeando previamente distintas técnicas de procesado de imagen, *data augmentation* y topologías de red existentes.

En la actualidad algunos observatorios astronómicos están usando sistemas de captación de imágenes nocturnas del cielo con la finalidad de detectar cualquier anomalía que se produzca. Estos sistemas, dependiendo de su complejidad, pueden realizar procesado automatizado casi en tiempo real de las imágenes para generar alertas en caso de detección de meteoros. En muchas otras ocasiones, este procesado automatizado no se realiza, y se requiere que un operador revise las imágenes por la mañana y en caso de detección proceda a generar el aviso correspondiente.

El trabajo realizado tiene el reto de proporcionar una herramienta *open source* para clasificar automáticamente las imágenes que contengan meteoros y así facilitar el trabajo del operador.

Otros retos del trabajo realizado consistían en no disponer de un dataset histórico y etiquetado, en disponer de un porcentaje muy bajo de imágenes de meteoros respecto a no-meteoros y en la dificultad de detectar a simple vista meteoros tenues presentes en imágenes. En base a estos retos, se han analizado y testeado distintas metodologías de pre-procesado de imágenes y técnicas de *data augmentation* para generar un modelo de clasificación que proporcione un buen rendimiento.

Abstract

The goal of this work is to develop an automated *pipeline* capable of detecting meteors through image processing and using Deep Learning techniques, such as convolutional neural networks (CNN), previously testing different image processing techniques, data augmentation procedures, and different network topologies.

Some astronomical observatories are currently using systems to capture night images of the sky to detect anomalies. These systems, depending on their complexity, can perform almost real-time automated processes of the images to generate alerts in case of meteor detection. In many other cases, this automated process is not performed, and an operator is required to review the images the next day, and in case of detection, proceed to generate the corresponding warning.

The work done has the challenge of providing an *open source* software to automatically classify images containing meteors and make the operator's job easier.

Other challenges of the work done are not having a historical and labeled dataset, having a very low percentage of meteor images compared to non-meteor, and the difficulty of low bright meteor detection in some images. Based on these challenges, different image pre-processing methodologies and data augmentation techniques have been analyzed and tested to generate a classification model that provides good performance.

Índice

Lista de figuras

Lista de tablas

1. Introducción

1.1 Contexto y justificación del Trabajo

El observatorio astronómico de Pujalt [1] realiza diariamente un análisis del cielo nocturno para la detección de eventos relevantes. Uno de los eventos que se monitorizan son los meteoros.

Los meteoros son fenómenos luminosos que se producen cuando un meteoroide (partículas de polvo y hielo que se encuentran en el espacio) atraviesa nuestra atmósfera, y por efecto de la fricción, se quema en las capas altas de esta.
Para determinar el brillo producido por un meteoro, y en general para cualquier cuerpo celeste, se usa el concepto de magnitud. Un cuerpo 100 veces más brillante que otro tiene 5 unidades menos de magnitud, por lo tanto, a menor magnitud mayor brillo.
Un cuerpo de 8 milímetros de tamaño genera un efecto óptico de magnitud +2

Existe un grupo de meteoros especialmente luminosos, con magnitud inferior a -4, llamados bólidos. Los bólidos son producidos por meteoroides con una masa desde un gramo hasta miles de toneladas.
El estudio de los meteoros y especialmente de los bólidos, es extraordinariamente valioso puesto que, una vez detectada su trayectoria y velocidad, se puede inferir la órbita que seguían en el sistema solar y determinar el origen de su procedencia. Además, trazando correctamente su trayectoria de descenso se pueden llegar a recuperar fragmentos del cuerpo que hayan impactado en la tierra, lo que llamamos meteoritos.

Para determinar con exactitud su trayectoria es requerido capturar imágenes del meteoro desde distintos puntos geográficos y centralizar estas capturas para su análisis. Esto es precisamente lo que entidades como la Red de Investigación Sobre Bólidos y Meteoritos (SPMN)[2] [11] realizan.
A nivel internacional, existen otras entidades que, de igual manera que el SPMN, realizan estas tareas de centralización de eventos para su análisis. El International Meteor Organitzation (IMO)[3] [13], la Southwestern Europe Meteor Network (SWEMN[4]) o la Cameras for Allsky Meteor Surveillance (CAMS) [5], son un ejemplo.

En la actualidad el Observatorio Astronómico de Pujalt, con su proyecto Alpha Sky[6], se dedica a capturar imágenes del cielo nocturno y posteriormente de

[1] https://observatoridepujalt.cat/
[2] http://www.spmn.uji.es/
[3] https://www.imo.net/
[4] http://www.swemn.org/
[5] http://cams.seti.org/
[6] https://observatoridepujalt.cat/alphasky/

forma manual analizarlas para detectar meteoros. En caso de detectar algún bólido, se reporta al SPMN para su análisis y detección de trayectoria.
La captación de imágenes que realiza el observatorio de Pujalt, se realiza mediante un equipo low-cost, formado por una Raspberry PI 4, una cámara CCD, un objetivo de 180º de visión y un software específico para la captación de imágenes desde un sistema operativo Raspbian (distribución de Linux para Raspberry).
El software usado, de licencia MIT, se llama AllSky[7], y dispone de un repositorio en Github con el código fuente. AllSky se ejecuta durante la noche y realiza capturas del cielo de 30 segundos de exposición, y genera un conjunto de imágenes JPG de 1304 x 976.

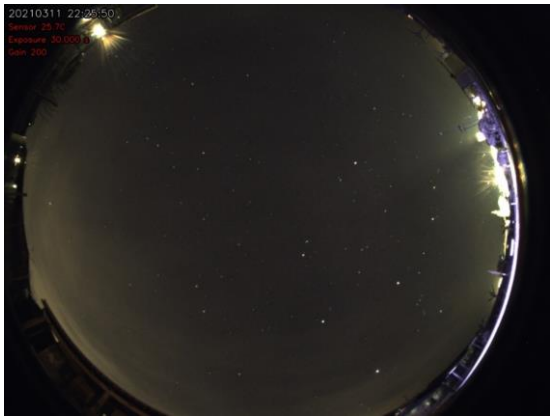


Figura 1 : Imagen captada por el software AllSky
Fuente: Observatori de Pujalt

Figura 2 : Bólido detectado
Fuente: Observatori de Pujalt

Mediante una exposición de 30 segundos se incrementan las posibilidades de detección de meteoros de alta magnitud que dejarían un rastro prácticamente imperceptible. Por otro lado, aplicar dicha exposición descarta el análisis en tiempo real, puesto que se generan dos imágenes por minuto.

Por la mañana, desde el observatorio, se realiza una revisión de las imágenes generadas durante la noche, que dependiendo de la época del año pueden ser

[7] https://github.com/thomasjacquin/allsky

entre unas 900 y 1.400. En caso de detectar algún bólido, este se reporta a la SPMN.
Es probable que algunos bólidos pasen desapercibidos para el operador, con lo que no se dispone de un etiquetado fiable de casos positivos y negativos.
No se realiza la detección de meteoros de bajo brillo al ser una tarea compleja para ser realizada manualmente por un operador.
El paquete de imágenes generadas cada noche, se almacena en la Raspberry solo durante 10 días, por lo tanto, no existe un histórico de imágenes y detecciones. Durante la ejecución de este proyecto, se realizó un almacenaje de imágenes generadas, consiguiendo acumular aproximadamente 57.000.

El objetivo principal de este trabajo es el de proporcionar al proyecto Alpha Sky, de una herramienta de análisis de las imágenes capturadas por el software AllSky, para la detección automatizada de meteoros y específicamente bólidos.
Dicha detección se debe realizar desde el mismo observatorio donde esté ubicada la Raspberry Pi, mediante el procesado de cada una de las imágenes que se van generando y usando un modelo de clasificación de Deep Learning.

En este caso, el tiempo de procesado de cada imagen no es crítico, puesto que se generan cada 30 segundos, pero si es importante maximizar el numero de detecciones realizadas para así disponer de más casos positivos que puedan alimentar al modelo.

Al finalizar el análisis de cada una de las imágenes que se van generando, el modelo deberá tomar la decisión de si la imagen contiene o no un meteoro y en caso afirmativo almacenar la imagen en la nube junto a una serie de metadatos informativos.
El operador podrá evaluar los casos detectados y cruzar la información con otras detecciones realizadas por otros observatorios que usen el mismo sistema de detección.
Adicionalmente podrá emitir una notificación al SPMN si lo considera adecuado.
En futuras iteraciones del producto, y en base a la capacidad de detección, se evaluará si emitir la notificación al SPMN de forma automática.

Los positivos detectados, ya sean verdaderos positivos o falsos positivos estarán disponibles para el reentreno del modelo.

1.2 Objetivos del Trabajo

Los objetivos a alto nivel que pretende cumplir el trabajo son:

- Evaluación de tecnologías existentes para la clasificación automática de imágenes y elección del *stack* tecnológico a usar en este trabajo
- Generar un modelo de Deep Learning mediante Redes Neuronales Convolucionales (CNN) para la clasificación automática de imágenes en dos grupos: Meteoro y No-Meteoro. Para la realización de este objetivo se deberá tomar en consideración la siguiente información:
    - El dataset del que se dispone en la actualidad se compone de 57.000 imágenes, con un tamaño total de 42GB. Las imágenes disponibles tienen una resolución de 1304 x 976 pixeles, con lo que es necesario realizar operaciones sobre ellas previo al proceso de *training* del modelo.
    - No se dispone de un etiquetado de las imágenes, con lo que se debe implementar un sistema de ayuda al etiquetado. Algunas imágenes contienen meteoros muy tenues que no se detectan a simple vista. Se deberán evaluar distintos mecanismos para facilitar el etiquetado del dataset, realizando operaciones previas sobre las imágenes del dataset:
        - *Resizing* de la imagen
        - Conversión a escala de gris
        - Restado de imágenes consecutivas para la eliminación del fondo de imagen (elementos estáticos). Ver imagen a continuación en la que se restan las dos primeras imágenes para generar una tercera en la que desaparecen los elementos estáticos. Ver Figura [3]
        - Data augmentation de los casos positivos mediante: *zoom, crop, rotate, brightness, contrast, dihedral, flip, dilate, erode*, etc.
    - Será necesario evaluar si las mismas transformaciones realizadas a las imágenes, para favorecer el etiquetado, se aplican también al dataset para realizar el proceso de *training*, mejorando así los resultados.
    - Existen muy pocos casos positivos detectados. El Observatori de Pujalt, solo aporta 18 imágenes de meteoros detectados, con lo que será necesario realizar una identificación manual de casos positivos no detectados y un proceso de *data augmentation* para enriquecer el dataset.
    - Será requerido crear un entorno de trabajo desde el que se pueda realizar el entrenamiento de modelos de clasificación de imágenes en condiciones óptimas
    - Será requerido evaluar distintas topologías de red para encontrar la que proporcione mejores resultados.
        - Usar técnicas de *Transfer Learning* con redes pre-entrenadas
        - En caso de no obtener buenos resultados con redes pre-entrenadas, evaluar la posibilidad de Usar Redes Neuronales Convolucionales (CNN) entrenadas ad-hoc para este trabajo

- Generar un software, de análisis de las imágenes captadas por el sistema. Este software debe implementar las siguientes funcionalidades:
    - Descarga del modelo de Deep Learning almacenado en la nube, para la ejecución del proceso de clasificación de imágenes
    - Procesado de ficheros grabados en disco por un software externo de captación de imágenes. En nuestro caso específico el software es AllSky.
    - Preparación previa de la imagen antes de la ejecución del modelo de clasificación:
        - Aplicar mismas transformaciones realizadas para el proceso de *training*.
    - Ejecución del modelo de Deep Learning para la clasificación automática de imágenes en dos grupos: Meteoro y No-Meteoro
    - Envío a un servidor centralizado en la nube (Amazon Web Services) de las imágenes con los casos positivos detectados y con metadatos que permitan identificar:
        - Observatorio desde el que se ha identificado
        - Fecha/Hora de la detección
        - *Scoring* obtenido
        - Etc.

Figura 3 : Restado de imágenes para eliminación de elementos estáticos
Fuente: Observatori de Pujalt

Todo el software desarrollado, ya sea para preparar el dataset, entrenar el modelo y realizar la inferencia, se puede encontrar en el repositorio Github creado para el trabajo:

https://github.com/pisukeman/guAIta

Este repositorio contiene:

- Software Python desarrollado:
    - guAIta.py: Cliente de inferencia. Detección de meteoros mediante análisis de las imágenes generadas por el sistema de detección. Permite detección en *batch* o en *real-time*
    - guAItaConfig.py: Fichero de configuración del cliente de inferencia
    - dataPreparation.py: Conjunto de funciones para realizar el pre-procesado de imágenes y preparación de las distintas versiones de dataset de training y test
    - guAIta_conda_env.yml: Fichero yaml con todas las librerías necesarias para instalar el cliente de inferencia.
- Carpeta /Training_and_Inference_Examples

    - Jupyter Notebooks en los que se muestra el proceso de training e inferencia
    - HTMLs y PDFs de ejemplos con el proceso de entrenamiento de la ResNet34, ResNet18 e inferencia sobre imágenes
- Carpeta /AWS_Lambda_Functions
    - Funciones Lambda de Amazon Web Services para la implementación de una API de recepción de casos positivos detectados y generación de herramientas para el operador del sistema

2. Estado del Arte

En este apartado se repasará el estado actual de la tecnología de *Image Clasification.* Se evaluarán las herramientas que se están usando en proyectos de clasificación de imágenes, así como trabajos de investigación al respecto. Más específicamente se revisarán trabajos de investigación sobre la detección automatizada de meteoros, para evaluar que modelos han dado mejor resultado y porqué.

En primer lugar, es importante tomar conciencia de la diferencia entre detección y clasificación de imágenes. El primer caso, la detección, tiene el objetivo de reconocer patrones dentro de una imagen y clasificarlos. Por lo tanto, engloba tanto clasificación como localización.
La clasificación de imágenes consiste en tratar la imagen como un todo y asignarle una etiqueta específica.

El campo científico del reconocimiento y clasificación de imágenes ha sufrido cambios drásticos en los últimos años. La aparición de plataformas como Google Colab o Kaggle han permitido experimentar mediante machine Learning de forma rápida y barata.
Unos años atrás, el reconocimiento o clasificación de imágenes requerían de cantidades ingentes de datos y grandes desarrollos realizados por grandes equipos de científicos con un nivel de conocimiento muy especifico.
En la actualidad y gracias al desarrollo tecnológico de este campo científico, cualquiera puede implementar softwares de reconocimiento y clasificación.
Cabe destacar que este auge tecnológico alrededor de la clasificación/reconocimiento de imágenes, viene propiciado por lo transversal de la tecnología y su uso en muchos campos.

Focalizando en la clasificación de imágenes, que es la tecnología que se usará en este trabajo, cabe destacar que existen muchos algoritmos para la clasificación de imágenes, supervisados y no supervisados, desde los más tradicionales como *Decision Trees, SVM, k-NN*, SIFT, HOG, redes neuronales, etc. hasta las CNN que se están imponiendo claramente en el campo de reconocimiento y clasificación de imágenes.
Un *benchmark* interesante para evaluar las distintas tecnologías en clasificación de imágenes es el *ImageNet Large Scale Visual Recognition Challenge*[8] (ILSVRC), challenge en el que, desde el 2012, los mejores resultados en clasificación de imágenes se obtienen mediante el uso de distintas topologías de CNNs.

Las CNN o Redes Neuronales Convolucionales, son un tipo de multi-layer Neural Network diseñadas para reconocer patrones visuales directamente de pixeles de imágenes, con un pre-procesado mínimo. Las CNN se engloban dentro del ámbito científico conocido como Deep Learning.
Se detalla a continuación información sobre las CNN en el ámbito de la clasificación de imágenes y más específicamente en detección de meteoros.

[8] http://image-net.org/challenges/LSVRC/

2.1 Redes Neuronales Convolucionales (CNN)

Las CNN son un tipo de red neuronal con aprendizaje supervisado que permite, mediante un conjunto de capas internas, detectar patrones visuales sobre un conjunto de imágenes para finalmente realizar un proceso de clasificación o regresión. En el caso de clasificación el resultado de procesar una imagen consistirá en asignar una etiqueta a dicha imagen. En el caso de regresión, el resultado de procesar una imagen consistirá en asignar un valor numérico.

En el ámbito de este trabajo, focalizaremos los esfuerzos en realizar un proceso de clasificación mediante CNNs, con lo que esperaremos que la red neuronal será capaz de asignar una etiqueta (meteoro, no-meteoro) para cada imagen que procese.
La arquitectura clásica de una red CNN, está compuesta por una serie de capas convolucionales, funciones de activación, capas de *Pooling* y finalmente una *full connected layer* para la clasificación o regresión.

El primer nivel de una CNN es la capa de entrada. La CNN recibe como entrada los pixeles de la imagen que se pretende analizar, teniendo en cuenta que una imagen RGB de 224x224 pixeles, está compuesta por tres niveles (R,G y B) de 224x224 pixeles.
El segundo nivel de una red CNN consiste en una capa de convolución. Estas capas aplican un filtro a un conjunto de pixeles consecutivos (5x5, 3x3, etc.) para generar un único valor para dicho conjunto.

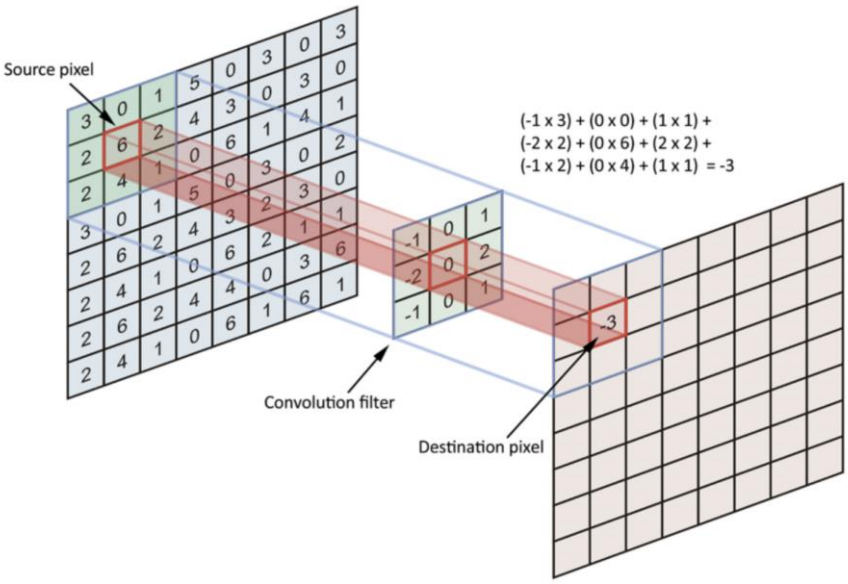


Figura 4: Aplicación de filtro de convolución
Fuente: https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

Estos filtros, también llamados *Kernels* son la base del proceso de abstracción que realizan las CNN. La aplicación de un filtro a una imagen conduce a la generación de un *feature map*
Para cada nivel de convolución se aplican distintos filtros, generando distintas versiones de la imagen de entrada, por lo tanto, al acabar una fase de convolución tendremos un conjunto de *feature maps*.

A continuación, se aplica una función de activación sobre los feature maps. Esta función de activación se encarga de activar o no activar cada una de las neuronas del *feature map* generado.

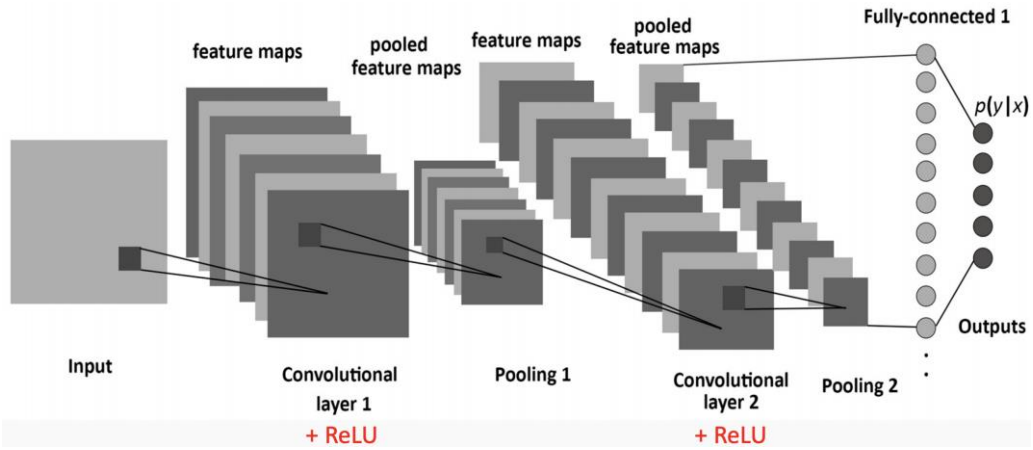


Figura 5: Ejemplo de CNN
Fuente: https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

Usualmente después de cada capa de convolución, o de algunas de ellas, se aplica una capa de *Pooling* que reduce la topología de la red y favorece el coste computacional del training.
La capa de *Pooling*, usualmente, aplica una función de max a los elementos de la red. En este ejemplo podemos observar como aplicamos un Max-Pooling de 2x2 para reducir a la mitad el tamaño de la red.

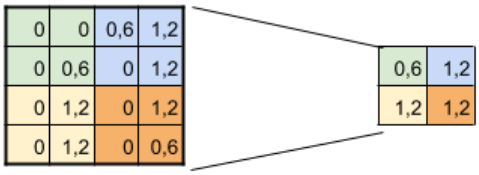


Figura 6: Ejemplo de Max-Pooling
Fuente: https:// www.aprendemachinelearning.com

Finalmente, la CNN converge a una última capa compuesta por una red neuronal full-connected que permite realizar la clasificación o regresión.
El proceso de *training* de una CNN implica iterar sobre los *kernels* o filtros que se aplican, modificándolos para así generar distintas abstracciones de las imágenes, buscando los *kernels* que deriven en un mejor porcentaje de clasificación o regresión en base a elementos de training previamente etiquetados.

2.2 CNN en el ámbito de *Image Clasification*

En la actualidad existen distintas variantes de CNN, redes con distintas topologías que aparecen para mejorar los procesos de clasificación, segmentación y reconocimiento de imágenes.

De entre las redes CNN más conocidas, destacan, LeNet-5 (1998) [1] creada por Yann LeCun, AlexNet (2012) creada por Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, VGG-16 (2015) [2] creada por Karen Simonyan, Andrew Zisserman y ResNet (2015) desarrollada por Kaiming He y otros.

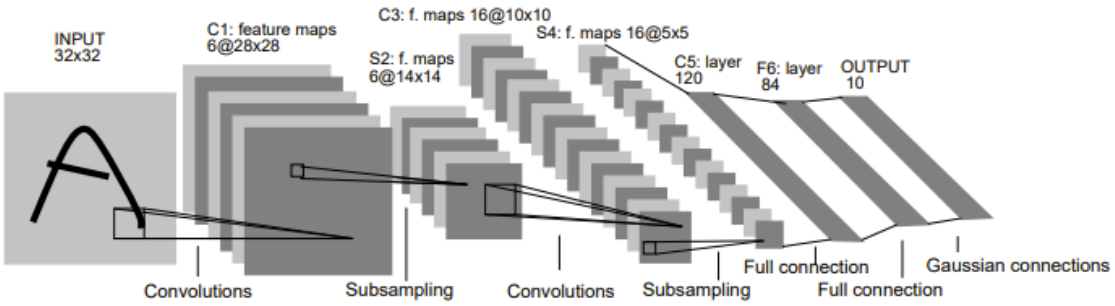


Figura 7: Arquitectura de LeNet-5
Fuente: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

D.Cecil y M.Campbell-Brown, de la Universidad de Ontario, detallan en su paper "*The Application of convolutional neural networks to the automation of a meteor detection pipeline*"[3] como el uso de las CNN les ha permitido llegar a un 99,8% de acierto en clasificación de imágenes de meteoros. Su *pipeline* de detección de meteoros parte del uso de videos del cielo nocturno, de los cuales reducen sus frames, convierten en imágenes y realizan operaciones entre imágenes consecutivas para quedarse solo con los pixeles cuyo brillo ha augmentado entre frame y frame.
Esta estrategia de restar en brillo imágenes consecutivas les permite generar unas regiones de interés (ROI) que convierten en imágenes que a posteriori etiquetan como meteoro o no-meteoro.

Otro paper interesante es el de Marcelo De Cicco y otros [4], "*Artificial intelligence techniques for automating the CAMS processing pipeline to direct the search for long-period comets".* En este paper de detalla como se ha podido automatizar el CAMS (Cameras for All-sky Meteor Surveillance) mediante CNNs, para clasificar meteoros de otros tipos de eventos que se produzcan en el cielo.

Dentro del mismo ámbito, un paper muy interesante es el de Yuri Galindo y Anna Carolina Lorena [5] (2018), "*Deep Transfer Learning for Meteor Detection",* en el que se detalla el uso de técnicas de transfer Learning sobre CNN pre-entrenadas, consiguiendo un score del 0,94.
El enfoque de *transfer learning* es muy interesante para reducir el coste en desarrollo y computacional requerido para entrenar una CNN ad-hoc para el trabajo.
El *transfer learning se* considera un campo específico del machine learning y la inteligencia artificial, que consiste en aplicar el conocimiento adquirido para realizar una tarea concreta sobre otra tarea distinta.

Para entrenar una CNN que permita realizar la detección de meteoros, usualmente se requerían implementar una serie de tareas:

- Definición de una topología de red de CNN. Tal y como se ha explicado en el apartado anterior, la topología de una CNN puede contener distintas capas de convolución, funciones de activación, capas de subsampling o pooling, y finalmente capas full-connected para realizar la clasificación o regresión
- Entrenar la red CNN previamente definida. El proceso de entrenamiento de una CNN depende directamente del tamaño de esta, y puede implicar un coste computacional muy elevado.
- Validación de resultados mediante dataset de test
- En base a los resultados adquiridos, se debe iterar sobre la definición de la topología de la red buscando mejorar los resultados

El proceso de entrenamiento de una red ad-hoc para el problema a resolver es costoso, en cuanto a tiempo de diseño, tiempo de prueba y error y tiempo computacional requerido.
El *transfer learning* consiste en adquirir de base una red CNN pre-entrenada con un conjunto extenso de imágenes generalistas. Esta CNN contendrá en sus distintas capas internas los niveles de abstracción requeridos para clasificar el conjunto de imágenes con el que ha sido entrenado.
Tomando como punto de partida esta red pre-entrenada, el *transfer learning* nos permite realizar modificaciones a la red para adaptarla al nuevo trabajo a realizar, por ejemplo, eliminar los pesos de la red *full-connected* de nuestra CNN, congelar los pesos de las capas de convolución y volver a recalcular la red *full-connected* en base al nuevo dataset de *train*.

El trabajo de Yuri Galindo y Anna Carolina Lorena, toman de partida una red ResNet. Esta red está entrenada mediante el conjunto de imágenes ImageNet.

Las ResNet, también conocidas como Deep Residual Nets son unas grandes candidatas en todo proceso de clasificación de imágenes mediante *transfer learning*. Según Galindo y Carolina, la elección de ResNet se basa en su performance y su capacidad de sostener arquitecturas muy profundas.
Las ResNet difieren de otras arquitecturas por la implementación de unas *shortcut connections*.

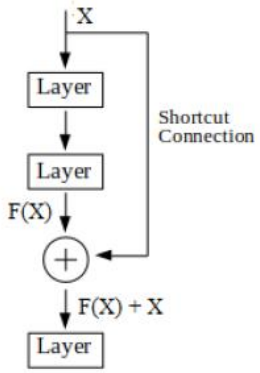


Figura 8: Uso de shortcut connections en residual networks
Fuente: Yuri Galindo y Anna Carolina Lorena [5] (2018), "Deep Transfer Learning for Meteor Detection",

Estas conexiones permiten que la entrada de una capa se sume a la entrada de una capa más profunda. Según los autores de ResNet, mediante esta estrategia, se resuelve el problema de degradación de las redes con muchas capas, y por lo tanto permiten crear ResNets extremadamente profundas, y permiten un aumento de performance respecto a otras topologías de red.

ResNet, en la actualidad dispone de distintas topologías, en base al numero de capas de la red, siendo las más conocidas la ResNet18, ResNet34, ResNet50 y ResNet101.

En su trabajo, Galindo y Carolina, comparan la performance de dichas configuraciones de ResNet para la clasificación de meteoros, y concluyen que su uso da unos resultados excelentes. El resultado del trabajo de Galindo y Carolina se toma en consideración para la elección de *transfer learning* como técnica a usar en este trabajo y ResNet como red pre-entrenada.

En otro *paper* interesante sobre el mismo ámbito, Peter S. Gural [6] evalúa distintas técnicas de Deep Learning como RNN o CNN para la clasificación de meteoros en vídeos.
Al igual que en los otros *papers* revisados, el porcentaje de éxito en clasificación de meteoros es elevado, ya sea mediante RNN con un 98,1%, con CNN con un 99,94%.

En definitiva, parece claro que las CNN son una solución que da buenos resultados en la clasificación de imágenes de meteoros. Adicionalmente se dispone de mucha información al respecto, ya sea con publicaciones o trabajos parecidos realizados.

De todas maneras, cabe destacar que en la actualidad existen técnicas fuera del ámbito de las CNN y el Deep Learning que se están usando en softwares comerciales de detección de meteoros, estas técnicas se detallaran en el apartado *2.4 Alternativas a las CNN en la clasificación de metoros*

2.3 Frameworks para implementación de CNN

En la actualidad existen distintos *frameworks* para la entrenar modelos de Machine Learning y específicamente CNNs. Mediante estos *frameworks* el desarollador puede abstraerse del hardware y software y centrarse en la implementación de los modelos mediante acceder a llamadas a APIs de alto nivel implementadas en Python o R, por ejemplo.

Específicamente en el ámbito del Machine Learning, destacan los siguientes *frameworks* (se detallan solo los que se consideran *key players* para modelado de CNNs):

- Scikit-learn[9]. Desarrollado por David Cornapeau y disponible como librerira Python. Scikit-learn está creada por encima de SciPy[10], una

[9] https://scikit-learn.org/stable/
[10] https://www.scipy.org/

librería *open source* para Python. SciPy usa NumPy [11]para realizar cálculos matemáticos, Matplotlib [12]para visualización, Pandas para manipulación de datos, etc.

- TensorFlow[13]. TensorFlow es uno de los *frameworks* de Machine Learning más populares. Desarrollado por Google, TensorFlow no solo aporta funcionalidades para el entrenamiento de modelos, si no que también aporta soporte al *data preparation*, *feature engineering*, y *model serving*.
  TensorFlow está disponible para distintas plataformas y puede ejecutarse en hardware standard con CPUs simples, así como en hardware especializado para tareas de AI con GPUs.
  El *core* de TensorFlow está desarrollado en Python, aunque existen otros módulos desarrollados mediante distintas tecnologías, como TensorFlow.js[14], desarrollado en JavaScript.
  Una de las características que hace de TensorFlow un *framework* con tanta aceptación por la comunidad, consiste en sus soluciones de inferencia en escenarios de IoT o Edge. En dichos escenarios, el dispositivo que debe realizar las inferencias tiene unas capacidades computacionales limitadas.

  Por encima de TensorFlow existe Keras[15]. Keras es una API de Machine Learning de alto nivel, que está integrado con TensorFlow 2.0. Keras permite acceder a las funcionalidades de TensorFlow desde una API de mucho más alto nivel, facilitando así las tareas más clásicas del modelado de Machine Learning.
  Keras está disponible para Python y R
- PyTorch[16]. PyTorch es un *framework open source* de Deep Learning, accesible mediante librerías en Python y C++, construido para ser flexible y modular. Esta basado en Torch una librería escrita en C, también open source, para implementar funciones con alto nivel computacional.
  PyTorch fué creado en Facebook en octubre del 2016, y en los últimos años ha adquirido cierta popularidad en los sectores más académicos.
  Al igual que TensorFlow, PyTorch es compatible con NumPy.
  Implementar redes neuronales en PyTorch es simple e intuitivo, y además el *framework* es compatible con sistemas solo con CPUs y con GPUs.
  PyTorch ha colaborado con Amazon Web Services, para desarrollar TorchServe[17],una herramienta de inferencia para modelos PyTorch con baja latencia y alta performance.
  De igual manera PyTorch también ha colaborado con Kubernetes para desarrollar TorchElastic [18]una herramienta *open source* para entrenar Deep Neural Networks mediante contenedores.
  También es interesante remarcar la existencia de PyTorch Hub[19], una herramienta que permite explorar modelos pre-entrenados de PyTorch.

[11] https://numpy.org/
[12] https://matplotlib.org/stable/index.html
[13] https://www.tensorflow.org
[14] https://www.tensorflow.org/js
[15] https://keras.io/
[16] https://pytorch.org/
[17] https://github.com/pytorch/serve
[18] https://github.com/pytorch/elastic
[19] https://pytorch.org/hub/

Por encima de PyTorch, existe Fast.ai[20]
Fast.ai es un *framework* de Deep Learning, creado por Jeremy Howard y Rachel Thomas que está ganando mucha popularidad los últimos meses por su facilidad de uso y buenos resultados.
Los beneficios de Fast.ai incluyen:

- Drástica reducción de código a desarrollar para implementar las tareas más comunes
- Comunidad de usuarios extensa. Documentación muy detallada
- *Framework* reactivo a las necesidades. Se implementan funcionalidades de alto nivel que cubren necesidades que la comunidad reclama.
- Funciones de alto nivel para *data augmentation*
- Funciones de alto nivel para *fine tuning*
- Funciones de alto nivel para análisis de hiperparámetros como *learning rate*
- Construido sobre PyTorch, que, por su naturaleza, es muy adecuado para experimentación e iteración.
- Compatibilidad con Google Colab, un servicio que proporciona notebooks con GPUs.

Si tomamos como indicador los *trends* de Stack Overflow, podemos observar como TensorFlow/Keras, que se desarrollaron con anterioridad, siguen sumando más actividad que sus competidores.
Cabe destacar el crecimiento de Pytorch, que, como se ha comentado gana popularidad en sectores más académicos y en plataformas como Kaggle.

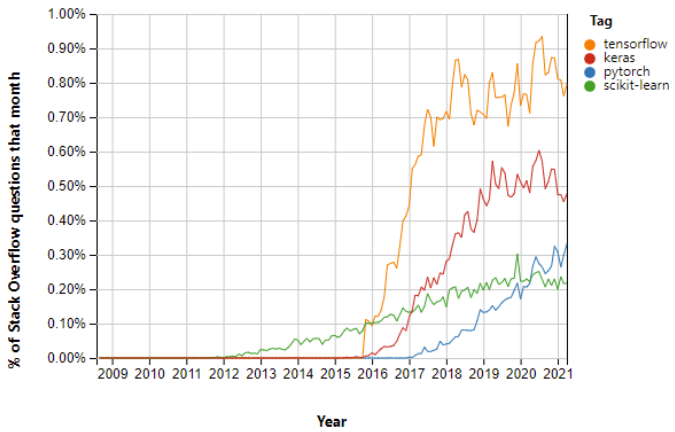


Figura 9: Actividad de las principales librerías de ML en Stack Overflow
Fuente: https://insights.stackoverflow.com/

Por último, analizando el *State of Data Science and Machine Learning 2020*[21] de Kaggle, podemos observar como Python sigue dominando en el mundo de los *frameworks* de machine Learning, y como Scikit-learn aparece en gran medida por su aplicabilidad en la resolución de múltiples problemas.

[20] https://www.fast.ai/
[21] https://www.kaggle.com/kaggle-survey-2020

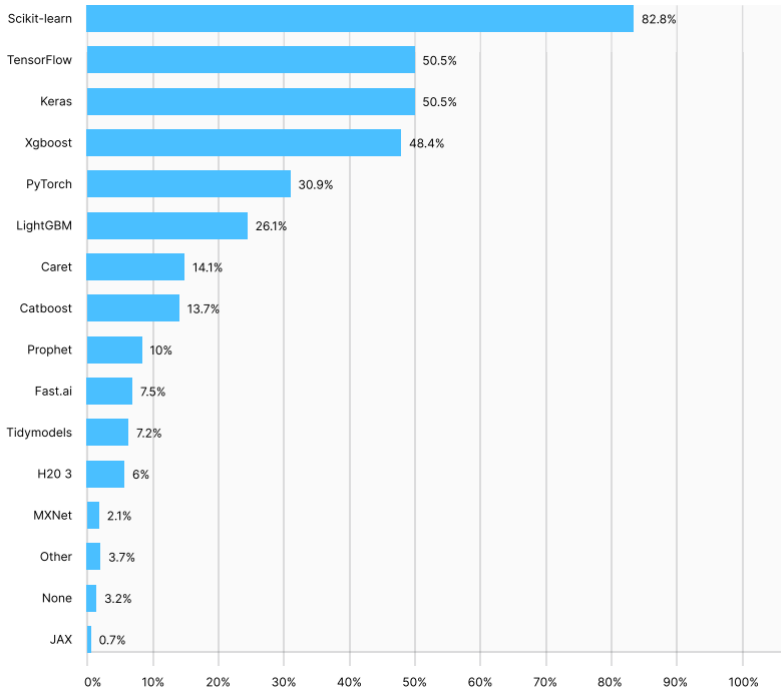


Figura 10: Machine Learning Framework Usage
Fuente: https://www.kaggle.com/c/kaggle-survey-2020

Si focalizamos en Deep Learning, podemos ver como TensorFlow y Keras siguen apareciendo de forma notable, mientras que cabe destacar el crecimiento de PyTorch de un 26% en el 2019 a un 30% en 202.

2.4 Alternativas a las CNN en la clasificación de meteoros

Se detallan a continuación algoritmos y softwares disponibles para la detección y clasificación de meteoros en videos o imágenes fuera del ámbito de las CNN, detallados por Peter S. Gural [7] en el documento "*Algorithms and Software for Meteor Detection*".

Entre los algoritmos destacan dos grandes grupos, los que tienen la funcionalidad de eliminar el *background* de la imagen y los que permiten detectar la forma del meteoro.
Para la eliminación del *background* se usan técnicas de *Clutter Suppression*, aplicar la media o restar *frames* consecutivos.
Para la detección de la forma del meteoro, existen técnicas más elaboradas, como por ejemplo *Matched Filter*, la transformada de *Hough*, *Orientation Kernel, Pixel Clustering*, etc.

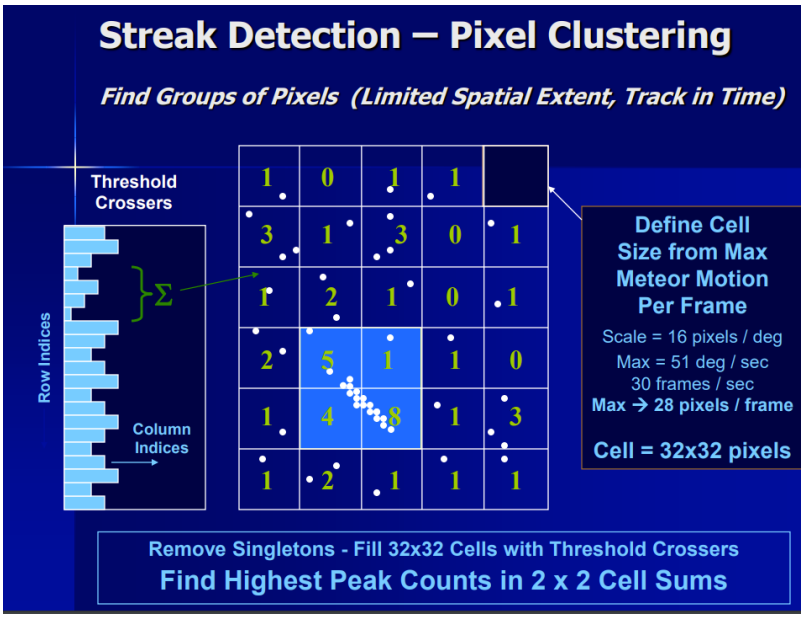


Figura 11: Detección de meteoro mediante Pixel Clustering
Fuente: *https://www.researchgate.net/publication/226649454_Algorithms_and_Software_for_Meteor_Detection*

Existen softwares de terceros para la detección de meteoros. Estos softwares usan los algoritmos que se han comentado, fuera del ámbito del Deep Learning
Unos ejemplos de estos softwares son: MeteorScan, MetRec [22] y UFOCapture[23]
Estos softwares son compatibles con Windows, con lo que su uso implica disponer de un PC o similar.
Adicionalmente cada software puede requerir de hardware específico, como por ejemplo MetRec que espera una señal de video en formato PAL o NTSC.

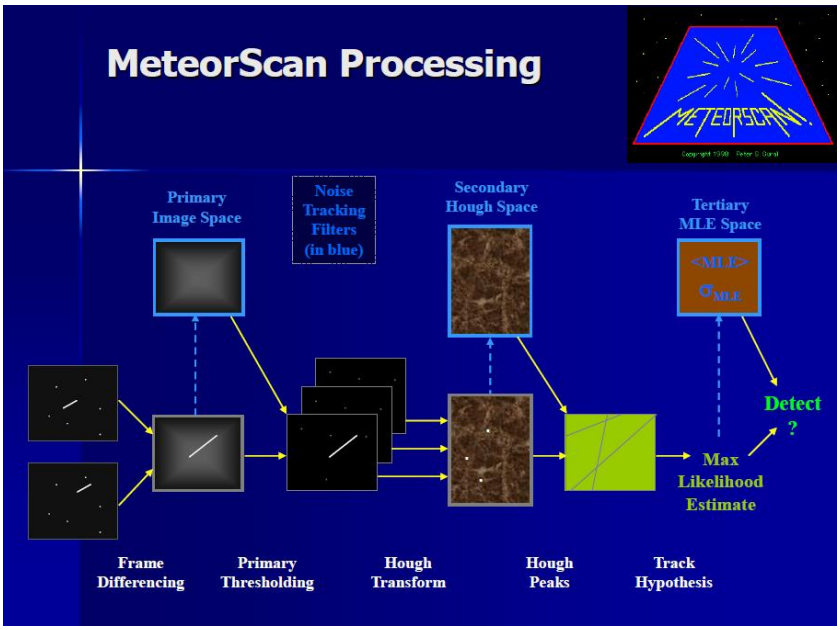


Figura 12: Pipeline para la detección de meteoros usado por MeteorScan
Fuente: *https://www.researchgate.net/publication/226649454_Algorithms_and_Software_for_Meteor_Detection*

[22] https://www.metrec.org/
[23] https://sonotaco.com/

Como se puede observar en la Figura [7], es muy usual en todo proceso de detección de meteoros, ya sea mediante CNN o mediante algoritmos de visión tradicionales, realizar una operación previa sobre dos imagenes consecutivas para conseguir la eliminación del fondo estático. En el caso específico de MeteorScan, se aplica una resta entre frames y un filtro con threeshold para eliminar ruido de fondo.
Esta estrategia se repite en varios de los papers o documentos revisados, con lo que se valorará en este trabajo como parte del pre-procesado de las imagenes previamente a la ejecución del modelo.

2.5 Conclusiones

El estado del arte de la detección de meteoros parece claramente decantarse hacia el Deep Learning, y más específicamente al uso de CNNs.
Cabe destacar que existen productos comerciales que no usan las CNNs para la detección de meteoros y que se basan en otras técnicas de visión por computador más tradicionales. Pero en base a los trabajos analizados se detecta una tendencia clara al uso de técnicas de clasificación mediante CNNs. Esta tendencia se fomenta en una clara evolución de esta tecnología durante los últimos años, acercándola a todos los públicos y *hardwares*.
Conceptos como el *transfer learning*, acercan más si cabe esta tecnología a cualquiera, puesto que permiten entrenar modelos con *datasets* pequeños y con un coste computacional relativamente bajo.
Trabajos consultados consiguen excelente rendimiento clasificando meteoros usando *transfer learning* y CNNs pre-entrenadas.
De entre las CNNs pre-entrenadas, destaca ResNet y sus distintas topologías de red.
Python se cimienta como el lenguaje de programación más adecuado, y específicamente *frameworks* como PyTorch destacan por su versatilidad.
De entre los *frameworks* analizados, destacan por su polivalencia Fast.ai / PyTorch.
Fast.ai proporciona una API de muy alto nivel para acceder a los recursos de PyTorch. En Fast.ai destacan una curva de aprendizaje muy elevada, una documentación completa y clara, una comunidad muy activa y una serie de recursos como Google Colab que permiten entrenar modelos de forma rápida y eficiente e iterar sobre distintas topologías de redes neuronales con un esfuerzo muy bajo.
Adicionalmente Fast.ai proporciona unas herramientas muy completas para la data augmentation.

Las conclusiones de la evaluación del estado del arte nos permiten tomar decisiones directas sobre el enfoque de este trabajo.
Se ha optado por:

- Realizar la detección de meteoros mediante una red ResNet y *transfer learning*.
- Usar Fast.ai como *framework* para entrenar e inferir

En los siguientes apartados se detallarán y justificarán las decisiones tomadas.

3. Implementación del trabajo

La implementación del trabajo consiste en desarrollar un *pipeline* completo que permita realizar la detección automatizada de meteoros y notificar dicha detección para que un operador la valide.
Se muestra a continuación un esquema en el que aparecen todas las piezas desarrolladas para este trabajo.

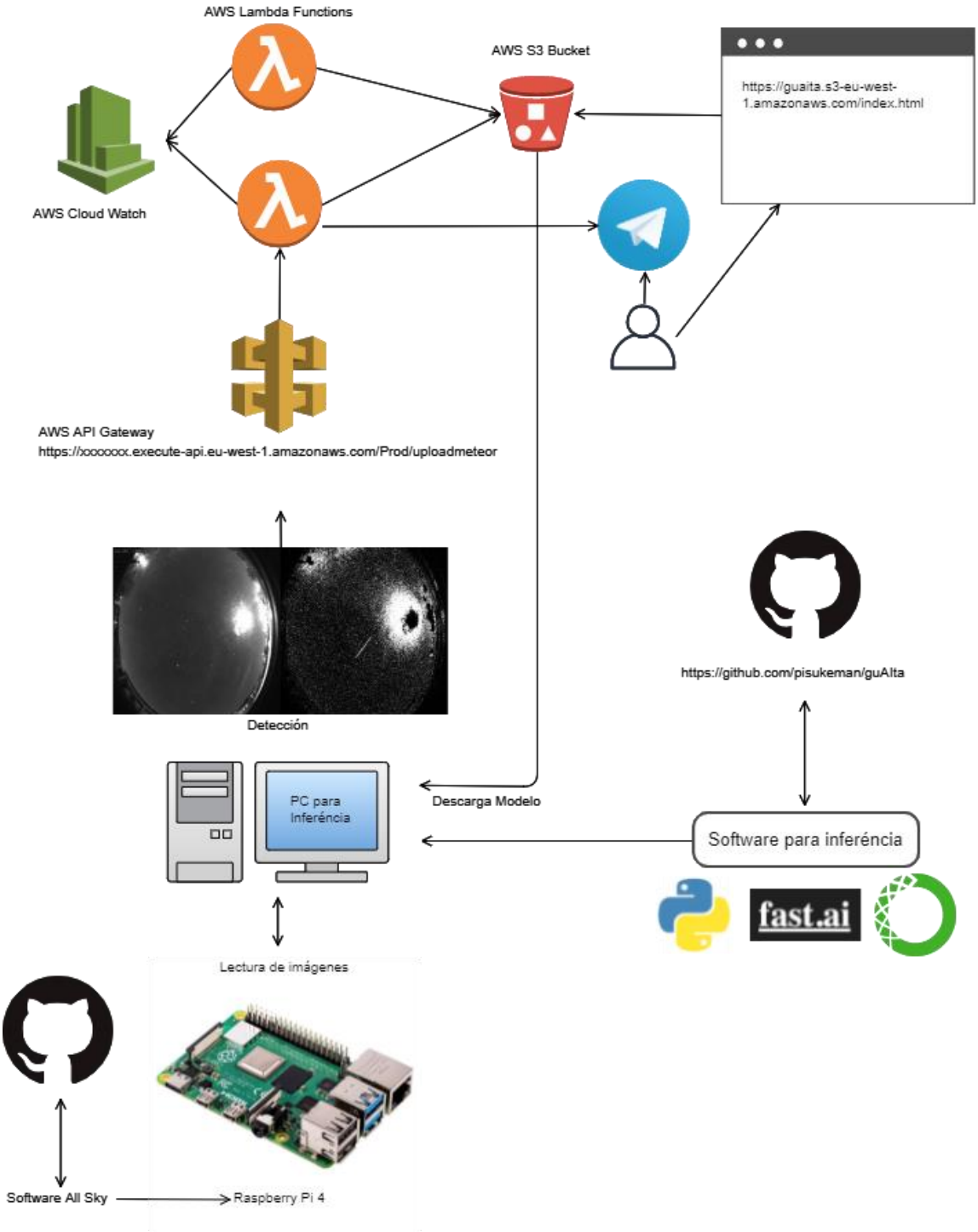


Figura 13: Esquema del pipeline para detección de meteoros
Fuente: elaboración propia

Se enumera a continuación cada uno de los elementos que se han implementado en este trabajo para completar el circuito de detección automatizada. Se realiza una explicación de alto nivel porqué en posteriores capítulos de esta memoria se detallará cada uno de los elementos:

- Modelo de Deep Learning para la detección de meteoros en base a imágenes capturadas por una Raspberry Pi4 mediante el software *open source* AllSky. Dicho modelo se ha creado en base a un dataset de imágenes propiedad del Observatori de Pujalt y cedido para este trabajo.
  El modelo se ha creado mediante el framework Fast.ai, usando técnicas de Transfer Learning y usando una CNN pre-entrenada ResNet34
  Para poder realizar el modelado se han requerido de una serie de funciones desarrolladas en Python y disponibles en el repositorio Github del trabajo.
    - Módulo de *data preparation (dataPreparation.py)*: Conjunto de funciones Python para generar distintas versiones de pre-procesado de imágenes y poder evaluar así la que nos de mejor rendimiento. Esta librería de trabajo contiene funciones que modifican las imágenes del dataset aplicando distintos filtros o operando entre ellas para generar imágenes en las que sea mas fácil la identificación visual de meteoros para el etiquetaje, y para facilitar el proceso de training del modelo eliminando el fondo y realzando ciertas características de la imagen.
      Mediante estas funciones se han generado distintas versiones del dataset. (se detallarán todas ellas en el apartado *3.2 Pre-procesado de imágenes*)

  El proceso de modelado se detallará en el apartado *3.3 Desarrollo del proceso analítico*
- Software Python para la inferencia (*guAIta.py*): Este software está disponible en el repositorio Github del trabajo y está pensado para ejecutarse desde algún PC que tenga acceso a la Raspberry.
  Se propone la instalación de un servidor Samba [24]en la Raspberry para así poder acceder a las imágenes que se generan desde el PC de inferencia. El software contiene un conjunto de funciones pensadas para realizar la inferencia en base a las imágenes que se van generando durante la noche.
  Este software implementa las siguientes funciones básicas:
    - Descarga de la última versión del modelo de detección de meteoros. La última versión del modelo se encuentra disponible en un *bucket* S3 de AWS. El software para inferencia se encarga de detectar la presencia del modelo en la instalación local y descargarlo si no está presente.
    - Inferencia en tiempo real: El software desarrollado escanea el directorio de la Raspberry donde se generan las imágenes y al detectar que se genera una imagen nueva le aplica ciertas transformaciones y la pasa por el modelo para su clasificación.
        - En caso de producirse un positivo y clasificarse la imagen como meteoro, se realiza una llamada a una API en AWS donde se procesa el positivo.

[24] https://www.samba.org/

- Inferencia en *batch*: El software desarrollado escanea el directorio de la Raspberry donde se han generado todas las imágenes de la noche anterior y procesa una a una las imágenes.
  Al igual que en la inferencia en tiempo real, el proceso aplica transformaciones a las imágenes y las pasa por el modelo en búsqueda de positivos. También de igual manera, en caso de un positivo lo notifica a través de la API en AWS.

Para la instalación del software de inferencia se recomienda el uso de Anaconda [25] como gestor de entornos, y se proporciona un fichero *yaml* [26] con las librerías necesarias para su ejecución (disponible en GitHub)
Se proporcionarán más detalles del software para inferencia en el capítulo *3.5 Implementación del Pipeline*

- Capa servidora en AWS para la gestión del *pipeline*: Se ha implementado una capa servidora en AWS mediante tecnología *serverless* (sin necesidad de ningún servidor o instancia dedicada al proyecto), usando servicios gestionados de AWS. La capa servidora da soporte a las siguientes funcionalidades del *pipeline*:
  - Permite que los PC para inferencia reporten casos positivos detectados. Mediante una llamada *https* a una API creada mediante el servicio API Gateway de AWS[27], los PC pueden enviar información de la inferencia realizada (mediante un fichero JSON), así como imágenes que permiten al operador validar si se trata de un positivo real o un falso positivo.
    - Al llamarse a la API, se ejecuta una función Lambda de AWS[28] que procesa el caso positivo.
      La función Lambda almacena el caso positivo e invoca un *bot* de Telegram que publica en un canal público la detección realizada.
    - El almacenaje de imágenes y documentos JSON se realiza en un *Bucket* S3 de AWS mediante una estructura de carpetas específica.
  - Almacenar todos los positivos detectados para su incorporación al proceso de entrenamiento del modelo para su mejora continua. Como se explicará a posteriori el numero de casos positivos en el dataset del que se dispone es muy bajo y por lo tanto es requerido que las detecciones que se van generando sean validadas por un operador e incorporadas al proceso de entrenamiento.
  - Almacenar la ultima versión del modelo para que los PC de inferencia puedan descargarlo.
  - Generar una página web estática que permite al Operador visualizar un resumen de las detecciones realizadas ordenadas por fecha. Esta página web la genera una función Lambda que se ejecuta periódicamente cada mañana, consulta las detecciones del día, genera una web *html* estática y la deja en un *Bucket* S3 de AWS para que así, pueda ser accesible desde un navegador.

[25] https://www.anaconda.com/
[26] https://yaml.org/
[27] https://aws.amazon.com/api-gateway/
[28] https://aws.amazon.com/es/lambda/

- Almacenar logs de la actividad en AWS Cloud Watch[29]

Se proporcionarán más detalles de la capa servidora y las herramientas para el operador (canal de Telegram y *website*) en el apartado *3.5 Implementación del Pipeline*

En los siguientes apartados se explicará como se ha construido este *pipeline* de detección de meteoros, poniendo especial énfasis en la parte del modelo de Deep Learning, que es el *core* de todo el trabajo desarrollado.

3.1 Descripción del dataset

Para la elaboración del trabajo no se disponía de un dataset histórico de imágenes etiquetadas. Desde el Observatori de Pujalt se mantenía una ventana de 10 días de imágenes, pero estas no estaban etiquetadas.
El operador revisaba manualmente las más de 1.000 imágenes que se generaban cada noche y solo detectaba aquellos meteoros cuya luminosidad era evidente y destacaba en el análisis visual rápido al que se sometían las imágenes.
Por lo tanto, en el momento de plantear el trabajo, solamente se disponía de una ventana de 10 días de imágenes, no etiquetadas y un histórico de solamente 18 detecciones de meteoros realizados y notificados al SPMN.

El software AllSky que genera las imágenes, lo hace mediante captaciones de 30 segundos de exposición. El operador puede configurar el sistema para determinar una hora de inicio y fin del proceso, con lo que el numero de imágenes generadas por noche puede variar.
Las imágenes que se generan son a color, JPG, con una resolución de 1304 x 976. El tamaño aproximado de cada imagen es de unos 700 Kb.
AllSky genera una carpeta específica para cada día y lo hace de forma automática manteniendo un patrón de año, mes y día concatenados (por ejemplo, el día 19/02/2021 se almacena en una carpeta llamada 20210219). De esta manera se pueden ordenar numéricamente las carpetas y coincide con su marca de tiempo.
Las imágenes se generan con una marca de agua en la parte superior izquierda, que identifica la fecha/hora de su captura, así como datos de la temperatura del sensor CCD de la cámara fotográfica, la exposición y el *gain* de cada captura.

Figura 14: Ejemplo de imágenes capturadas con distintas casuísticas
Fuente: Observatori de Pujalt

[29] https://aws.amazon.com/es/cloudwatch/

Como se puede observar en la figura [12], existen varias casuísticas que pueden alterar notablemente la imagen capturada.
Existen días/horas en los que hay presencia de la luna, existen horas en las que la luminosidad varia notablemente a causa de la presencia del sol (alba y ocaso), hay noches nubosas, noches claras, noches con humedad en la lente, etc.

Estas casuísticas, juntamente con la marca de agua presente en las imágenes, pueden dificultar el aprendizaje de un modelo. (se detallará el trabajo realizado para mitigar estas casuísticas en el siguiente capítulo)

Para la generación del dataset se realizó una primera tarea, que consistió en manualmente almacenar un conjunto de datos de imágenes suficientemente grande para afrontar el training de una red neuronal convolucional.
Para ello se realizó una copia manual de los datos de un conjunto de días, mayoritariamente consecutivos.
Los días de los que se disponen imágenes, así como información sobre su volumetría son:

Fecha	Identificador	Número Imágenes	Tamaño
19/02/2021	20210219	1.413	1,01 GB
06/03/2021	20210306	1.305	947 MB
07/03/2021	20210307	1.489	1,08 GB
10/03/2021	20210310	1.395	0,98 GB
11/03/2021	20210311	1.391	954 MB
12/03/2021	20210312	1.383	924 MB
13/03/2021	20210313	1.378	938 MB
14/03/2021	20210314	1.373	924 MB
15/03/2021	20210315	1.367	933 MB
16/03/2021	20210316	1.362	939 MB
17/03/2021	20210317	1.367	967 MB
18/03/2021	20210318	1.358	982 MB
19/03/2021	20210319	1.222	848 MB
20/03/2021	20210320	1.341	956 MB
21/03/2021	20210321	1.490	1,08 GB
22/03/2021	20210322	1.390	1,04 GB
23/03/2021	20210323	1.589	1,22 GB
24/03/2021	20210324	1.608	1,24 GB
25/03/2021	20210325	1.684	1,24 GB
26/03/2021	20210326	1.570	1,09 GB
27/03/2021	20210327	1.728	1,20 GB
28/03/2021	20210328	1.396	1,02 GB
29/03/2021	20210329	1.657	1,22 GB
30/03/2021	20210330	1.583	1,17 GB
31/03/2021	20210331	1.284	974 MB
01/04/2021	20210401	1.392	1,03 GB
02/04/2021	20210402	1.259	936 MB
03/04/2021	20210403	1.261	930 MB

04/04/2021	20210404	1.255	880 MB
05/04/2021	20210405	1.249	881 MB
06/04/2021	20210406	1.244	858 MB
07/04/2021	20210407	1.238	851 MB
08/04/2021	20210408	1.239	895 MB
09/04/2021	20210409	1.727	1,24 GB
10/04/2021	20210410	1.508	1,12 GB
11/04/2021	20210411	1.231	876 MB
12/04/2021	20210412	1.210	850 MB
		51.936	36,9 GB

Tabla 1 : Data set para training
Fuente: Elaboración propia

A modo de resumen, se dispone de datos de 37 días. Estos datos suponen 51.936 imágenes y un total de 37 GB de información.
Para el dataset de training se estima que es una buena aproximación disponer de días consecutivos, para así, cubrir casuísticas periódicas, como, por ejemplo, las fases lunares.
Para afrontar este trabajo con mayores garantías, sería interesante poder disponer de imágenes de distintas épocas del año, en las que probablemente se reproducirían la totalidad de eventos que pueden afectar a las imágenes generadas. Al no disponer de un histórico se trabajará con este dataset y en todo caso el trabajo realizado permitirá la incorporación a posteriori de más imágenes para enriquecer aún más el modelo generado.

Una revisión visual del dataset nos permitió ver que efectivamente se disponía de imágenes con distintas fases lunares, así como con distintas meteorologías: Noches claras, nubladas, lluvia, alta humedad, etc.

Estos datos se movieron a un entorno de desarrollo local, puesto que el elevado tamaño hacia complicado trabajar con ellos en entornos *cloud*.
La estrategia al respecto consistió en trabajar localmente con las imágenes hasta tener una versión definitiva del dataset en el formato final para entrenar el modelo, con imágenes de tamaño reducido y más fácilmente transportables a entornos *cloud* como Google Colab o Kaggle.

Adicionalmente al dataset para training, y como buena practica para evaluar el rendimiento del modelo, se generó un dataset específico para testear el modelo.
Existen distintas estrategias para afrontar el *testing* del modelo, algunas de ellas consisten en separar un conjunto de imágenes del dataset de training para realizar posteriormente el test de la performance del modelo.
En nuestro caso se descartó este procedimiento por el siguiente motivo:

- Una característica importante del dataset para training es que, al estar compuesto por imágenes temporalmente consecutivas, en ocasiones, son muy parecidas.
  Si optamos por generar un subconjunto de imágenes para test del conjunto total de imágenes para training, es muy probable que el

rendimiento del modelo sea engañoso y nos produzca resultados sobredimensionados.

Por este motivo, se optó por generar la captura manual de 4 días adicionales, apartados temporalmente del dataset de *train*, para así poder evaluar el modelo con imágenes que nunca ha visto en fase de *train* y validar que no se produce *overfitting*.

Para ello se generó un pequeño dataset específicamente pensado para testear el modelo una vez entrenado.

El dataset para *testing* se compone de los siguientes días y volumetría:

Fecha	Identificador	Número Imágenes	Tamaño
18/04/2021	20210418	1.181	850 MB
22/04/2021	20210422	1.544	1,40 GB
29/04/2021	20210429	1.416	1,14 GB
04/05/2021	20210504	1.105	924 MB
		5.246	5,13 GB

Tabla 2: Data set para test
Fuente: Elaboración propia

Adicionalmente al dataset de *train* y *test*, se dispone de 18 imágenes de detecciones de meteoros de baja magnitud (muy brillantes). Estas 18 imágenes son detecciones manuales de meteoros realizadas por los operadores del Observatori de Pujalt.

Figura 15: Imagenes con detecciones positivas
Fuente: Observatori de Pujalt

Las imágenes de la figura [15] muestran casos extremos, muy poco probables, que, en la actualidad, por la facilidad de detección visual, son los únicos que se reportan.

Por el contrario, los meteoros menos brillantes, los llamados popularmente, estrellas fugaces, se producen con mayor frecuencia, pero no se reportan por tener menos interés científico y por la dificultad de detectarlos con el examen visual que realiza el operador.

En nuestro caso, el trabajo realizado pretende ser capaz de detectar ambas casuísticas, los muy luminosos y los menos luminosos, asumiendo que las propiedades de las CNN nos permitirán detectar los destellos luminosos que se producen, ya sean estos muy luminosos y mas gruesos, o menos luminosos y delgados.

Por lo que se refiere a las 18 imágenes con casos positivos reportados, se da el caso que algunas están presentes en el dataset de training, pero la mayoría no. Estas imágenes se usaron para validar el modelo, para ello se incorporaron al dataset de test. Adicionalmente después de cada entrenamiento del modelo se revisó específicamente que el modelo fuera capaz de detectar estas imágenes como positivas y con un *scoring* elevado. De esta manera si el modelo nos da buen rendimiento en el dataset de test (mayoritariamente compuesto por meteoros débiles) y adicionalmente validamos que los 18 ejemplos con casos positivos son detectados, podremos asumir que el modelo nos cubrirá detecciones de meteoros débiles y más luminosos.

Como ya se ha comentado con anterioridad, el dataset del que se dispone, ya sea para *train* o para test, no está etiquetado, por lo tanto, será requerido realizar un proceso manual de etiquetaje de las 52.000 imágenes de *train* y las 5.200 de test. Para realizar el proceso de etiquetado, se pre-procesarán las imágenes disponibles para realzar elementos móviles en imágenes consecutivas.
El proceso que se realiza para facilitar el etiquetado consiste en:

- Convertir las imágenes a escala de gris
    - Mediante esta conversión las imágenes pasan a ser de 1304 x 976 pixeles, con un valor de 0 a 255 para cada pixel, indicando el nivel de gris, siendo 0 negro y 255 blanco.
- Emparejar imágenes consecutivas
- Restar las dos imágenes consecutivas para que todos los elementos iguales en ambas imágenes se conviertan en 0 (negro)
    - El restado de imágenes aplicado resta el valor de los pixeles de ambas imágenes y en caso de que el resultado sea menor que 0, se convierte a 0. Es decir, restando la imagen A de la imagen B, se consigue que pixeles de A que son mas brillantes que en B se muestren, mientras que los pixeles de A que son menos brillantes que en B o iguales de brillo que en B se conviertan a 0 (negro).
    - Mediante esta operación esperamos que, si en A aparece un objeto más brillante que en B, se muestre, mientras que las zonas que se mantienen iguales en ambas imágenes desaparezcan.
- Aplicar un *threshold* de 3,5 para que todos los pixeles inferiores a este valor se conviertan en 0 (negro) y los superiores se conviertan a 255 (blanco)
- Realizar un *resize* de la imagen resultante a 400 x 400 pixeles
    - Al aplicar un *resize* se aplica un algoritmo de interpolación que puede añadir pixeles grises. En nuestro caso se aplica INTER_AREA, que es el algoritmo recomendado en reducción de imágenes.

El resultado de estas transformaciones permite facilitar la detección de meteoros para el etiquetaje, que era lo deseado en esta fase del trabajo, pero adicionalmente nos elimina elementos estáticos del fondo (como por ejemplo la marca de agua) y nos puede facilitar el entrenamiento del modelo. En el capitulo siguiente se comentarán distintas transformaciones y se evaluará su impacto en un proceso de entrenamiento de una ResNet.

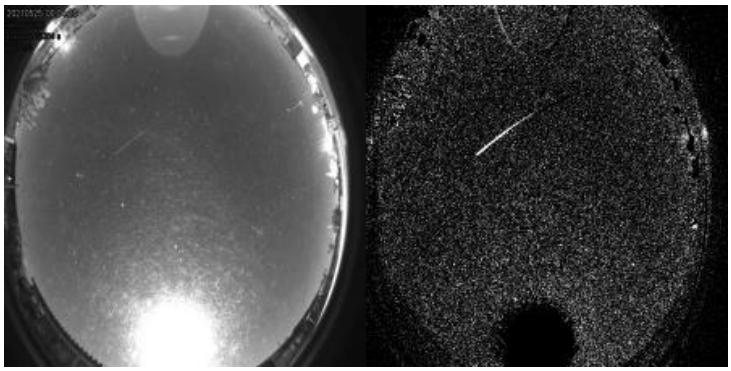


Figura 16: Transformación de imágen para el etiquetado
Fuente: Observatori de Pujalt y elaboración propia

Por lo que al valor del *threshold* aplicado se refiere, el valor de 3.5 se calculó mediante prueba y error siguiendo la siguiente metodología:

- Se seleccionaron un conjunto de imágenes positivas del día 19/02/2021, y de entre ellas se seleccionaron imágenes con meteoros tenues. Tomando como ejemplo esas imágenes, se realizaron distintas pruebas variando el *threshold* hasta encontrar el valor menor que permitiese ver los meteoros. De esta manera nos aseguramos de que no se pierdan meteoros, pero también que los elementos del fondo con valores menores al *threshold* se eliminen.
- Cabe destacar que se tomó como base el trabajo realizado por D.Cecil y M.Campbell-Brown, de la Universidad de Ontario, que en su paper "*The Application of convolutional neural networks to the automation of a meteor detection pipeline*"[3] detallan las operaciones de pre-procesado de imágenes para detección de meteoros. Una de las operaciones que realizan consiste en la aplicación de un *threshold* que cuantifican en 3,5. Tomando de base este valor, se realizó la comprobación anterior.

En la siguiente figura se puede observar en la imagen de la izquierda el resultado de aplicar un *threshold* de 3.5, mientras que en la de la derecha se ha aplicado uno de 4. Como se puede observar el tenue meteoro presente en la imagen desaparece al aplicar el *threshold* de 4.

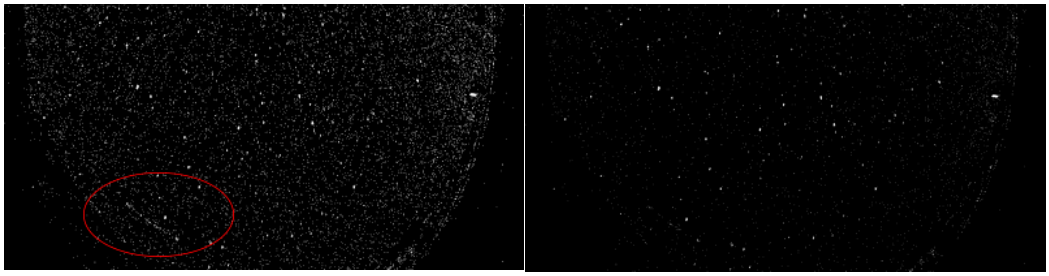


Figura 17: Aplicación de threshold en imágenes de meteoros
Fuente: Observatori de Pujalt y elaboración própia

Para efectuar estas modificaciones a la totalidad del dataset, se desarrollaron una serie de funciones Python y se usó la librería OpenCV[30]
Las funciones desarrolladas se pueden consultar en la librería Github del trabajo, en el fuente *dataPreparation.py*

Una vez transformadas las imágenes se procedió al etiquetado manual. Como cualquier proceso manual, el fallo humano es más que probable, con lo que, a posteriori durante la fase de Desarrollo del proceso analítico se realizó un proceso de *Dataset Curation* en el que se detectaron fallos de etiquetaje y se corrigieron.

Cabe destacar que es costoso diferenciar visualmente meteoros poco brillantes y trazas luminosas dejadas por satélites artificiales, como por ejemplo la Estación Espacial Internacional. En este caso, durante la fase de etiquetaje se opta por clasificar como meteoro cualquier traza luminosa que sea factible de ser un meteoro.
Dado el limitado número de imágenes disponibles y dado que los meteoros se producen con poca frecuencia, se opta por clasificar como meteoro todas las trazas luminosas detectadas, con la finalidad de que el modelo sea reactivo a estos eventos y los detecte.
Con el tiempo, y mediante la colaboración del operador que determine si se trata de un meteoro o un satélite artificial se podrá generar un conjunto de casos positivos, que excluya los satélites artificiales, suficientemente grande como para que el modelo sea capaz de clasificar entre meteoro, satélite y no-meteoro.

Se muestra a continuación el resultado del proceso de etiquetaje manual.

	No-meteor	Meteor	%Meteor	Total
Training Set	50.295	1.567	3,12%	51.862
Test Set	5.008	300	5,99%	5.308
	55.303	1.867	3,38%	57.170

Tabla 3: Resultado del etiquetado de imágenes
Fuente: Elaboración propia

Como se puede observar en la tabla anterior, existe un porcentaje de meteoros muy bajo en el dataset. Vale la pena remarcar otra vez que se han etiquetado como meteoros imágenes de estelas luminosas en el cielo que podían ser ocasionadas por satélites artificiales.

Otro punto que remarcar es el porcentaje de meteoros en el dataset de Test. Como se puede observar es muy superior al del dataset de Train.
Esto se debe a que se han colocado intencionadamente en el dataset de test imágenes de meteoros que se desean testear, por ejemplo, los 18 casos positivos reportados, así como imágenes de meteoros disponibles en condiciones distintas, como por ejemplo en noches nubladas, con doble aparición de meteoros en una misma imagen, etc. Estas imágenes que se han añadido en el dataset de Test se han eliminado del dataset de Training.

[30] https://opencv.org/

También cabe remarcar que unas pocas imágenes se han eliminado por ser defectuosas, contener errores de captura y no ser válidas para el modelado.

Por último, hay que destacar también que como es evidente, el dataset disponible esta totalmente desbalanceado. En todo proceso de clasificación es deseable que el número de casos de cada clase sea parecido, en caso contrario puede ocasionar problemas para balancear la red neuronal y los elementos de una clase concreta, minoritaria, pueden quedar diluidos dentro del proceso de training. Este problema será afrontado en el apartado de Desarrollo del Proceso Analítico.

Se muestra a continuación el proceso de creación del dataset definitivo para training y test, mediante comandos Python de Fast.ai. Se proporciona directamente capturas del Jupyter Notebook (se puede consultar la totalidad del notebook en la librería Github del Trabajo)

```
origin = Path("C:/Development/meteor_detector/dataset/v7_adjusted/dataset/")
```

```
rows_values = []
for dataset in ["test_set", "training_set"]:
    for meteor in ["meteor", "no-meteor"]:
        for fn in (origin/f"{dataset}/{meteor}").glob("*/"):
            rows_values += [(str(fn)[len(str(origin)):], dataset, meteor)]
df = pd.DataFrame(rows_values, columns=["fn", "dataset", "meteor"])
df
```

	fn	dataset	meteor
0	\test_set\meteor\image-20210219194325.jpg	test_set	meteor
1	\test_set\meteor\image-20210219195726.jpg	test_set	meteor
2	\test_set\meteor\image-20210220000355.jpg	test_set	meteor
3	\test_set\meteor\image-20210220061647.jpg	test_set	meteor
4	\test_set\meteor\image-20210220063819.jpg	test_set	meteor
...	...	...	...
57165	\training_set\no-meteor\image-20210413064838.jpg	training_set	no-meteor
57166	\training_set\no-meteor\image-20210413064908.jpg	training_set	no-meteor
57167	\training_set\no-meteor\image-20210413064938.jpg	training_set	no-meteor
57168	\training_set\no-meteor\image-20210413065008.jpg	training_set	no-meteor
57169	\training_set\no-meteor\image-20210413065038.jpg	training_set	no-meteor

57170 rows × 3 columns

```
df.groupby(["dataset","meteor"]).size()
```

```
dataset       meteor
test_set      meteor         300
              no-meteor     5008
training_set  meteor        1567
              no-meteor    50295
dtype: int64
```

Se puede observar como se crea un dataset en Fast.ai, con la ayuda de las llamadas a la API de alto nivel disponibles en la librería.

Cabe destacar la facilidad con la que Fast.ai es capaz de cargar un dataset mediante llamadas a las funciones de *Path* y *DataFrame*.
Fast.ai mediante *Path* escanea la carpeta que recibe por parámetro y es capaz de preparar una estructura que luego permite con mucha facilidad construir el dataset. En nuestro caso la estructura de carpetas en disco es la siguiente:

- *dataset*
  - *training_set*
    - *meteor*
    - *no-meteor*
  - *training_set*
    - *meteor*
    - *no-meteor*

Mediante esta estructura de carpetas, cargadas en memoria con la función Path, se crea un diccionario en Python en el que tenemos 4 columnas:

- *Id* de la imagen (autogenerado)
- *fn*: ruta y nombre de la imagen
- *dataset*: etiqueta que identifica donde está esa imagen. Los valores posibles son training_set o test_set
- *meteor*: clase de la imagen para clasificación. Los valores posibles son meteor, no-meteor

Este diccionario es pasado como parámetro a la función *DataFrame* que se encarga de crear el objeto *DataFrame* pertinente.
Disponer de un *DataFrame* Fast.ai nos permite agilizar todos los procesos que se explicarán en el apartado de Desarrollo del Proceso Analítico.

Por último, el *DataFrame* creado se puede almacenar como un *csv* y disponer así de la información sobre que imágenes contienen meteoros. (el *csv* generado está disponible en el respositorio Github del trabajo con el nombre *index.csv*)

```
df.to_csv("C:/Development/meteor_detector/dataset/index.csv", index=False)
```

El proceso de etiquetado se realizó mediante aplicar unas transformaciones a las imágenes que facilitaban la detección visual de meteoros. Estas transformaciones han resultado ser las que dan mejor rendimiento para entrenar el modelo, pero era necesario probar otras transformaciones y evaluar su rendimiento.
En el siguiente capitulo se detallan otras transformaciones probadas

3.2 Pre-procesado de las imágenes

Una vez realizado el proceso de clasificación y asumiendo que pueden existir errores de etiquetado fruto de realizar el proceso manualmente, el siguiente paso consistió en validar cual era la mejor manera de pre-procesar las imágenes que generaría el software AllSky antes de enviarlas al modelo para la inferencia.
Para ello era necesario implementar funciones Python que permitieran alterar las imágenes del dataset aplicando distintas técnicas y colocarlas en sus correspondientes carpetas de meteoro o no-meteoro. De esta manera podríamos

realizar un breve proceso de entrenamiento de una ResNet y evaluar que transformaciones son las que dan mejor resultado.

Para realizar este análisis, se estimó que no era requerido usar todo el dataset, solo un subconjunto. De esta manera se agiliza el proceso de entrenamiento de la ResNet y se pueden sacar conclusiones.

El primer pasó consistió en desarrollar funciones en la *librería dataPreparation.py* para generar distintas versiones de un subconjunto del dataset de imágenes disponibles.

De forma esquemática se muestra la herramienta desarrollada:

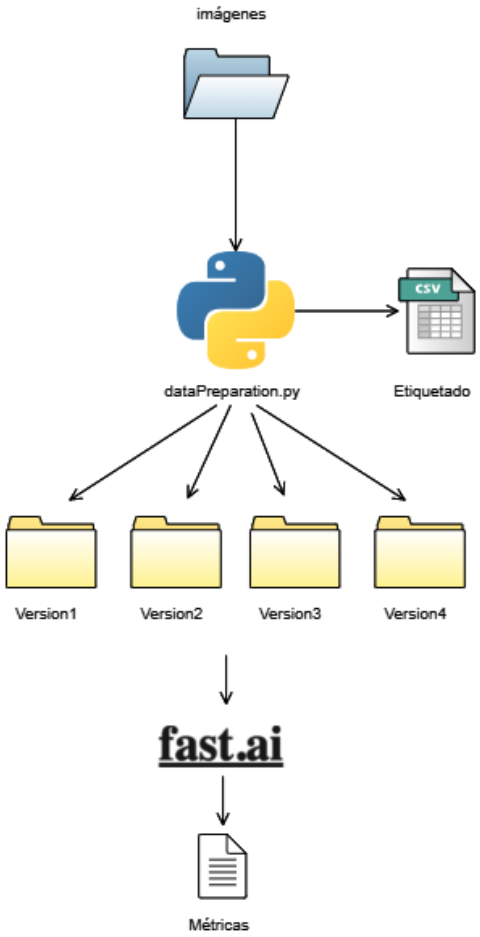


Figura 18: Generación de distintas versiones del dataset
Fuente: Elaboración propia

Como se puede observar en la figura anterior, una serie de funciones Python se encargaban de leer del dataset original de imágenes, y , crear una carpeta para cada versión del dataset generado (con un conjunto de transformaciones aplicadas). Cada carpeta de versión creada contenía una subcarpeta de meteoro y otra de no-meteoro. Las funciones Python transformaban cada imagen y la colocaban en la carpeta correspondiente basándose en la información contenida en el *csv* creado en la fase de etiquetado de imágenes.

Este proceso podía ejecutarse para todo el dataset o solo para un subconjunto del dataset original (un numero de días concreto), para así agilizar el procedimiento de *train* de la ResNet.

Se detalla a continuación las distintas versiones de pre-procesado de imagen que se evaluaron:

- Version1
    - Se transforman las imágenes a escala de gris
    - Se restan dos imágenes consecutivas
    - Se aplica un *Threshold* de 3.5
    - Se realiza un *resize* de la imagen a 224x224 sin deformar la imagen.
        - Para evitar deformar la imagen se crea un marco negro para convertir la imagen original en cuadrada
        - El *resize* se aplica con *interpolation* INTER_AREA. Esta *interpolation* es la recomendada por OpenCV para reducción de imágenes, pero el resultado no es 100% blanco y negro, puede añadir ruido en tonos de gris.
- Version2
    - Se transforman las imágenes a escala de gris
    - Se restan dos imágenes consecutivas
    - Se aplica un *Threshold* de 3.5
    - Se realiza un *resize* de la imagen a 224x224 sin deformar la imagen:
        - Para evitar deformar la imagen se crea un marco negro para convertir la imagen original en cuadrada
        - No se aplica *interpolation*. Se valida mediante histograma que solo existen pixeles blancos y negros.
- Version3
    - Se transforman las imágenes a escala de gris
    - Se restan dos imágenes consecutivas
    - Se aplica un *Threshold* de 3.5
    - Se realiza un *resize* de la imagen a 224x224. La imagen queda ligeramente estirada por no convertirla en cuadrada antes del *resize*. No se aplica *interpolation*.
- Version4
    - Se transforman las imágenes a escala de gris
    - Se restan dos imágenes consecutivas
    - Se aplica un *Threshold* de 3.5
    - Se realiza un *resize* de la imagen a 224x224. La imagen queda ligeramente estirada por no convertirla en cuadrada antes del *resize*. Se aplica *interpolation*.

Se aporta a continuación una visualización de cada una de las versiones generadas, siendo *Version1* la de más a la izquierda, seguida de *Version2*, *Version3*, y finalmente *Version4* a la derecha.

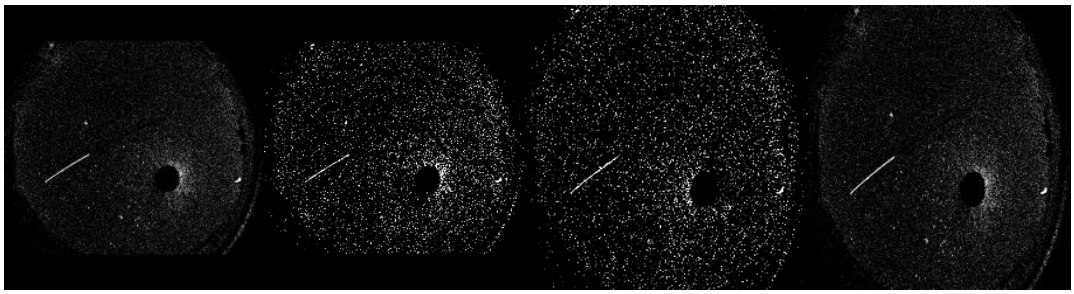


Figura 19: Visualización de las distintas transformaciones testeadas
Fuente: Elaboración propia

Para cada una de estas versiones, se genera un pequeño dataset de test y se realiza un breve entrenamiento de una ResNet34 mediante *transfer learning* y con solo 4 *epochs*. En el próximo capítulo se aportará más información sobre las ResNet y sobre el proceso de *train*, entrando en detalle sobre cada uno de los factores que involucran un proceso de training de una red CNN mediante *transfer learning.*
Para esta fase de validación de que transformación es la más apropiada nos basta con un dataset reducido y pocas *epochs*.

Se detalla a continuación para cada versión de dataset implementado el resultado que nos proporciona el modelo.
Hay que tener en cuenta que el entrenamiento realizado se ha efectuado sobre un subconjunto de datos del dataset general, mediante pocas *epochs* y que las métricas de rendimiento que se aportan se han generado a partir del subconjunto de *validation*. El subconjunto de *validation* consiste en un grupo de imágenes que Fast.ai aparta del dataset para *train* y que usa para validar el funcionamiento del modelo.

- *Version1*

```
interp.print_classification_report()
```

	precision	recall	f1-score	support
meteor	0.84	0.75	0.79	246
no-meteor	0.99	1.00	1.00	10133
accuracy			0.99	10379
macro avg	0.92	0.87	0.89	10379
weighted avg	0.99	0.99	0.99	10379

- *Version2*

```
interp.print_classification_report()
```

	precision	recall	f1-score	support
meteor	0.91	0.46	0.61	247
no-meteor	0.99	1.00	0.99	10132
accuracy			0.99	10379
macro avg	0.95	0.73	0.80	10379
weighted avg	0.99	0.99	0.98	10379

- *Version3*

```
interp.print_classification_report()
```

	precision	recall	f1-score	support
meteor	0.88	0.41	0.56	247
no-meteor	0.99	1.00	0.99	10132
accuracy			0.98	10379
macro avg	0.93	0.71	0.78	10379
weighted avg	0.98	0.98	0.98	10379

- *Version4*

```
interp.print_classification_report()
```

	precision	recall	f1-score	support
meteor	0.91	0.74	0.82	247
no-meteor	0.99	1.00	1.00	10132
accuracy			0.99	10379
macro avg	0.95	0.87	0.91	10379
weighted avg	0.99	0.99	0.99	10379

Estas pruebas realizadas deben tomarse como parte del proceso de resolución del trabajo y como herramienta para enfocar cuales son las transformaciones que mejor afectan al rendimiento del modelo. Son pruebas pensadas para ser rápidas de ejecución y aportar información que nos ayude en la toma de decisión. Estas pruebas, aunque simples, al ser en igualdad de condiciones, nos permiten ver que transformaciones son las que se desempeñan mejor.

Las métricas de *precision, recall, f1-score y support*, se explicarán en detalle en el próximo capitulo. Para el objetivo que tenemos en este apartado nos basta basarnos en *f1-score* como métrica para evaluar de forma rápida el rendimiento de las distintas versiones

Como resultado de este proceso, se optó por mantener las mismas transformaciones que se efectuaron para facilitar el etiquetaje, es decir, la *Version4*.
Realmente no hay mucha diferencia entre los resultados de la *Version1* y la 4, en realidad ambas transformaciones solo difieren en que en un caso se generó una versión cuadrada de la imagen antes de aplicar el *resize* y en el otro caso

se aplicó el *resize* deformando ligeramente la imagen. Parece claro que Fast.ai y las ResNet, para este caso concreto no encuentran problemas con una ligera deformación de la imagen, y tiene sentido puesto que lo que diferencia una imagen que contiene un meteoro de otra que no lo contiene es la traza luminosa que aparece. Dicha traza no difiere mucho si se estira ligeramente la imagen.

Por último, hay que destacar que se han realizado muchas más transformaciones a las imágenes, pero no se han incluido en este resumen por carecer de relevancia. Se han aplicado transformaciones en las que se realiza un proceso de *erode* y se reducen los pixeles sin vecinos, otras transformaciones añadiendo brillo y contraste, etc.
Estas transformaciones se han descartado mediante exploración visual de las imágenes generadas. Mayoritariamente se han descartado en base a evaluar las imágenes que según la clasificación manual realizada deberían contener meteoros y en cambio ya no era posible o complicado distinguir visualmente el meteoro después de la transformación.

Un tema importante que se debe tener en cuenta es que el pre-procesado de imagen que dio mejor resultados y que por lo tanto se usó para el modelado, implica restar dos imágenes consecutivas.
Esto tiene implicaciones claras en el software de inferencia, puesto que requerirá siempre de poder acceder al conjunto total de imágenes generadas o como mínimo a dos consecutivas.
Esta limitación se evaluó y al no considerarse un problema se optó por seguir con este pre-procesado.

A destacar que, tal y como se comentará en el próximo capitulo, el modelo resultante da buenos resultados incluso sin realizar la resta de imágenes consecutivas, aunque al estar entrenado con imágenes restadas, obviamente el mayor rendimiento lo da cuando estas vienen de esta manera.

3.3 Desarrollo del proceso analítico

Llegados a este punto, se dispone de un dataset con imágenes pre-procesadas y etiquetadas manualmente, usando dos etiquetas: meteoro y no-meteoro.
El trabajo realizado previamente nos permitió disponer de un dataset con la estructura peticionada por Fast.ai, hecho que permitió facilitar notablemente todas las tareas que se realizaron dentro del ámbito de desarrollo del proceso analítico.

En este capitulo, se detallará todo el proceso realizado para la creación del modelo de clasificación meteoro/no-meteoro, se detallarán todos los pasos realizados y se argumentarán las decisiones tomadas.
Cabe destacar que, al arrancar el proceso analítico, se aplicaron de base dos conceptos clave discutidos y argumentados en capítulos anteriores:

- Pre-procesado de imágenes para facilitar la detección de meteoros.
  En el capítulo anterior se han detallado distintas transformaciones experimentadas en fases previas al proceso analítico. Esas transformaciones permitieron ver que, funcionalmente, restar dos imágenes consecutivas es positivo. Mediante el restado de imágenes se

eliminan componentes estáticos de las imágenes, que a priori no aportan ningún tipo de valor analítico al proceso de entrenamiento, y se realzan los elementos que aumentan de brillo entre una imagen y otra.
De base para el proceso analítico asumiremos que es buena estrategia restar dos imágenes consecutivas, y ello, nos conllevará asumir que durante el resto de los capítulos de este trabajo siempre hablaremos de imágenes que son fruto de la resta de dos consecutivas.
Asumir el restado de imágenes como técnica clave, se demuestra efectivo en varios de los trabajos reportados en el capítulo de *Estado del Arte*, incluyendo trabajos no realizados mediante técnicas de Deep Learning.
    - Cabe destacar que el hecho de trabajar con imágenes restadas permite mostrar con mayor nitidez meteoros poco brillantes, pero en ocasiones se han encontrado casuísticas muy concretas que perjudican la detección. Estas casuísticas se explicarán en el apartado de conclusiones y se aportarán propuestas de trabajo futuro para disminuir su impacto.
- Uso de técnicas de Transfer Learning y redes ResNet pre-entrenadas. Partimos de base con técnicas de Transfer Learning por varios motivos:
    - En primer lugar, porqué en el trabajo de Yuri Galindo y Anna Carolina Lorena [5] (2018), "*Deep Transfer Learning for Meteor Detection",*nos dejan claro que el resultado de aplicar Transfer Learning y específicamente mediante ResNet, da unos resultados excelentes.
    - En segundo lugar, porqué el proceso analítico vinculado al entrenamiento de redes neuronales convolucionales es extremadamente costoso en términos computacionales. Dicho proceso tiene una metodología de prueba y error, y es requerido realizar multitud de entrenamientos para llegar a un modelo que de buenos resultados. Realizar estos entrenamientos con redes CNN diseñadas ad-hoc para el trabajo sería extremadamente lento.
      El Transfer Learning permite realizar entrenamientos rápidos, puesto que solo se realiza un training parcial de la red para adaptarla al nuevo juego de datos que se le proporciona (distinto al juego de datos con la que se entrenó)

Como se ha comentado con anterioridad, para el proceso analítico se optó por usar Fast.ai, basado en PyTorch, como framework de desarrollo en Python.
En base al gran tamaño del dataset de imágenes, se optará por trabajar en local mediante Visual Studio Code y Jupyter Notebooks, usando Anaconda como gestor de entornos de trabajo.

El equipo y software del que se dispone y sobre el que se han realizado las tareas detalladas en este documento se compone de:

- Hardware con GPU para el entrenamiento de CNN. En la actualidad se dispone de un PC con una CPU i7 a 2,6GHz, 16GB de RAM y una NVIDIA geForce GTX 1650
    - Es importante disponer de una GPU NVIDIA, puesto que fast.ai recomienda específicamente su uso por:
        - Alta velocidad
        - Soporte nativo de PyTorch a CUDA

        - Altamente optimizado para Deep Learning mediante cuDNN (*NVIDA CUDA Deep Neural Network library*)
    - Se priorizó trabajar desde un entorno local para poder manejar mejor el extenso dataset de imágenes. Las alternativas al trabajo local consistían en usar *kernels* de Google Colab o Kaggle. Ambas soluciones SAAS soportan las ultimas versiones de Fast.ai
- Raspberry Pi 4 para testear el *pipeline* de detección de meteoros. En la actualidad se dispone de una Raspberry Pi 4 con 4GB de RAM y sistema operativo Raspbian instalado.
    - Se usó la Raspberry para validar que era factible disponer de un software para PC que pudiera realizar el análisis de imágenes almacenadas en la Raspberry mediante acceder a ellas vía servidor Samba. Como se ha comentado, existen dos modalidades de acceso a las imágenes para realizar la inferencia: en *batch* o en tiempo real.
    - Usando la Raspberry se validó que era factible inferir desde un PC remoto las imágenes almacenadas en la Raspberry, sin ocasionar coste computacional inasumible para el dispositivo.
- Cuenta en Amazon Web Services. Se dispone de una cuenta en AWS para el despliegue de los servicios *cloud* requeridos.
- Repositorio Github para el código implementado.

A continuación, se detallan a modo de subcapítulos, las tareas realizadas dentro del proceso analítico realizado.

3.3.1 Transfer Learning con ResNet34

Para poder detallar el proceso analítico realizado es importante aportar conocimiento sobre algunos conceptos de los que se hablaran a continuación.
Tal y como ya se ha comentado, la estrategia a seguir consistirá en realizar Transfer Learning mediante una ResNet, y específicamente mediante una ResNet34

3.3.1.1 ResNet34

La red ResNet34, como se indica en su nombre, está compuesta por 34 capas, mayoritariamente capas de convolución y una capa de neuronas *full-connected* final que es la que realiza el proceso de clasificación.

A efectos prácticos, tenemos 33 capas convolucionales que se encargan de detectar elementos visuales en las imágenes y finalmente una capa que toma la decisión de clasificar la imagen con una etiqueta u otra en base a si aparecen o no ciertos elementos visuales detectados en la fase de convolución.

Fast.ai permite descargar y usar una ResNet34 mediante acceso a los recursos que Pytorch proporciona[31]. Pytorch dispone de un conjunto de redes pre-entrenadas que pone a disposición de cualquiera que desee entrenarlas mediante metodologías de Transfer Learning.

[31] https://pytorch.org/vision/stable/models.html

Para ser exactos des de Pytorch y consecuentemente desde Fast.ai se puede acceder a configuraciones de ResNet de 18, 34, 50, 101 y 152 capas.
En la siguiente imagen se puede ver como se componen cada una de las topologías de ResNet.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figura 20: Arquitectura detallada de los modelos ResNet
Fuente: https://pytorch.org/hub/pytorch_vision_resnet/

Como se puede observar, en todos los casos, las ResNet se dividen en 5 bloques convolucionales. La capa de entrada y salida para todas las configuraciones es la misma.
Fast.ai nos permite, conocer exactamente las capas funcionales de la red que usamos, por lo tanto, si queremos analizar más al detalle que es lo que hace cada una de las capas mostradas en la imagen anterior, podemos hacerlo pidiendo a Fast.ai que nos imprima la información del modelo.
Por ejemplo, en las dos primeras líneas en la imagen anterior, podemos ver como la entrada es una capa conv1 seguida de una *max pool*. Si peticionamos a Fast.ai que nos muestra la estructura de estos dos elementos podemos ver:

```
Sequential(
  (0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
```

Se observa como a la primera capa convolucional se le aplica un *kernel* de tamaño 7x7, a continuación, se aplica *BatchNorm2d*[32], que es una función de Pytorch para aplicar normalización, seguidamente se aplica *ReLU* como función de activación y al resultado de *ReLU* se aplica un *MaxPool2d* de Pytorch, que nos reduce el tamaño de la salida de la capa.

Es importante remarcar la ultima capa de las ResNet, la capa que permite realizar la clasificación. Esta capa implementa una función llamada *softmax.* Mediante *softmax* podemos realizar la clasificación y asignar un porcentaje a

[32] https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html

cada imagen. Este porcentaje nos indicará las probabilidades que le otorga el modelo de pertenecer a una clase u otra.

También es importante remarcar, que en base a la documentación de Pytorch, el tamaño mínimo de imagen para entrenar las ResNet es de 224 x 224 pixeles. Tomaremos este valor como referencia y será el que usaremos.
Adicionalmente, se aporta el detalle de los elementos que forman la ResNet34 en el *Anexo1 Estructura de una ResNet34*

3.3.1.2 Training e hiperparámetros

Para comprender algunos de los pasos realizados en el proceso analítico, es importante conocer como se realiza un proceso de aprendizaje en CNNs.
Habitualmente el algoritmo de aprendizaje usado en CNNs se conoce como Descenso del Gradiente. El descenso del gradiente nos permite saber como ajustar los parámetros de nuestra red de tal manera que se minimice su desviación a la salida.
Básicamente lo que hace el descenso del gradiente es:

- Introducimos el conjunto de imágenes para *train* y esperamos que la CNN nos devuelva la clase de cada una de ellas. En nuestro caso meteoro, no-meteoro
- Evaluamos la función de coste o función de perdida (*loss*). La elección de la función de *loss* debe ser ajustada al tipo de problema que queremos resolver. Es importante saber si nuestra red está pensada para regresión o clasificación y en este último caso, entre cuantas clases se pretende clasificar.
    - En nuestro caso concreto la función de *loss* elegida es *CrossEntropy*
    - Fast.ai específicamente selecciona la función de *loss* en base al tipo de problema a resolver
- Nuestro algoritmo de aprendizaje trata de minimizar la función de *loss*, puesto que un *loss* cercano al 1 implica que el error en la clasificación es muy elevado, mientras que un *loss* cercano al cero implica que el numero de errores en la clasificación es bajo
- A continuación, se calcula el gradiente como la derivada multivariable de la función de *loss* con respecto a los parámetros de la red.
  Podríamos mostrarlo gráficamente como la pendiente de la tangente a la función de coste en el punto donde nos encontramos evaluando los pesos actuales.
- Con el gradiente calculado actualizaremos los pesos de la red restando el valor actual al del gradiente calculado multiplicando por una tasa de aprendizaje. Esta tasa de aprendizaje o *learning rate* es uno de los hiperparámetros más importantes en el proceso de aprendizaje.

Este procedimiento lo repetimos hasta que nuestra función de *loss* esté minimizada y el rendimiento del modelo sea el esperado. Cada una de las iteraciones que realiza el algoritmo se llama *epoch* y en ella todas las imágenes disponibles para training son evaluadas.

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Figura 21: Funciones de coste para los problemas más comunes
Fuente:https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8

La arquitectura de una CNN se considera un hiperparámetro del modelo. Los hiperparámetros permiten controlar el proceso de aprendizaje de un modelo.
Adicionalmente a los hiperparámetros que afectan a la arquitectura del modelo, existen también hiperparámetros vinculados al algoritmo de aprendizaje, que tal y como hemos comentado, quizás el más relevante es el *learning rate*.

Es importante conocer los hiperparámetros vinculados al proceso de aprendizaje, porqué afectaran al proceso de training y directamente al rendimiento posterior del modelo. Cabe remarcar que vamos a realizar proceso de *transfer tearning*, con lo que también es importante conocer como es este procedimiento.

3.3.1.3 Transfer Learning

Transfer Learning en Fast.ai consiste en :

- Arrancamos de base con una red pre-entrenada. En nuestro caso una ResNet
  - Se elimina la última capa, la de clasificación y una capa con pesos aleatorios es colocada en su lugar
  - Se congelan los pesos pre-entrenados de las capas convolucionales y se entrena solamente la última capa generada clasificando en base a nuestro dataset
  - Cuando la última capa está funcionando correctamente se descongelan los pesos pre-entrenados de las capas convolucionales y se entrena el modelo completo.
  - En este punto, asumiendo que la función de *loss* ya es baja, los cambios en las convolucionales deberían ser menores.

Este procedimiento de *transfer learning*, permite reducir drásticamente el tiempo de entrenamiento de una CNN proporcionando buenos resultados.
Como ya se ha comentado con anterioridad, partes del procedimiento analítico realizado implican prueba y error, y eso deriva en muchos entrenamientos.
Si estos entrenamientos se deben realizar desde una CNN creada específicamente para el proyecto, el tiempo requerido y coste computacional pueden ser inasumibles.
Mediante *transfer learning*, se puede afrontar un trabajo de clasificación mediante CNNs sin aportar de base restricciones al numero de capas de la red, al tamaño del dataset, etc. restricciones que se impondrían por el coste computacional requerido.

Una vez explicada la arquitectura de las ResNet, el proceso de aprendizaje con los hiperparámetros más importantes, y el procedimiento de *transfer learning* se procede a detallar el proceso analítico realizado.

3.3.2 Balanceado y carga del dataset

El primer paso consiste en balancear el dataset de *train*.

Como hemos comentado en puntos anteriores, se dispone de un dataset totalmente desbalanceado, tenemos solo un 3% del total de imágenes pertenecientes a la clase meteoro, mientras que el 97% restante pertenecen a la clase no-meteoro.
Claramente este es un problema que se debe solucionar, porqué de otro modo los pocos casos de meteoros quedarían totalmente diluidos en la red.

Para solucionar este problema se realizó una modificación sobre el dataset, igualando los valores de ambas clases. Se aporta a continuación el código Python realizado para el balanceo:

```
#Balanceo del grupo de Train
df_no_meteor_train = df[(df["meteor"]=="no-meteor") & (df["dataset"]=="training_set")].sample(n=1567)
df_no_meteor_test  = df[(df["meteor"]=="no-meteor") & (df["dataset"]=="test_set")].sample(n=1000)
df_meteor_train    = df[(df["meteor"]=="meteor")    & (df["dataset"]=="training_set")]
df_meteor_test     = df[(df["meteor"]=="meteor")    & (df["dataset"]=="test_set")]

df=pd.concat([df_no_meteor_train,df_no_meteor_test,df_meteor_train,df_meteor_test])
df.groupby(["dataset","meteor"]).size()
```

```
dataset       meteor
test_set      meteor        300
              no-meteor    1000
training_set  meteor       1567
              no-meteor    1567
dtype: int64
```

Se realizaron las siguientes modificaciones:

- Se reduce el numero de elementos no-meteor del training_set a 1.567, para igualarlo al numero de elementos de meteor
- Se reduce el numero de elementos no-meteor del test_set a 1.000

Reducir el número de imágenes de no-meteor para training, podría ser un problema. Obviamente con cuantas más imágenes alimentemos el modelo tendremos mejores resultados, pero en nuestro caso hay que remarcar una casuística importante. Las imágenes son consecutivas.
Esto implica que si seleccionamos 10 imágenes consecutivas de no-meteor, es muy probable que sean extremadamente similares.
Esta casuística abre la puerta a realizar un *sampling* de las imágenes de no-meteor sin que ello implique perder mucha información.
De todas maneras, como hemos comentado, el proceso de training es un proceso de prueba/error, y afortunadamente se dispone de un dataset de test suficientemente extenso como para validar si la reducción de imágenes de la clase no-meteor afecta al rendimiento en su clasificación.

En nuestro caso, se comprobó que el *sampling* no afectaba drásticamente al rendimiento, mientras que el hecho de no balancear ambas clases imposibilitaba el tener un modelo eficiente en conjunto.

Cabe remarcar también, que, siguiendo la misma lógica, se ha realizado un *sampling* del dataset de test, reduciendo de 5.008 no-meteoros a 1.000 no-meteoros. De esta manera agilizamos el proceso de test y conseguimos atenuar el efecto de tener imágenes consecutivas muy parecidas.

A continuación, con el dataset balanceado, se realiza el proceso de carga de datos.
Fast.ai proporciona librerías de alto nivel para cargar los datos desde un *DataFrame*.
En nuestro caso, se optó por usar el siguiente código para crear un *ImageDataLoader*.

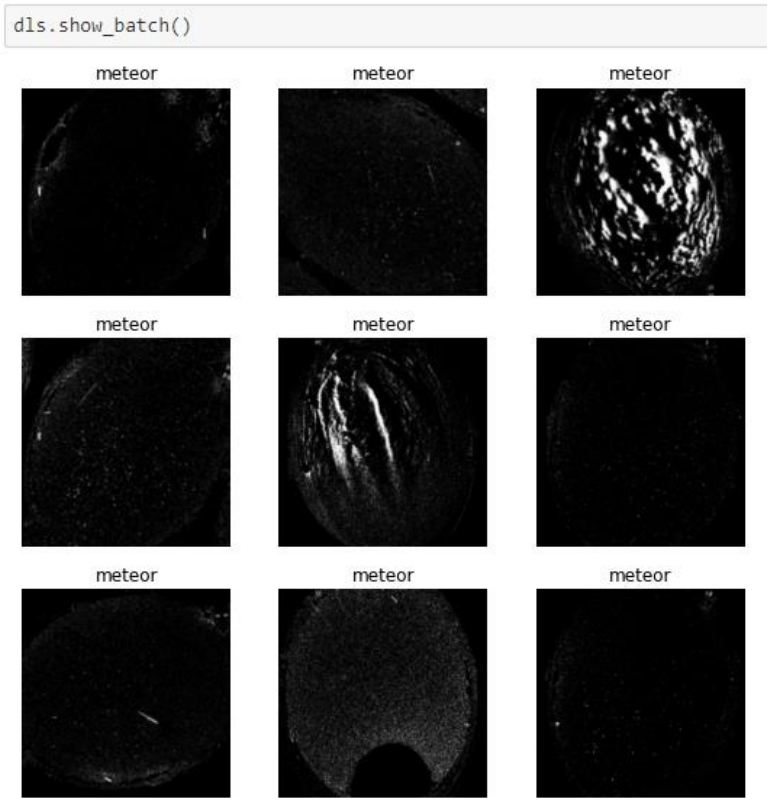
```
dls = ImageDataLoaders.from_df(df[df["dataset"]!="test_set"],
                               folder=origin ,
                               bs=32,
                               batch_tfms=aug_transforms(max_rotate=180,max_warp=0,max_zoom=0),
                               item_tfms=[Resize(224)],
                               fn_col=0,
                               label_col=2,
                               shuffle_train=True,
                               drop_last=True,
                               valid_pct=0.2,
                               num_workers=0)
```

Esta función de alto nivel permite cargar los datos que se han definido con anterioridad como un *DataFrame* (la creación del *DataFrame* se ha detallado en el apartado *3.1 Descripción del Dataset*)
Como se puede observar en el código proporcionado, se crea un *ImageDataLoader* indicando que el origen son las imágenes de *train* (no pertenecen a test_*set*)
Con *ImageDataLoader*, creamos un elemento que permitirá cargar en *batch* el conjunto de imágenes que tenemos en disco, aplicando, si se quiere, transformaciones.
Fast.ai permite indicar muchos parámetros adicionales que vale la pena detallar:

- *bs*: *Batch size*. Identifica el numero de imágenes que se cargaran en cada iteración.
- *batch_tfms*: Indica las transformaciones que se aplicarán a las imágenes que se van cargando. En el ejemplo que se aporta se puede ver como se aplican las transformaciones contenidas en *aug_transforms()*. Fast.ai proporciona una serie de transformaciones por defecto, pero en nuestro caso, es importante evaluar específicamente las transformaciones para entender su impacto. (Esta parte se detallará en el siguiente apartado de *Overfitting y Data Augmentation*)
- item_*tfms*: Transformaciones directas efectuadas antes de realizar el proceso de carga en *batch*. En nuestro caso pedimos a Fast.ai que nos realice un *Resize* a 224x224.
- *fn_col* y *label_col*: para identificar si se quiere cargar parcialmente imágenes o clases.

- *shuffle_train*: Mezclar las imágenes de *train*. Si lo marcamos a true, provocamos que se vayan cargando imágenes de forma desordenada.
- *drop_last*: Elimina el último *batch* en caso de que no llegue al numero de elementos indicado en *bs.*
- *valid_pct*: Porcentaje para indicar que un conjunto de datos se reserve para validación. Las imágenes que se reserven para validación permitirán, al finalizar cada *epoch*, conocer como está funcionando el modelo.
  Es importante explicar un poco más el concepto de validación. En todo proceso de entrenamiento de un modelo, es requerido que al finalizar cada *epoch*, el modelo se valide contra un conjunto de datos reservados específicamente para tal tarea. Los *frameworks* que nos ayudan a crear modelos, suelen permitir separar directamente un porcentaje de los datos de *train*, para *validation*. Es importante tener clara la diferencia entre imágenes de *train* y de validación porqué durante la ejecución del entrenamiento del modelo se nos proporcionará información del valor de la función de perdida del grupo de *train* y del de validación, y este dato nos permitirá conocer si se produce *Overfitting* [33]en nuestro modelo.

Mediante la función *show_batch* de Fast.ai podemos peticionar al *ImageDataLoader* que nos muestre *batch* de imágenes, donde se habrán aplicado las transformaciones configuradas.



[33]https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d

Por lo que a las transformaciones se refiere, es importante remarcar el trabajo realizado al respecto.
No todas las transformaciones que propone Fast.ai por defecto son aplicables en nuestro trabajo, por lo tanto, es importante tener claro las que se aplican y si aportan valor.
El hecho de aplicar transformaciones a las imágenes de los *batch* es crucial para solucionar los problemas presentes en el dataset: *Overfitting*, falta de ejemplos positivos e imágenes consecutivas muy parecidas entre si.
Para solucionar estos problemas necesitamos generar *Data Augmentation* en base a transformaciones en las imágenes de nuestro dataset.

3.3.5 *Overfitting* y Data Augmentation

El proceso de Data Augmentation consiste en realizar modificaciones a las imágenes de nuestro dataset, por ejemplo, rotaciones, *flip* vertical y horizontal, cambios de luminosidad, etc.
Mediante estos cambios, conseguimos:

- Reducción del *bias* del modelo hacia alguna clase particular de los datos, permitiendo al modelo generalizar bien
- Aumentar el numero de ejemplos en clases menos representadas en el dataset.
- Reducción del *Overfitting*

En general estos puntos problemáticos se han mitigado mediante el balanceo del dataset, pero cabe destacar que nuestro dataset contiene pocas imágenes. Específicamente solo disponemos de 1.500 imágenes de meteoros (entre las que existen imágenes de satélites artificiales). Si entrenamos un modelo con “solo” 1.500 imágenes de meteoros, es muy probable que nuestra red llegue a memorizar estos ejemplos y se produzca un *Overfitting*

El *Overfitting* se identifica rápidamente:

- Durante el *training* del modelo. Se puede dar el caso que el valor de *loss* sea bajo para *training*, pero en cambio aumente para *validation*. En este caso estaríamos viendo un síntoma claro de que nuestro modelo esta memorizando las imágenes de *train* y pierde rendimiento con las imágenes de validación (no usadas para *train*).
- Durante la evaluación del modelo. Si vemos que nuestro modelo ha dado buen rendimiento con datos de training, pero en cambio no mantiene el mismo rendimiento en imágenes que no ha visto nunca.

En ambos casos veremos que claramente se esta produciendo un problema de *Overfitting* y nuestro modelo no sabe generalizar, solo sabe recordar las imágenes que ha visto durante el entrenamiento.

Por otro lado, validar que no tenemos *Overfitting*, es factible. En nuestro caso hemos optado por separar totalmente un dataset de Test. Tal y como se ha comentado anteriormente, la validación del modelo no la basaremos en los datos de rendimiento sobre *train* o sobre *validation*, lo haremos sobre un dataset especifico creado en base a un conjunto de días totalmente separados del dataset de *train* y validación.

De esta manera, el verdadero rendimiento de nuestro modelo será el que nos proporcione con imágenes nunca vistas y separadas temporalmente del conjunto de imágenes usadas para *train* y *validation*.

Para poder reducir el *Overfitting*, existen distintas técnicas:

- Conseguir mas datos. Obviamente cuantos más datos proporcionemos a nuestro modelo más fácilmente podrá generalizar. Actualmente se dispone de los datos que se han explicado en el apartado de *Descripción del Dataset,* pero obviamente en el futuro se dispondrán de más datos y eso debería aumentar el rendimiento del modelo.
- Utilizar arquitecturas que permitan generalizar. En nuestro caso las ResNet
- Reducir la complejidad de la arquitectura. En nuestro caso, optamos por validar ResNet18 y ResNet34. Estas arquitecturas proporcionaron el rendimiento suficiente para evitar requerir el uso de otras más complejas.
- Utilizar técnicas de *Data Augmentation*

Como hemos comentado com anterioridade, Fast.ai proporciona APIs de alto nível para configurar el processo de *Data Augmentation*.
Por defecto Fast.ai proporciona una serie de transformaciones generalistas que se pueden indicar cuando se crea el *ImageDataLoader*. De esta manera cuando el *loader* va proporcionando imágenes al modelo se las proporciona ya transformadas.
De esta manera Fast.ai proporciona un mecanismo para que todas las imágenes que se proporcionan al modelo sean distintas entre si. En cada *epoch* se aplican transformaciones a las imágenes y estas difieren ligeramente de las originales.
Así, las probabilidades de *Overfitting* se reducen significativamente, pero podemos provocar efectos no deseados.
Si las transformaciones que realizamos generan imágenes que difieren mucho de las de test, la eficiencia del modelo no será la deseada.
Por lo tanto, el primer paso es conocer las transformaciones que aplica Fast.ai y ver cuales encajan con nuestro trabajo y cuales no.

Por defecto, Fast.ai aplica las siguientes transformaciones[34]:

aug_transforms(
mult=1.0,
do_flip=True,
flip_vert=False,
max_rotate=10.0,
min_zoom=1.0,
max_zoom=1.1,
max_lighting=0.2,
max_warp=0.2,
p_affine=0.75,
p_lighting=0.75,
xtra_tfms=None,
size=None,

[34] https://docs.fast.ai/vision.augment.html#aug_transforms

mode='bilinear',
pad_mode='reflection',
align_corners=True,
batch=False,
min_scale=1.0,
)

Se detallan las transformaciones aplicadas por defecto:

- *Flip* horizontal pero no vertical. No aplica el vertical porqué en múltiples aplicaciones de clasificación de imágenes no sería correcto realizar un *flip* vertical. Imaginemos un clasificador de imágenes de casas. Difícilmente nos seria de utilidad entrenar el modelo con imágenes en las que el tejado está en la base de la imagen.
- Rotación de la imagen de hasta 10 grados
- Aplicar zoom a la imagen
- Cambios de iluminación
- Aplicar *Warp*[35]

Fast.ai parametriza también el impacto de estas modificaciones mediante parámetros como *p_affine*, *p_lighting*, *mult*, que permiten indicar la probabilidad de que se realicen estos cambios y con que intensidad se aplican.
Otro parámetro importante es el *pad_mode*. Por defecto Fast.ai aplica '*reflection'*. Esto quiere decir que en casos en los que, por ejemplo, se produzca rotación o zoom, y algunas zonas de la imagen queden sin contenido, Fast.ai las rellena automáticamente con zonas en negro, propagando el borde o generando un reflejo de la imagen.

Figura 22: Data Augmentation en Fast.ai
Fuente: https://docs.fast.ai/vision.augment.html

En nuestro caso, se evaluaron distintas transformaciones, realizando distintas pruebas, y finalmente se consideró apropiado realizar las siguientes transformaciones a nuestro dataset:

- Mantener las transformaciones por defecto, exceptuando:
  - Rotación de hasta 180 grados de las imágenes
  - No aplicar transformación warp
  - No aplicar zoom para no perder información

[35] https://docs.fast.ai/vision.augment.html#Warping

- Aplicar *reflection* como *pad_mode*
  - Se testeó específicamente usar *pad_mode* con *zeros* y se detectó una bajada en el rendimiento del modelo cuando se evaluó contra el dataset de test.
    - El *recall* de la clase meteor bajó de un *0.98* a un *0.79*
    - El *recall* de la clase no-meteor bajó de un *0.94* a un *0.87*
      (El valor de recall y otras métricas se detallará en el siguiente capítulo.)
- Generar las imágenes con resolución 400 x 400 y peticionar a Fast.ai que genere un *resize* a 224 x 224.
  - De esta manera, facilitamos el trabajo de transformación a Fast.ai proporcionándole imágenes mayores de lo necesario y dándole margen para aplicar transformaciones más cómodamente.

Con técnicas de *DataAugmentation* implementadas sobre el *ImageDataLoader*, podemos afrontar el siguiente paso que consiste en entrenar el modelo

3.3.3 Creación del modelo con ResNet34

Mediante el *ImageDataLoader* creado, podemos crear un *cnn_learner* en Fast.ai

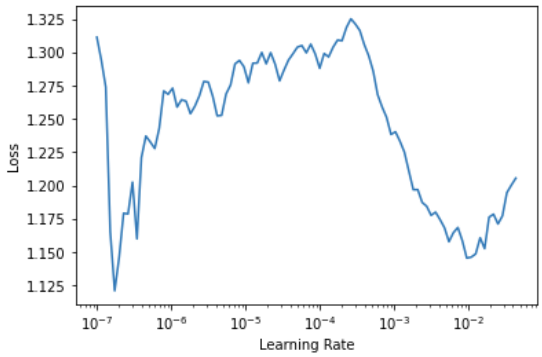
```
learn = cnn_learner(dls, resnet34,
                metrics=[error_rate, accuracy, Precision(average='macro'), F1Score(average='macro')],
                    cbs=[EarlyStoppingCallback(patience=25), ShowGraphCallback()]
                   )
```

Como se puede observar, la creación del *learner* es extremadamente fácil, solo es requerido el *ImageDataLoader*, el modelo de la red que queremos usar, en nuestro caso una ResNet34 y las métricas que queremos que se nos devuelvan. Adicionalmente le pasamos un conjunto de *callbacks* para que el *learner* interactúe con el Jupyter Notebook y actualice la gráfica con el valor de la función de *loss*.

Con el *learner* creado, podemos pasar a la definición de los hiperparámetros que necesitamos para el proceso de entrenamiento: *learning rate* y numero de *epochs*.
Para ello, podemos usar funciones de alto nivel que Fast.ai nos proporciona. Una de las más interesantes y de gran utilidad es *lr_find*

```
learn.lr_find(end_lr=0.1)
```

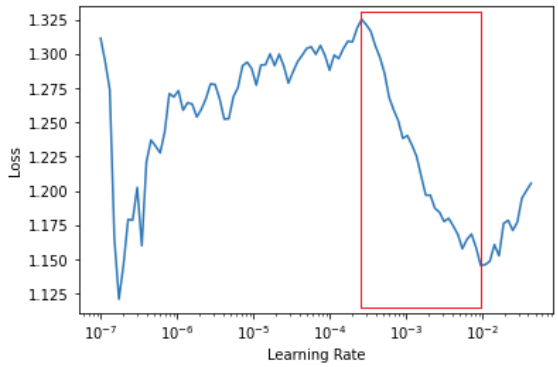
```
SuggestedLRs(lr_min=0.0009549926035106182, lr_steep=0.0005248074885457754)
```



La ejecución de *lr_find* nos devuelve una gráfica con la función de los y el *learning rate*. La gráfica nos permite ver los puntos donde el valor de *loss* disminuye en base a un valor de *learning rate* concreto.
Esta gráfica debe interpretarse de la siguiente manera:

- Hay que descartar los picos y descensos puntuales
- Hay que buscar donde se produce el descenso continuado más importante del valor de *loss*
    - Una vez identificado, hay que buscar el punto medio de la recta que provoca el descenso más elevado y seleccionar ese *learning rate* para el procedimiento de training.

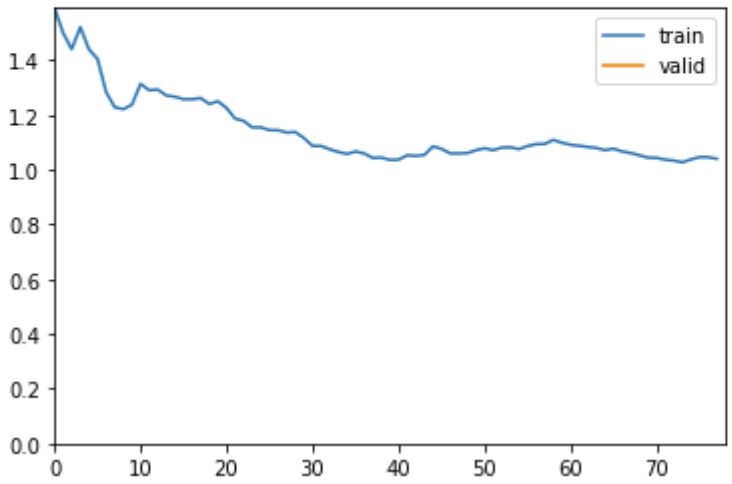
En nuestro caso, seleccionamos un *learning rate* de 0,003



Para determinar el numero de *epochs* para training, lo hacemos en base a prueba y error, basándonos en lanzar el proceso de training y evaluar cuando se estabilizan los valores de *loss* para *train* y *validation*.
En nuestro caso, seleccionamos 35 *epochs* y procedemos a la ejecución del entrenamiento de la red mediante el método *fine_tune* del *learner* creado anteriormente.

```
learn.fine_tune(35,3*1e-3)
```

epoch	train_loss	valid_loss	error_rate	accuracy	precision_score	f1_score	time
0	1.040598	0.864652	0.343450	0.656550	0.667424	0.653035	00:36



Fast.ai nos muestra una primera epoch y nos muestra un conjunto de métricas que debemos tener en consideración:

- *epoch*: número de epoch o iteración
- *train_loss*: resultado de la función de loss para el grupo de imágenes de training al finalizar esta epoch.
- *valid_loss*: resultado de la función de loss para el grupo de imágenes de validación al finalizar esta epoch.
- *error_rate*: numero de predicciones incorrectas sobre el total de predicciones realizadas.
- *accuracy*: numero de predicciones correctas sobre el total de predicciones. Nos muestra el porcentaje de acierto de nuestro modelo de forma genérica sin desglosar por cada una de las clases.
- *precisión_score*: nos responde a la pregunta de que porcentaje de identificaciones fue correcta para la clase. Por ejemplo, cuando hemos etiquetado meteoro, que porcentaje de veces hemos acertado
- *recall*: numero de casos que se han etiquetado correctamente sobre el total de casos con esa etiqueta. Es decir, cuantas imágenes hemos etiquetado como meteoros sobre el total de imágenes que contenían meteoros. Se diferencia de precisión porqué solo evalúa los aciertos sobre el total de cada clase.
- *f1_score*: es una medida que combina precisión y recall. Se define como la media harmónica de las dos medidas. La formula para calcular el f1_score sería:
    - *2 * ((precisión_score * recall) / (precisión_score + recall))*

El resultado de la ejecución de la totalidad de las 35 *epochs* es:

epoch	train_loss	valid_loss	error_rate	accuracy	precision_score	f1_score	time
0	0.735807	0.599756	0.255591	0.744409	0.752998	0.740982	0:39
1	0.632747	0.466677	0.212460	0.787540	0.793237	0.785764	0:43
2	0.570768	0.445019	0.210863	0.789137	0.812992	0.786149	0:41
3	0.489949	0.375685	0.180511	0.819489	0.819430	0.819451	0:39
4	0.467536	0.366025	0.175719	0.824281	0.824185	0.824217	0:41
5	0.450777	0.335763	0.146965	0.853035	0.859379	0.852697	0:40
6	0.402171	0.352846	0.145367	0.854633	0.864141	0.854066	0:41
7	0.394599	0.383515	0.185304	0.814696	0.822695	0.812861	0:46
8	0.369270	0.341042	0.193291	0.806709	0.827280	0.802537	0:47
9	0.344086	0.292294	0.127796	0.872204	0.875967	0.872074	0:50
10	0.340840	0.357549	0.140575	0.859425	0.872904	0.858594	0:54
11	0.321911	0.268450	0.107029	0.892971	0.894920	0.892938	0:51
12	0.302367	0.271105	0.113419	0.886581	0.886548	0.886516	0:48
13	0.292068	0.280740	0.118211	0.881789	0.889225	0.881479	0:55
14	0.284487	0.268533	0.107029	0.892971	0.895300	0.892925	0:51
15	0.288731	0.271188	0.111821	0.888179	0.890906	0.887765	0:49
16	0.254424	0.225396	0.092652	0.907348	0.907631	0.907347	0:49
17	0.269397	0.266347	0.107029	0.892971	0.900228	0.892708	0:52
18	0.258594	0.253337	0.091054	0.908946	0.910590	0.908927	0:53
19	0.245117	0.282585	0.099042	0.900958	0.905475	0.900836	0:49
20	0.226274	0.256328	0.091054	0.908946	0.909721	0.908944	0:54
21	0.215593	0.244790	0.091054	0.908946	0.908882	0.908906	0:54
22	0.208339	0.222912	0.076677	0.923323	0.923363	0.923316	0:52
23	0.214027	0.210979	0.065495	0.934505	0.934719	0.934503	0:53
24	0.198686	0.229733	0.094249	0.905751	0.908132	0.905710	0:59
25	0.178995	0.215816	0.079872	0.920128	0.920588	0.920128	0:58
26	0.187185	0.211741	0.081470	0.918530	0.919576	0.918525	0:53
27	0.184476	0.220584	0.084665	0.915335	0.915546	0.915334	0:52
28	0.166746	0.225217	0.079872	0.920128	0.920274	0.920125	0:52
29	0.154931	0.226683	0.083067	0.916933	0.917218	0.916932	0:53
30	0.159274	0.228395	0.078275	0.921725	0.921814	0.921720	0:51
31	0.162290	0.232581	0.078275	0.921725	0.921725	0.921715	0:52
32	0.151031	0.222880	0.073482	0.926518	0.926486	0.926506	0:52
33	0.140756	0.226795	0.078275	0.921725	0.921814	0.921720	0:54
34	0.146109	0.230272	0.076677	0.923323	0.923363	0.923316	0:54

Tabla 4: Resultado del training del modelo ResNet34
Fuente: Elaboración propia

El resultado del proceso de training del modelo se puede evaluar viendo las métricas comentadas anteriormente. En la tabla anterior podemos ver para cada *epoch* realizada las métricas comentadas con anterioridad.

Destacan los valores elevados de *accuracy, precisión_score y f1_score*, todos ellos por encima del 90%, lo que nos permite ver que el proceso de aprendizaje se ha desarrollado correctamente.
Cabe remarcar que estos porcentajes elevados no son reales. Son porcentajes generados durante el proceso de *learning*.
La verdadera evaluación del modelo se debe realizar sobre el grupo de imágenes de test.
Adicionalmente, podemos observar en la gráfica siguiente la evolución de los valores de la función de *loss* para *train* y *validation*. La gráfica nos muestra en el eje de las x los distintos *batch* que forman el entrenamiento. (el *ImageDataLoader* está configurado para servir imágenes en grupos de 32)
Como se puede observar, se produce un descenso del valor de *loss* durante los distintos *batch* de datos que se le van proporcionando al modelo.

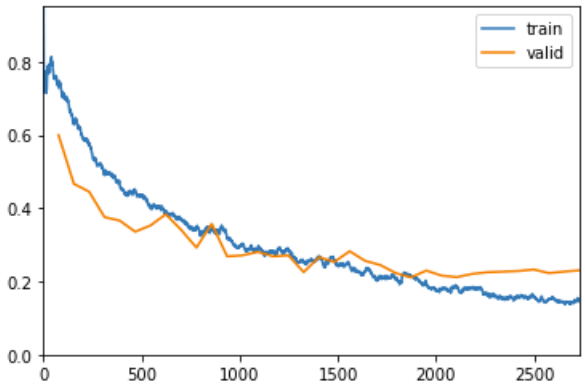


Figura 23: Gráfica de loss para train y validation (ResNet34)
Fuente: Elaboración propia

A continuación, es crucial validar el modelo contra el dataset de test. Para ello se usa una función específica de Fast.ai que permite ver las métricas por etiqueta.

```
interp = ClassificationInterpretation.from_learner(learn,
dl=learn.dls.test_dl(df[df["dataset"]=="test_set"], with_
labels=True, bs=100))
interp.print_classification_report()
```

	precision	recall	f1-score	support
meteor	0.82	0.98	0.89	300
no-meteor	0.99	0.94	0.96	1000
accuracy			0.95	1300
macro avg	0.91	0.96	0.93	1300
weighted avg	0.95	0.95	0.95	1300

Estas métricas, algo menores que las que se mostraban en el proceso de training, si que deberían mostrar las capacidades reales del modelo, y deberían reproducirse en un entorno real.

A destacar, que el modelo consigue una *precision*, *recall* y *f1-score* muy elevadas detectando la clase no-meteor, y algo más bajas para la meteor.
Un tema interesante es que precisamente la métrica que se quiere maximizar para la clase meteor, nos da resultados excelentes.
Esta métrica es el *recall*. El *recall* sobre la clase meteor, nos está indicando el porcentaje de veces que hemos etiquetado meteor sobre el total de veces que teníamos un meteor. Es decir, nos está indicando que no se nos escapan meteoros.
Con esta métrica a un 98%, tenemos garantías de que, si se produce un meteoro, lo detectará.
Por otro lado, la precisión de la clase meteor está en un 82%, este valor se debe a que la precisión evalúa las veces que se ha etiquetado meteor y se ha acertado. En este caso, esta métrica se perjudica por los falsos positivos que se producen etiquetando meteor cuando no lo es.
Cabe remarcar que el dataset de test no está balanceado, es decir, solo un 6% del dataset de test son meteoros (5.008 no-meteoros, 300 meteoros). En nuestro caso, hemos reducido el tamaño de la clase de no-meteoros a 1.000, para agilizar el proceso de test y además añadir un punto de muestreo que pueda ayudar a reducir el efecto de imágenes consecutivas muy parecidas. Finalmente, el dataset de test queda compuesto por 1.000 no-meteoros y 300 meteoros.
La diferencia entre el número de no-meteros y meteoros, provoca que, aunque el modelo funcione muy bien detectando no-meteoros, en un porcentaje bajo, se equivoca, y este numero de equivocaciones impacta en la precisión de la clase meteor, compuesta por un numero muy inferior de ejemplos.
En la siguiente figura podemos ver la matriz de confusión del modelo.

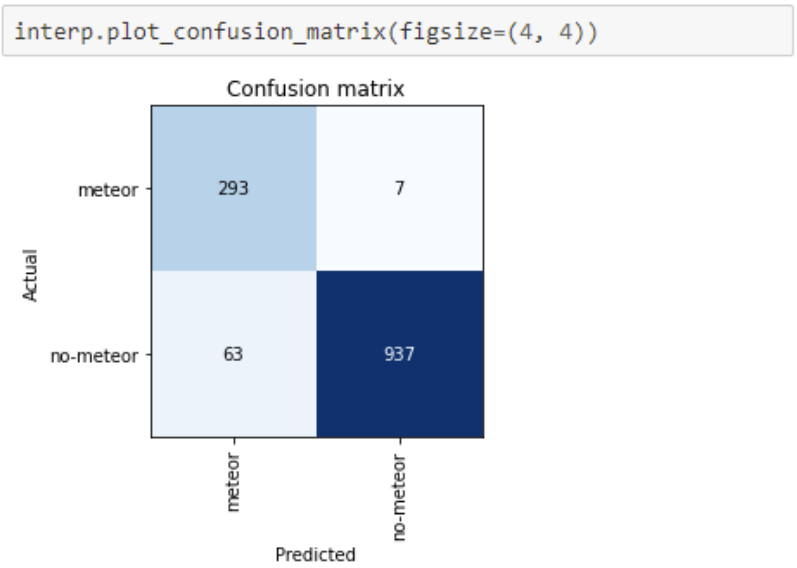


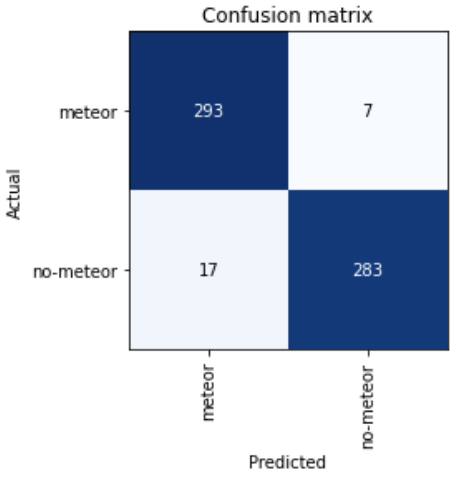
Figura 24: Matriz de confusión del modelo ResNet34
Fuente: Elaboración propia

Si para el mismo modelo, realizamos un balanceo del dataset de test y nos quedamos con la misma cantidad de meteoros y de no-meteoros, es decir, nos quedamos con 300 meteoros y 300 no-meteoros, entonces los porcentajes de *precisión y f1-score* varían considerablemente.

```
df.groupby(["dataset","meteor"]).size()
```

```
dataset       meteor
test_set      meteor        300
              no-meteor     300
training_set  meteor       1567
              no-meteor    1567
```

	precision	recall	f1-score	support
meteor	0.95	0.98	0.96	300
no-meteor	0.98	0.94	0.96	300
accuracy			0.96	600
macro avg	0.96	0.96	0.96	600
weighted avg	0.96	0.96	0.96	600



En este caso, podemos ver como porcentualmente tenemos el mismo *recall*, pero al disponer de menos casos de no-meteoro, aunque el porcentaje de acierto se mantenga, se generan menos falsos positivos de la clase no-meteor y eso afecta positivamente a la precisión de la clase meteor (y consecuentemente también al *f1-score*). Son distintas maneras de mostrar los mismos resultados, pero el modelo en base al *recall* actual, sabemos que generará un 6% de falsos positivos (meteoros que no lo son), y que detectará un 98% de los meteoros que analice.

Por otro lado, hay que comentar que, dadas las condiciones actuales de pocas imágenes positivas, tal y como se ha comentado, el valor que se pretende maximizar es el de *recall* de la clase meteor. Mediante un *recall* elevado, dispondremos de un modelo que será eficiente detectando meteoros y esto nos permitirá seguir aumentando el dataset disponible de detecciones para seguir mejorando el modelo.
Aunque se generen un 6% de falsos positivos, la tarea del operador será mucho más llevadera, reduciendo drásticamente el número de imágenes a revisar.

En base a estos valores, podemos realizar ciertas estimaciones:

- En una noche de verano, donde se generen unas 1.000 imágenes en el Observatori de Pujalt, y de estas un 3% sean meteoros, tendremos 970 imágenes de no-meteoros.

- Dado que nuestro *recall* sobre la clase no-meteoro es de un 94% se generarán unos 58 falsos positivos.
- Dado que nuestro *recall* sobre meteoros es del 98%, detectaremos 29 meteoros de los 30.

Estos valores se tendrán que refrendar en base a la ejecución del modelo en un entorno real.

Una vez finalizado el training del modelo, lo exportamos al formato pkl[36] y lo cargamos al *Bucket* S3 de AWS para que esté disponible para los clientes Python de inferencia.
El formato pkl permite almacenar el modelo y posteriormente cargarlo y usarlo para inferencia, sin necesidad de volver a realizar el procedimiento de entrenamiento.
En nuestro caso, el tamaño del archivo pkl generado para una ResNet34 es de 85 Mb.

Con el modelo entrenado, podemos acceder al listado de los *top losses*, que nos permitirá detectar errores manuales de etiquetado y corregir nuestro dataset, lo que llamamos *dataset curation*

3.3.4 Dataset curation

La primera tarea que realizar dentro del proceso analítico es asegurar que el dataset del que se dispone está correctamente etiquetado.
Esta tarea se puede realizar mediante iteraciones manuales sobre el dataset, buscando posibles fallos en el etiquetado, pero el proceso sería costoso y nuevamente abierto al fallo manual.
Fast.ai nos proporciona una herramienta muy útil, el análisis de los *top loss*
El procedimiento para realizar un análisis de *top loss* consiste en realizar un entrenamiento completo de una red, y una vez entrenada, peticionar a Fast.ai que genere un listado de aquellas imágenes que han sido etiquetadas de forma errónea y además con un *scoring* alto. Es decir, pretendemos que Fast.ai nos enseñe los errores más destacados.
Por ejemplo, mediante este código Python, generamos el conjunto de *top losses* de nuestro dataset de test, viendo en la columna *fn* si la imagen se trata de un meteoro o no, y viendo en la columna *meteor* su etiqueta real.

[36] https://pytorch.org/tutorials/beginner/saving_loading_models.html

```
losses,idxs = interp.top_losses(10)
df[df["dataset"]=="test_set"].iloc[idxs]
```

	fn	dataset	meteor
4665	\test_set\no-meteor\image-20210505004757.jpg	test_set	no-meteor
110	\test_set\meteor\image-20210419055134.jpg	test_set	meteor
4791	\test_set\no-meteor\image-20210505015835.jpg	test_set	no-meteor
1519	\test_set\no-meteor\image-20210422215737.jpg	test_set	no-meteor
583	\test_set\no-meteor\image-20210418232222.jpg	test_set	no-meteor
2751	\test_set\no-meteor\image-20210423050439.jpg	test_set	no-meteor
1610	\test_set\no-meteor\image-20210422222805.jpg	test_set	no-meteor
4836	\test_set\no-meteor\image-20210505022107.jpg	test_set	no-meteor
4632	\test_set\no-meteor\image-20210505003055.jpg	test_set	no-meteor
2735	\test_set\no-meteor\image-20210423045638.jpg	test_set	no-meteor

La idea detrás del análisis de *top loss* consiste en detectar errores de etiquetado, corregirlos y volver a entrenar el modelo.
Este procedimiento se realizó repetidas ocasiones, encontrando fallos del etiquetado manual y corrigiéndolos.
Tras cada corrección se reentrena el modelo y se vuelve a peticionar el análisis de *top loss* hasta que no se detectan errores claros de etiquetado.

También podemos peticionar a Fast.ai que nos muestre el *top loss* mediante imágenes. Como podemos observar, Fast.ai nos muestra para cada imagen la clase que ha inferido el modelo, la clase actual, el valor de *loss* y el *scoring* que le otorga el modelo

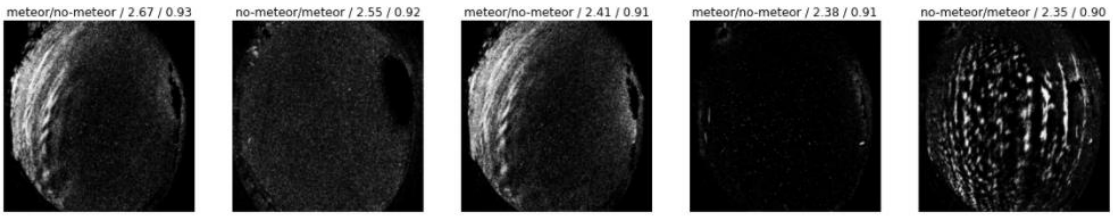


Figura 25: Análisis de Top Loss
Fuente: Elaboración propia

A destacar que podemos realizar este análisis sobre el dataset de test o sobre el de *train*, lo que nos permite reparar los problemas de etiquetado de ambos datasets.

De esta manera el proceso de entrenamiento del modelo queda completo:

- Procesado inicial de imágenes
- Balanceo de dataset
- *Data augmentation*

- Entrenamiento del modelo
- Evaluación de resultados
- *Dataset curation*
- Entrenamiento del modelo con el dataset refinado
- Evaluación de resultados
- Publicación del modelo para su uso en inferencia

A continuación, se evaluarán alternativas al procedimiento de training realizado con la ResNet34.

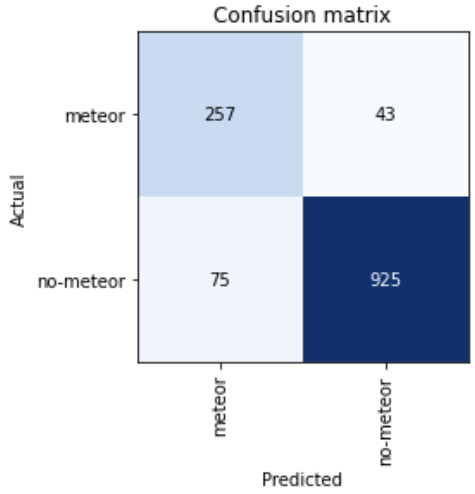
3.3.5 Creación del modelo con ResNet18

Adicionalmente al procedimiento de training realizado con la ResNet34, se cree conveniente evaluar el mismo procedimiento de training, con exactamente el mismo dataset, pero sobre una ResNet18.
La arquitectura de una ResNet18 es muy parecida a la ResNet34, con la única diferencia del numero de capas convolucionales que las componen. La ResNet18 contiene 18 capas y por ello menor capacidad.

Evaluar el funcionamiento de una versión reducida de red, puede ser interesante pensando en la inferencia.
El proceso de entrenamiento lo podemos realizar de forma separada, con recursos de GPU disponibles y sin restricciones de tiempos de respuesta. Pero la inferencia tiene unos requerimientos más estrictos. Para inferir, necesitamos pre-procesar la imagen a analizar y realizar una inferencia en nuestra red neuronal. Este proceso, juntamente a notificaciones en caso de detección, guardar logs, etc. se debe realizar en menos de 30 segundos, puesto que queremos que nuestro sistema de inferencia será capaz de trabajar en tiempo real a medida que las imágenes se van generando. Sumado a este requerimiento, cabe destacar que la idea es que la inferencia se realice lo mas cerca de la fuente de imágenes, idealmente, directamente en la Raspberry, o en un PC en la misma red local que la Raspberry.
Para ello, evaluar alternativas de menor tamaño, como la ResNet18, puede ser interesante. Si la ResNet18 mantiene un rendimiento parecido, podría ser una alternativa para entornos de inferencia con menor capacidad computacional en los que se agradecerá un tiempo de inferencia menor con una red menor.

La totalidad de las 35 *epochs* del entrenamiento de la ResNet18, así como la gráfica de *loss*, se puede consultar en el *Anexo 2: Proceso de entrenamiento de la ResNet18*
A continuación, se muestran las métricas y la matriz de confusión sobre el dataset de test.

	precision	recall	f1-score	support
meteor	0.77	0.86	0.81	300
no-meteor	0.96	0.93	0.94	1000
accuracy			0.91	1300
macro avg	0.86	0.89	0.88	1300
weighted avg	0.91	0.91	0.91	1300



Como se puede ver, el rendimiento de la ResNet18 es algo menor que la ResNet34. Las métrica que hemos indicado como clave en el proyecto y la que queremos maximizar, el *recall* de la clase meteoro, cae de un 98% a un 86%.
La precisión del modelo con la ResNet18 es menor, en general y específicamente para ambas clases.
De todas maneras, el *recall* del 86% sobre meteoros no es un mal valor. Si volvemos al ejemplo usado en el punto anterior y asumimos una noche donde pueda haber unos 30 meteoros, usando este modelo, detectaríamos aproximadamente 26.

3.4 Evaluación de los modelos y análisis de resultados

Claramente nos decantamos por usar una ResNet34, y lo hacemos por el porcentaje de *recall* sobre la clase meteoro, de un 98%. Esta cifra, tal y como se ha comentado, nos permitirá detectar una gran cantidad de meteoros y seguir evolucionando el modelo con más ejemplos positivos.

Se descarta seguir iterando con arquitecturas más complejas de ResNet, puesto que se considera que el rendimiento proporcionado por la ResNet34 es suficientemente bueno.

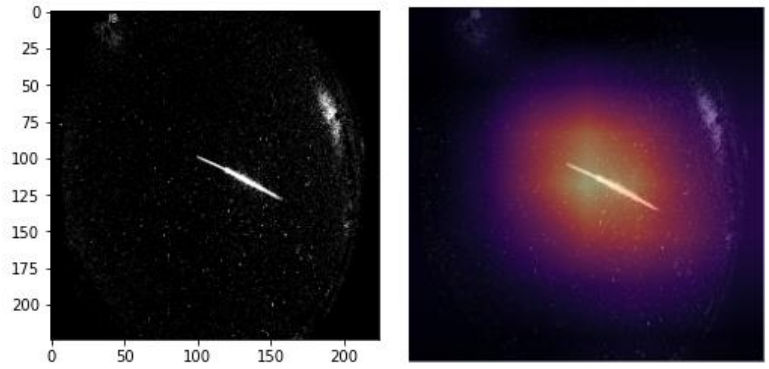
Se confirma que la técnica de *transfer learning* da buenos resultados. Esta parte es interesante comentarla porqué vale la pena remarcar que la ResNet34 ha estado inicialmente entrenada con la base de datos de imágenes imagenet, que esta formada por imágenes totalmente generalistas, sin temática concreta.

Parece claro que las CNN son capaces de abstraer conceptos genéricos de las imágenes, y que estos conceptos, son luego aplicables para entrenar modelos con imágenes que no tienen mucho o nada que ver con las de imagenet.

Mediante validar la performance del modelo en un dataset específico de test (con imágenes de días distintos a los usados para train), hemos podido descartar que se produzca *overfitting* en el entrenamiento y que el modelo es capaz de generalizar, pero vale la pena asegurar que el modelo cuando detecta un meteoro lo hace por las razones que queremos.
Las CAM [37](class activation maps), fueron introducidas por Bolei Zhou et al. en "*Learning Deep Features for Discriminative Localitzation*" [8]. El concepto consiste en usar la salida de la ultima capa convolucional juntamente con las predicciones para generar un *heatmap* (mapa de calor) que permita entender porqué el modelo ha tomado la decisión. Podemos sobreponer a la imagen que estamos proporcionando al modelo, una representación de las neuronas que se activan para establecer que la imagen pertenece o no a una clase. De esta manera podemos comprobar visualmente las regiones de la imagen que generan activaciones de neuronas de la CNN que provocan que la imagen se clasifique como meteoro.

El código Python para la generación de los *heatmaps* se puede consultar en los archivos: *Training_process_example.jpynb, Train_resnet34.html* o *Train_resnet18.html*, todos ellos disponibles en el repositorio Github del trabajo. El código usado para implementar las CAM en el trabajo se extrae de la documentación de Fast.ai

A modo de ejemplo, se aportan análisis de *heatmaps* realizados sobre imágenes que se pretenden evaluar.



[37] https://github.com/fastai/fastbook/blob/master/18_CAM.ipynb

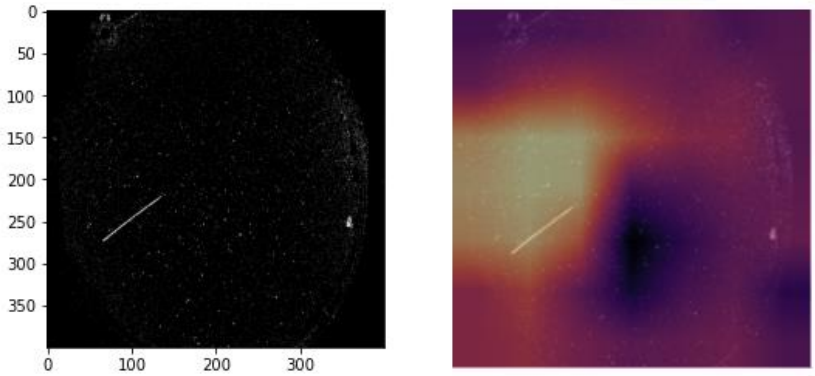


Figura 26: Análisis de heatmap sobre imágenes de meteoros
Fuente: Elaboración propia

Como se puede observar, la detección de los meteoros se realiza en base a las regiones de las imágenes en las que aparece una estela, con lo que parece claro que la detección se hace por los criterios correctos.
De todas maneras, las imágenes contienen pocos elementos, son mayoritariamente negras, con lo que vale la pena asegurar este aspecto buscando imágenes con más carga visual.

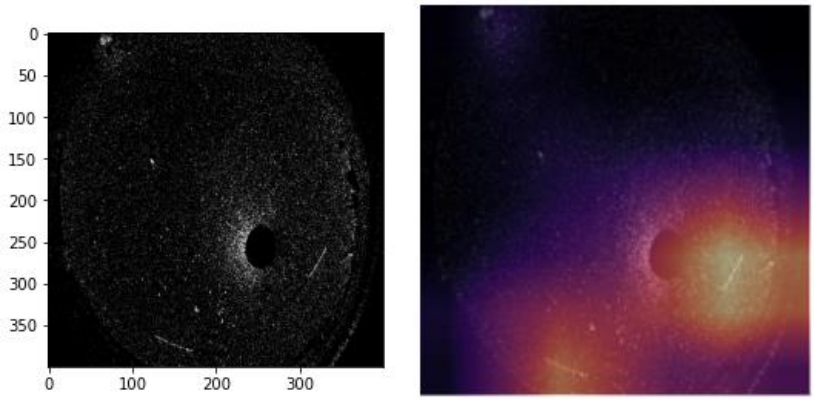


Figura 27: Análisis de heatmap con dos meteoros
Fuente: Elaboración propia

En este caso, tenemos dos meteoros y podemos ver como el modelo clasifica esta imagen como meteoro en base a las dos regiones que nos muestra el *heatmap* y que coinciden con ambos meteoros.

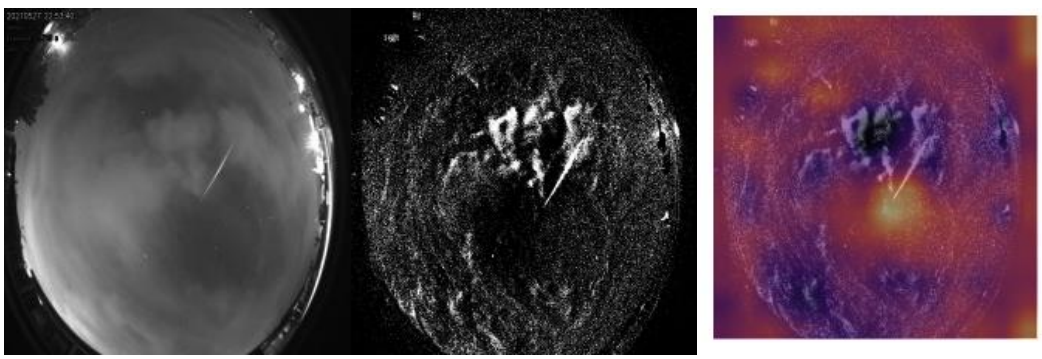


Figura 28: Análisis de heatmap con imágen con presencia de nubes
Fuente: Elaboración propia

En la figura anterior, podemos ver como se detecta un meteoro en una imagen con presencia de nubes. La regiones de activación son algo más dispersas, pero si buscamos la región de activación más intensa, podemos ver como se corresponde con la estela del meteoro.

Otro tema que vale la pena comentar es que el modelo, tiene capacidad de generalizar. Se han realizado pruebas proporcionando directamente al modelo imágenes sin restar, simplemente convertidas a gris y redimensionadas a 400x400. El modelo, que ha sido entrenado mediante imágenes restadas, es capaz, en menor medida, de detectar meteoros y clasificar las imágenes correctamente.

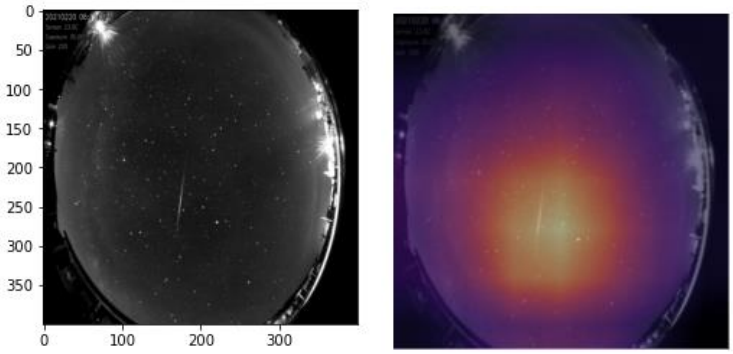


Figura 29: Análisis de heatmap con imágen sin restar
Fuente: Elaboración propia

En general, el modelo clasifica correctamente y lo hace en base a criterios que a priori parecen correctos. Aunque cabe remarcar, que como ya se ha comentado en otros puntos de esta memoria, uno de los hándicaps del trabajo consistía en partir con menos de 20 casos positivos. Durante el proceso de etiquetado, se vio que se disponía de más casos positivos, pero siempre y cuando se considerara un caso positivo todo aquello que fuese una estela en el cielo, asumiendo que lo que detectábamos podía ser perfectamente la estela dejada por un satélite artificial. De esta manera, se obtuvo un conjunto suficiente de ejemplos que, permitieron entrenar el modelo.

Por lo tanto, a modo de conclusiones del apartado de Desarrollo del proceso analítico, vale la pena remarcar que la ResNet34 y *transfer learning* son buenas opciones para crear un modelo de detección de meteoros, aunque la falta de ejemplos positivos, en nuestro caso, nos ha llevado a desarrollar un modelo algo más genérico que es capaz de detectar estelas en el cielo, pudiendo ser estas causadas por satélites artificiales.
Este aspecto se comentará en el apartado de evaluación de resultados de la prueba de concepto, puesto que existen procedimientos para reducir drásticamente el numero de satélites artificiales detectados, usando el modelo desarrollado.

3.5 Implementación del *Pipeline*

Una vez desarrollado el modelo de detección de meteoros, el siguiente paso consistió en desarrollar las herramientas requeridas para implementar un *pipeline* de detección.
La idea detrás del *pipeline* consiste en que cualquier observatorio que disponga del software *open source* AllSky, o cualquier software que permita capturar imágenes de 180º del cielo nocturno, pueda instalar un software en su red local. Dicho software va a acceder a las imágenes generadas, va a realizar la inferencia y notificará de forma centralizada las detecciones realizadas.
Para ello, es requerido una capa *cloud* y un software para inferencia.

3.5.1 Desarrollo de capa cloud AWS *serverless*

Para la implementación de la capa *cloud*, se elige Amazon Web Services. La elección se base en los siguientes motivos:

- Costes. Mediante AWS y desarrollando la arquitectura *cloud* siguiendo patrones *serverless*, el coste resultante es prácticamente nulo
- Se dispone de una cuenta en AWS previamente creada.
- Se tiene conocimiento de los servicios AWS requeridos para el trabajo.

Las funcionalidades requeridas para la plataforma *cloud* son:

- API para notificar actividad y detecciones de los distintos observatorios que usen el sistema.
    - La API debe ser gestionada y no requerir de una instancia (servidor) dedicada
- Capacidad de almacenaje barata y elástica, sin necesidad de aprovisionar una cantidad de almacenaje previa para el trabajo.

En base a estos requerimientos, se establece que la arquitectura a desplegar en AWS es:

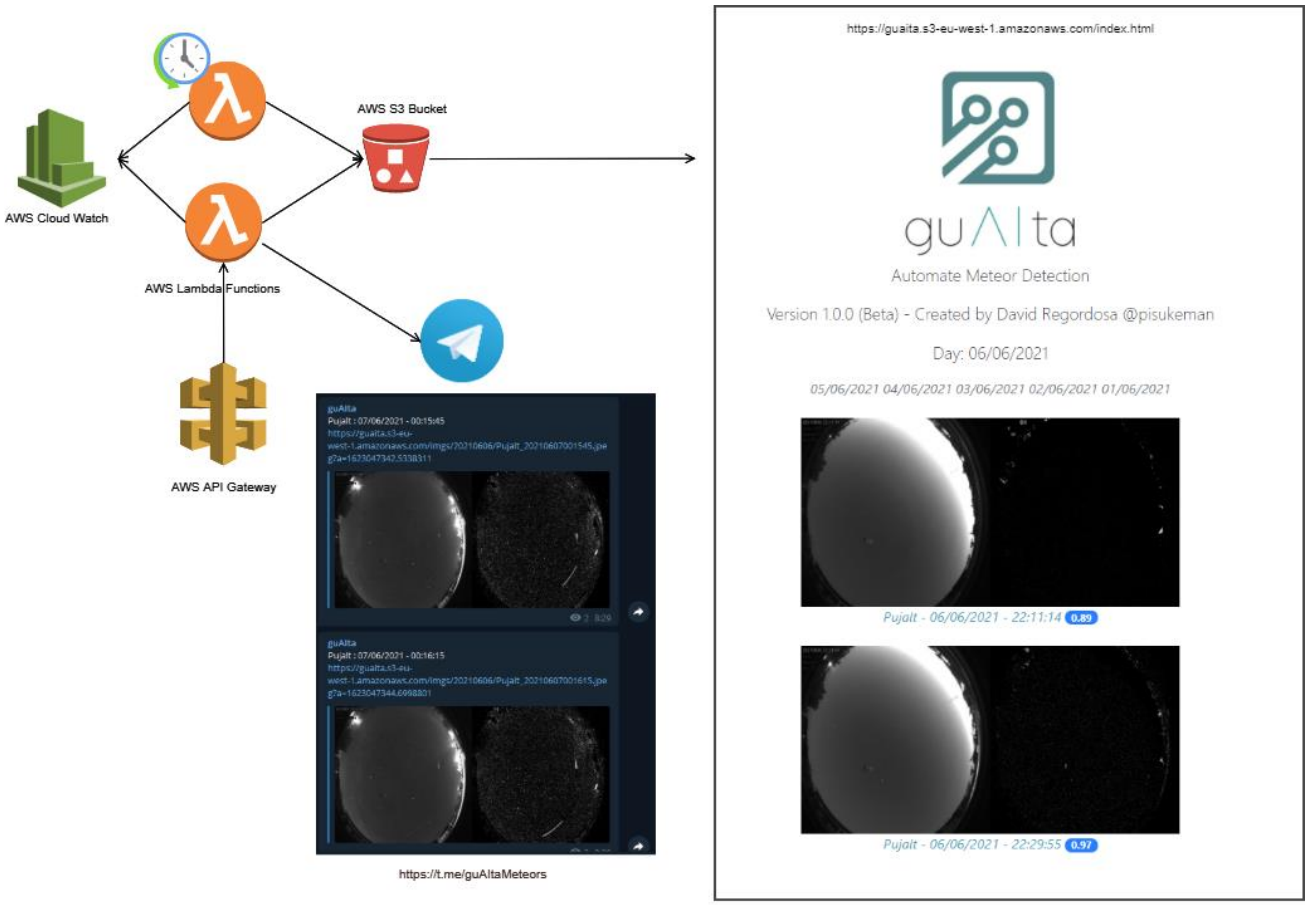


Figura 30: Arquitectura AWS del pipeline
Fuente: Elaboración propia

3.5.1.1 AWS API Gateway

El primer servicio que desarrollar en AWS consiste en una pequeña API que permita a la aplicación cliente informar sobre el proceso de inferencia.
Para ello se usó el servicio AWS API Gateway. Mediante API Gateway, AWS nos genera un *endpoint* al que podemos enviar peticiones HTTPS POST.
En nuestro caso, el *endpoint* generado es del estilo:
https://xxxxxxxxxx.execute-api.eu-west-1.amazonaws.com/Prod/uploadmeteor

Mediante este único *endpoint*, las aplicaciones clientes podrán enviar un JSON en el que informen de:

- Se inicia el proceso de escaneo de imágenes (solo si se ejecuta el proceso en *batch* no en tiempo real)
- Se ha detectado un meteoro
- Finaliza el proceso de escaneo de imágenes (solo si se ejecuta el proceso en *batch* no en tiempo real)

Para indicar que se inicia el proceso de escaneo de imágenes, la aplicación cliente envía el siguiente JSON:

```
{
   "log":"start",
   "obs_id":"xxxx"
}
```


En el JSON, solamente se pasan dos elementos, el elemento log con el valor '*start*' y el *obs_id* que contiene el identificador del observatorio. El valor de *obs_id* es un campo alfanumérico que el administrador del sistema proporciona a cada observatorio.

Para indicar que finaliza el proceso de escaneo de imágenes, la aplicación cliente envía el siguiente JSON:


```
{
   "log":"end",
   "obs_id":"xxxx",
   "total":1200
}
```


A diferencia del anterior, contiene el valor '*end'* en el elemento *log* y adicionalmente se envía un campo total con el número total de imágenes procesadas.

Tanto el envío del *start* como el *end* van vinculados a una ejecución en *batch* de las imágenes de una noche. Como ya se ha comentado, la aplicación cliente puede estar escaneando en tiempo real en búsqueda de imágenes que se generen o puede realizar un análisis en *batch* de todas las imágenes generadas durante una noche.

Tanto en *batch* como en tiempo real, si se produce una detección, la aplicación cliente envía el siguiente JSON:


```
{
   "id":"20210606230537",
   "date":"06/06/2021",
   "time":"23:05:37",
   "score":"0.998887564",
   "telegram":"True",
   "obs_id":"xxxxx",
   "processed":"07/06/2021 08:22",
   "folder":"20210511",
   "img":"/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQE
}
```


Este JSON, contiene los siguientes campos:

- *Id*: identificador de la imagen. En nuestro caso, AllSky genera las imágenes con el año, mes, día y hora concatenadas. Este es nuestro identificador único de la imagen para cada observatorio.
- *Date*: Fecha de captación de la imagen
- *Time*: Hora de captación de la imagen
- *Score*: Valor del *scoring* que ha generado el modelo para clasificar esta imagen como meteoro
- Telegram: parámetro que se configura en el cliente y que indica si queremos que esta detección se publique en el canal de Telegram del proyecto
- *Obs_id*: Identificador del observatorio

- *Processed*: Fecha y hora en que se ha realizado la inferencia
- Folder: Carpeta que se está analizando. AllSky genera una carpeta para cada día
- *Img*: Imagen codificada en base64. Esta imagen es la concatenación de la imagen original y la imagen que se ha procesado. Como ya se ha comentado, el modelo trabaja aplicando ciertas transformaciones a las imágenes, con lo que cuando se produce una detección de meteoro, se opta por notificar tanto la imagen original como la que ha procesado el modelo. Esta parte se detallará en el apartado de *Desarrollo de herramientas para inferencia*

Cuando API Gateway recibe una llamada HTTPs POST, la reenvía a una función Lambda AWS.

3.5.1.2 Funciones Lambda AWS

Las funciones Lambda son funciones Python (en nuestro caso), que se ejecutan en la nube bajo demanda. Las funciones lambda no residen en ningún servidor específico, AWS levanta los recursos de memoria y CPU requeridos para su ejecución cada vez que es requerida.
En nuestro caso, se desarrollaron dos funciones lambda, una para gestionar las peticiones que nos llegan a través de la API, y otra para generar un resumen diario de las detecciones realizadas en forma de HTML.
Estas dos funciones lambda son *guAlta* para la API y *generateWebsite* para la generación del resumen diario.

3.5.1.2.1 Lambda guAlta

Esta primera lambda va vinculada a API Gateway, de manera que, cada vez que la API recibe una llamada POST, AWS levanta la lambda y le pasa por parámetro la llamada HTTP para que la procese.
El código fuente de esta función lambda está disponible en el repositorio Github del proyecto.
Las funcionalidades que ejecuta esta lambda son:

- Al recibir una petición POST de la aplicación cliente, la analiza y detecta si se trata de una llamada de log (informando del inicio/fin de un proceso *batch*), o bien si se trata de una detección de un meteoro.
    - En caso de ser una llamada de log, la aplicación lambda ejecuta a su vez una llamada HTTP hacia la API de Telegram para publicar que se ha iniciado o acabado el procesado de imágenes. Esta parte se detallará en el apartado de *Desarrollo de herramientas para el operador*
- Si la llamada notifica una detección de un meteoro, se realizan las siguientes acciones:
    - Se almacena en un *Bucket* S3 de AWS, dentro de la carpeta /detections y dentro de la carpeta del día que se está analizando, el JSON recibido. El JSON se almacena con un nombre de archivo creado a partir de concatenar el id de observatorio y el id de la imagen.

- Se almacena en el mismo *Bucket* S3 de AWS, dentro de la carpeta /imgs y dentro de la carpeta del día que se está analizando, la imagen que se ha recibido. La imagen se almacena con un nombre de archivo creado a partir de concatenar el id de observatorio y el id de la imagen.
- Si en el JSON que se ha recibido, el parámetro Telegram venia a True, se notifica la detección al canal de Telegram del proyecto, pasando información de la detección y un link a la imagen en el *Bucket* S3

Mediante esta operativa, la función lambda, ordena las detecciones dentro de las carpetas correspondientes en el *Bucket* S3 y, si es requerido, notifica al canal de Telegram de la detección realizada.
La URL del canal de Telegram es:

- https://t.me/guAltaMeteors

3.5.1.2.1 Lambda generateWebsite

Esta lambda, al igual que la anterior, se ejecuta bajo demanda.
En este caso, el desencadenante que provoca que se ejecute esta lambda es un evento configurable de AWS CloudWatch Events[38].
Mediante el evento de CloudWatch Events, cada día a las 9 de la mañana, AWS ejecuta esta lambda y se realizan las siguientes funcionalidades:

- La lambda accede al *Bucket* S3 y específicamente a la carpeta /detections y selecciona todos los JSONs de las detecciones de la noche anterior.
- Mediante estos JSONs, la lambda crea un archivo HTML estático en el que muestra todas las detecciones de la noche anterior con sus imágenes asociadas.
- La lambda guarda este HTML en el *Bucket* S3, con el nombre equivalente al año, mes y día de la noche anterior y también genera una copia con el nombre *index.html*
  - De esta manera, se puede acceder a la web de detecciones mediante acceder al fichero *index.html* del *Bucket* S3
- La Lambda adicionalmente, realiza una llamada a la API de https://api.n2yo.com[39]. Este servicio gratuito, permite consultar los tránsitos de distintos satélites artificiales. En nuestro caso, peticionamos a la API de n2yo por los tránsitos de la estación espacial internacional, muy visibles durante la noche.
  - La petición a la API de n2yo permite especificar una latitud y longitud del observatorio y un número de días, y nos devuelve un JSON con información de los tránsitos de los satélites artificiales peticionados. El JSON resultante lo almacenamos también en S3 indicando que son los tránsitos previstos para la noche siguiente y para el observatorio x.
  - La Lambda, al crear el HTML con el resumen de la noche anterior, usa el JSON almacenado en S3 para dejar informado en el HTML los tránsitos de la estación espacial internacional. De esta manera

[38] https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/WhatIsCloudWatchEvents.html
[39] https://api.n2yo.com

al visualizar el HTML se pueden ver las detecciones de la noche anterior y adicionalmente información sobre los tránsitos de la ISS para los distintos observatorios, y ayudar así al operador a identificar falsos positivos.

La URL de la web generada cada día a las 9 de la mañana es:

- https://guaita.s3-eu-west-1.amazonaws.com/index.html

Mediante esta estrategia, un HTML estático se genera automáticamente cada mañana a las 9 y se almacena en S3.
Dado que el *Bucket* S3 es de acceso público, la web se puede acceder directamente consultando el *Bucket*.
De esta manera tenemos una web auto-generada con el resumen de las detecciones efectuadas, que no necesita servidor de hosting alguno.

3.5.1.3 Bucket S3

Como se ha comentado en los puntos anteriores, el almacenaje de las detecciones efectuadas se realiza en un *Bucket* S3 de AWS.
S3 nos permite almacenar cualquier tipo de archivo en una estructura de carpetas a un coste bajo.
En nuestro caso, la estructura del *Bucket* S3 se compone de las siguientes carpetas y elementos:

- Raíz: En la Raíz del *Bucket* S3 tenemos los siguientes archivos/carpetas:
  - *Index.html*: Web con el resumen de las detecciones realizadas el último día analizado
  - *yyyymmdd.html*: Un archivo HTML resumiendo las detecciones de cada uno de los días analizados.(para poder visualizar un histórico)
  - *Obs_id_iss.json*: Un archivo para cada observatorio con información de los transitos visibles de la ISS, extraída de n2yo.
  - Carpeta */Detections* con las deteciones realizadas
    - Sub-Carpeta */yyyymmdd*: se genera una subcarpeta para cada día
      - *Obs_id_yyyymmddhhmmss.json*: Archivos JSON con cada una de las detecciones efectuadas para ese día.
  - Carpeta */Imgs* con las imágenes de las detecciones realizadas
    - Sub-Carpeta */yyyymmdd*: se genera una subcarpeta para cada día
      - *Obs_id_yyyymmddhhmmss.jpeg*: Archivos JPEG con cada una de las imágenes de detecciones efectuadas para ese día.
  - Carpeta */Models*:
    - *guAIta_latest_version.pkl*: Archivo pkl con la última versión del modelo disponible. El software cliente se descarga este fichero al arrancar.

De esta manera, S3 se convierte en el único repositorio donde almacenamos toda la información del proyecto, ya sean los JSON con información sobre las detecciones, las imágenes, el modelo, y, además, un conjunto de HTML que permiten revisar todas las detecciones realizadas mediante el navegador.

3.5.1.4 Coste

En todo desarrollo en la nube, es importante tener en cuenta el coste.
En nuestro caso, el diseño de la infraestructura *cloud* se ha realizado mediante patrones que minimizan el coste.
El uso de tecnología *serverless* con las lambdas AWS, permite crear una API y un *backend* simple, con un coste extremadamente bajo.
El único servicio que es susceptible de generar coste es el almacenamiento en el Bucket S3.
Se aporta un detalle del coste de la plataforma *cloud* en AWS durante los meses de mayo y junio del 2021, conteniendo junio una parte estimada por no estar el mes finalizado. Durante los meses de mayo y junio se ejecutó la prueba de concepto en el Observatori de Pujalt (ver apartado *Despliegue de Prueba de Concepto*), con lo que el coste es real y aplicable a un solo observatorio.
Como se puede ver en la tabla siguiente, el único servicio que genera coste es S3, aunque el coste es prácticamente nulo.

Service	May 2021	Jun 2021*	Service Total
Total cost ($)	0.01	0.01	0.02
S3 ($)	0.01	0.01	0.02
API Gateway ($)	0.00	0.00	0.00
Lambda ($)	0.00	0.00	0.00

Tabla 5: Desglose del coste por Servicio de AWS

Se realiza a continuación una estimación del coste aproximado de la plataforma simulando un uso más elevado. Para ser exactos se ha estimado el coste en base a que se generen 10.000 detecciones de meteoros por mes :

- AWS S3
  - 10 GB mensuales de almacenaje en S3
  - 10.000 PUT HTTP mensuales sobre S3 (simulando que se detectan 10.000 meteoros al mes)
  - 10.000 GET HTTP mensuales sobre S3
  - 10 GB de información transmitida desde S3
- AWS Lambda
  - 10.000 ejecuciones de la lambda que se invoca al producirse una detección
    - 805 ms de duración de la ejecución de la lambda (se ha calculado el tiempo medio de ejecución)
    - 128 MB de memoria requerida para la ejecución de la lambda
- API Gateway
  - 10.000 ejecuciones de la API

El coste de esta infraestructura sería de 1,18 USD mensuales.
Se aporta un enlace[40] a la estimación del coste en AWS detallado por servicio.

[40] https://calculator.aws/#/estimate?id=06b2d94e4b716c343ddbfee8d3cc29594172b4ab

A tener en cuenta que, de los 1,18 USD mensuales, 0,81 USD corresponden al *Bucket* S3.

3.5.2 Desarrollo de herramientas para inferencia

Por lo que se refiere a la parte cliente, es requerido desarrollar un software que permita inferir sobre las imágenes generadas y reportar en caso de detección de meteoro.
Todo lo referente a comunicar una detección, ya se ha explicado en el apartado anterior, en la parte de API, detallando los JSON que se usan para informar de una detección, así como para indicar que se ha arrancado/parado el proceso de inferencia en *batch*.

La herramienta cliente se ha desarrollado en Python, y el código está disponible en el repositorio Github del proyecto.
Todas las funcionalidades del software de inferencia se han desarrollado dentro del fuente *guAIta.py*
El software para inferencia consta de dos funcionalidades, *guAItaDayAnalisis*, que permite la inferencia de una noche entera en *batch*, y *guAItaRunner*, que realiza la inferencia de las imágenes que se van generando en tiempo real.

3.5.2.1 Inferencia en batch

La inferencia en *batch* se realiza mediante la ejecución de la función *guAItaDayAnalisis*.
Esta función está preparada para realizar las siguientes tareas:

- Revisar si se dispone del modelo descargado en local. En caso de no disponer del modelo, se procede a descargarlo desde el *Bucket* S3
- En caso de que el sistema esté configurado para enviar información a Telegram, se realiza un llamada a la API de AWS para indicar que arranca el proceso de inferencia de ese observatorio.
    - A continuación, una Lambda procesa el mensaje y realiza una llamada a la API de Telegram, que desencadena que en el canal de Telegram del proyecto aparezca el mensaje indicando que el proceso de escaneado en *batch* ha arrancado
- Detectar cual es la última carpeta que se ha generado dentro del directorio de generación de imágenes.
    - Como se ha comentado, el software AllSky genera una carpeta para cada día y en ella va guardando las imágenes que captura.
- Generar un listado de todas las imágenes disponibles en la última carpeta generada (las imágenes más recientes).
- Para cada imagen:
    - Aplicar las transformaciones detalladas en el apartado *Pre-procesado de las imágenes:* conversión a gris, restado con la imagen anterior, *threshold* y *resize*. (todas las transformaciones se aplican mediante librerías de OpenCV)
    - Realizar la inferencia mediante el método *predict* [41] del modelo.

[41] https://docs.fast.ai/learner.html#Learner.predict

- *Predict* nos responde con una clase (meteor, no-meteor) y un scoring para cada una de las dos clases.
- Al resultado de *Predict,* en caso de que la clase resultante sea meteor, le aplicamos un *threshold* configurable. De esta manera el operador tiene un mecanismo para determinar a partir de que *scoring*, considera que se quiere notificar una detección de un meteoro.

  Este punto es crucial y merece la pena detallarlo.

  Sin este *threshold*, cualquier imagen que el modelo considere que es un meteoro sería notificada. Esto implicaría que una imagen clasificada como meteoro con un 99% de certeza (*scoring* 0.99) se trataría de igual manera que una imagen clasificada como meteoro con un 51% de certeza (*scoring* 0.51)

  Mediante este *threshold*, el operador puede ajustar el funcionamiento del software de detección. Si lo que quiere el operador es un modelo que minimice los falsos positivos, entonces puede ajustar el valor del *threshold* al alza.
- En caso de tener una detección se procede de la siguiente manera:
  - Se crea un JSON con la información de la detección (detallada en el apartado de *API Gateway*)
  - Se crea una imagen con el resultado de concatenar la imagen original en gris y la imagen después de la transformación aplicada. Se sitúa una imagen al lado de la otra y se genera una única imagen de escala de grises de 448 x 224
    - Se codifica la imagen a Base64 y se añade al JSON
  - Se envía el JSON con la detección a la API creada en AWS

- Al finalizar las imágenes disponibles, si el sistema está configurado para enviar información a Telegram, se realiza un llamada a la API de AWS para indicar que el proceso de inferencia de ese observatorio ha finalizado.
  - Una Lambda notifica a la API de Telegram de igual manera que para el mensaje de arrancada del proceso.

3.5.2.2 Inferencia en tiempo real

La inferencia en tiempo real se realiza mediante la ejecución de la función *guAItaRunner*.

Esta función está preparada para realizar las siguientes tareas (se omite detallar las funcionalidades que ya se han detallado en el apartado anterior):

- Revisar si se dispone del modelo descargado en local. En caso de no disponer del modelo, se procede a descargarlo desde el *Bucket* S3
- Se arranca un bucle infinito. En cada iteración del bucle se realiza:
  - Se revisa cual es la última carpeta creada por AllSky. Esta comprobación se hace al arrancar el proceso y cada 300 segundos.
  - Se revisan dos parámetros de configuración que indican la hora de inicio del proceso y la hora de finalización.
    - Estos dos parámetros permiten indicar al proceso que funcione solo de 20:00 de la noche a 07:00 de la mañana,

por ejemplo, y que quede en estado dormido mientras no está dentro de esa franja horaria.
    - Se genera un listado de las imágenes disponibles en la carpeta destino.
    - Si la última imagen disponible no se ha procesado aún, se procesa:
        - Realizamos el pre-procesado sobre la imagen
        - Realizamos la inferencia sobre la imagen resultado del pre-procesado
    - En caso de que la clase resultante de la inferencia sea meteor, le aplicamos un *threshold* configurable y determinamos si se debe notificar.
        - En caso de notificación se genera el JSON (de la misma manera que se ha documentado en el apartado anterior) y se envía a la API de AWS.
    - Se procede a realizar un *sleep* durante un numero de segundos configurable. De esta manera el proceso deja de consumir recursos durante ese tiempo.
        - A destacar que AllSky genera imágenes cada 30 segundos, con lo que, al acabar la inferencia de una imagen, el operador puede indicar que se aplique un tiempo prudencial de x segundos en el que el proceso dormirá.

Como se ha comentado en este apartado y en el anterior, los dos procesos disponibles para inferencia requieren de unos parámetros configurables.
Estos parámetros se detallan en el siguiente apartado.

3.5.2.3 Configuración del sistema de inferencia

En el fichero *guAItaConfig.py*, el operador dispone de una serie de parámetros que le permiten configurar el funcionamiento de los dos procesos de inferencia. Estos parámetros son:

- *guAIta_main_folder*. Carpeta en la que trabaja el software de captación de imágenes. Esta es la carpeta en la que el software de inferencia se posiciona para escanear en búsqueda de cada una de las carpetas diarias que se crean.
    - Esta carpeta puede ser del propio dispositivo de inferencia o bien una carpeta compartida en red local.
- *guAIta_start_time*. *String* que permite indicar la hora de inicio del escaneo en tiempo real. Ejemplo: “2000” para indicar las 20:00 de la noche.
- *guAIta_end_time*. *String* que permite indicar la hora de finalización del escaneo en tiempo real. Ejemplo: “0800” para indicar las 08:00 de la mañana.
- *guAIta_obs_id*. *String* que permite indicar el ID de observatorio. Este valor debería ser único para cada observatorio, permitiendo así identificar el origen de las notificaciones de meteoros.
- *guAIta_URL*. URL para acceder a la API en AWS
- *guAIta_enable_telegram*. Parámetro que permite al operador indicar si quiere que las detecciones del observatorio se publiquen en el canal publico de Telegram

- *guAIta_enable_extra_log*. Parámetro que permite al operador indicar si quiere que el proceso de inferencia genere una línea en pantalla para cada imagen inferida o solo para las detecciones positivas.
- *guAIta_threshold*. Parámetro que permite al operador indicar el *threshold* a aplicar al *scoring* de una imagen clasificada como meteoro para notificarla. Ejemplo: 0.85
- *guAIta_sleep_between_imgs*. Parámetro que permite al operador indicar los segundos de *sleep* al finalizar una inferencia. Ejemplo: 10

Se aporta a continuación una captura de la ejecución del proceso de inferencia, donde se pueden ver imágenes clasificadas como no-meteor, imágenes clasificadas como meteor pero no notificadas como *Detection* porqué no superan el *threshold* de 0.85 e imágenes clasificadas como meteor que se convierten en *Detections* porqué su *scoring* supera el 0.85.

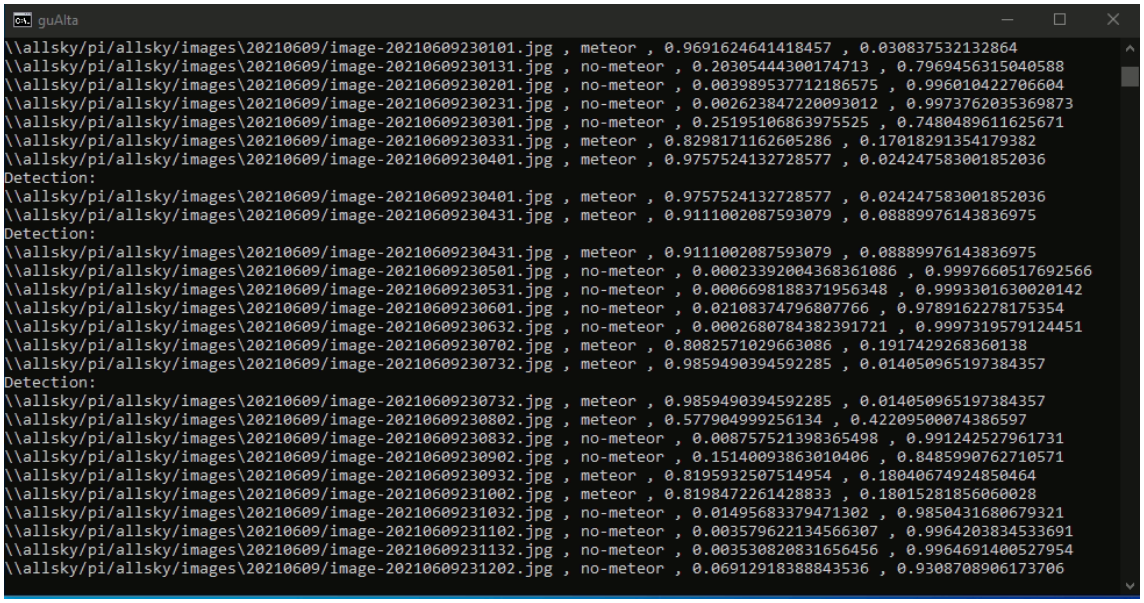


Figura 31: Ejecución del proceso de inferencia
Fuente: Elaboración propia

3.5.3 Desarrollo de herramientas para el operador

El último elemento del *pipeline* para detección de meteoros consiste en dos herramientas que permitan al operador visualizar el proceso de detección.
Estas dos herramientas, son, la pagina web [42]calculada a diario y el canal publico de Telegram[43].
La página web como ya se ha comentado, se genera automáticamente a diario, en base a un evento programado en AWS y contiene la información de las detecciones de la noche anterior.

[42] https://guaita.s3-eu-west-1.amazonaws.com/index.html
[43] https://t.me/guAItaMeteors

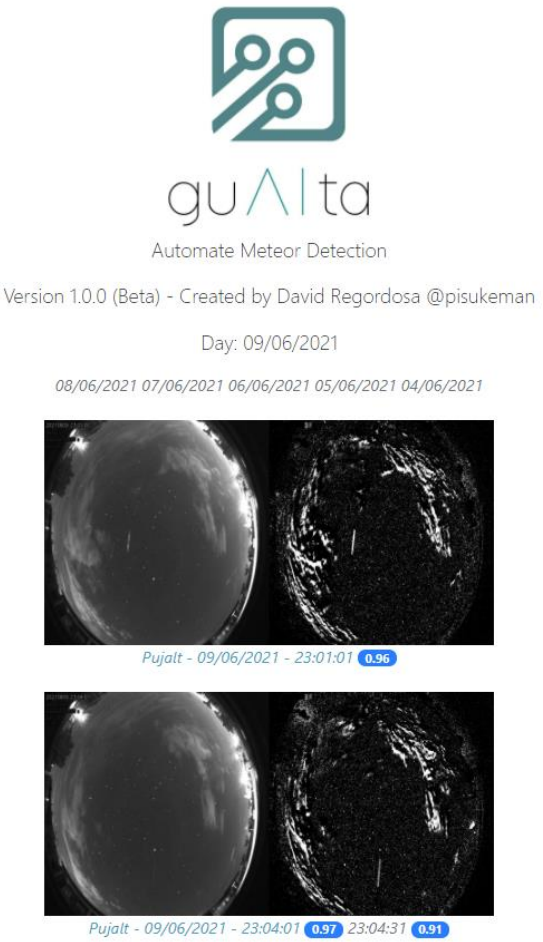


Figura 32: Web con las detecciones diarias
Fuente: Elaboración propia

Las funcionalidades de la web son:

- Muestra las detecciones de la noche anterior, mediante una fotografía donde se puede ver la imagen original y la procesada. Se muestra el nombre del observatorio, la fecha, la hora y el *scoring.* Al *clickar* se proporciona acceso a la imagen
- Adicionalmente, si se han producido dos o más detecciones consecutivas, las muestra agrupadas. De esta manera el operador puede descartar fácilmente falsos positivos producidos por el transito de un satélite artificial (es mucho más lento que un meteoro y por lo tanto deja trazas en varias fotografías consecutivas). Dos detecciones consecutivas se pueden observar en la segunda detección de la imagen anterior. Notar que aparecen dos detecciones separadas entre sí 30 segundos.
- Acceso a los días anteriores. La web es un HTML estático que genera una Lambda AWS cada día. La lambda genera el HTML con el nombre específico del día que está analizando. De esta manera, podemos acceder a días anteriores.
- Información para cada observatorio de los tránsitos visibles de la ISS. (esta información se muestra en el *footer* de la página)

La otra herramienta a disposición del operador es el canal de Telegram.

El canal de Telegram muestra las detecciones realizadas, con información sobre el Observatorio, fecha y hora.

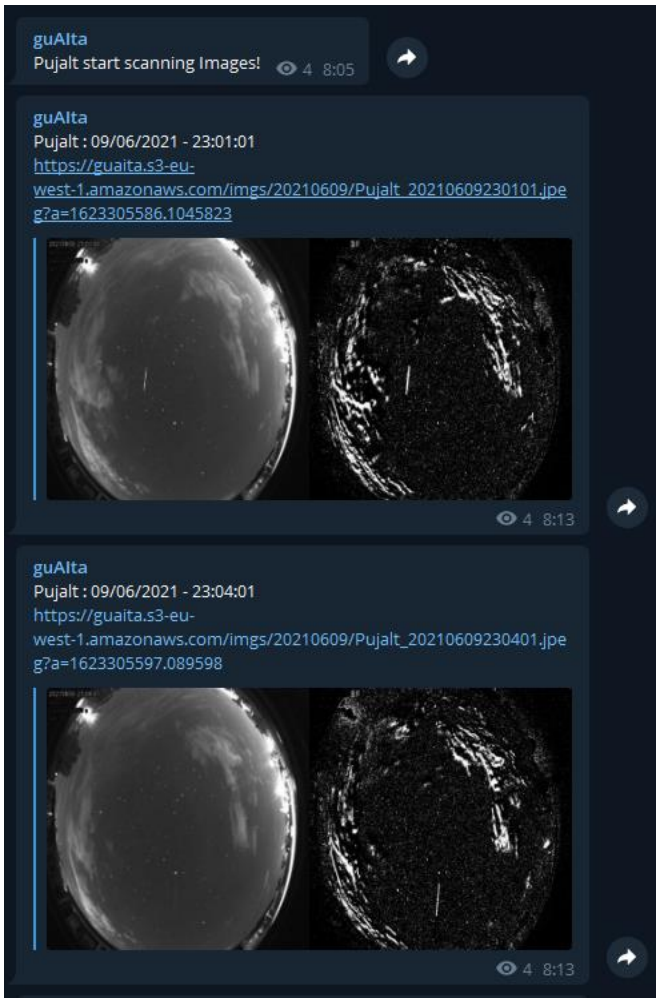


Figura 33: Canal de Telegram
Fuente: Elaboración propia

El canal de Telegram contiene dentro de sus *subscribers* un *bot* creado con permisos de administrador. Mediante la API de Telegram, las funciones Lambda pueden peticionar al *bot* que publique mensajes en el chat público del canal.

A través del canal de Telegram y la web, el operador puede visualizar un resumen de la actividad de la noche.
Adicionalmente si los observatorios utilizan el software para inferencia en modalidad tiempo real, el canal de Telegram mostrará cada detección unos segundos después de que se produzca.

4. Despliegue de Prueba de Concepto

4.1 Despliegue solución en Observatori Pujalt

Durante los meses de mayo y junio del 2021 se realizó una prueba de concepto que consistió en desplegar el *pipeline* de detección de meteoros en el Observatori de Pujalt.
Dicho *pipeline* inferirá mediante el modelo explicado en el apartado de *Creación del modelo con ResNet34*
Para poder realizar la prueba de concepto, se desplegó la solución cloud detallada anteriormente y las herramientas para el operador. Estas dos partes se desplegaron de forma centralizada en la nube de AWS.
Por lo que a la herramienta de inferencia se refiere, se realizó una instalación en local en el Observatori de Pujalt, en un PC con Windows 10 que tenia acceso a la red local del observatorio.

El proceso de instalación del software de inferencia se realiza de forma muy rápida gracias al fichero *guAIta_conda_env.yml*, disponible en el repositorio Github del proyecto, que define exactamente las librerías requeridas.
El proceso de instalación del *pipeline* en el Observatori de Pujalt consistió en:

- Validar que desde el observatorio se accedía correctamente a las URLs de la API en AWS
- Instalar un servidor Samba en la Raspberry Pi que realiza la captura de imágenes mediante el software AllSky
- Validar que, desde el PC dedicado a la inferencia, se tenia acceso a la carpeta Samba de la Raspberry Pi, y que, por lo tanto, el PC de inferencia podría acceder a las imágenes que AllSky generaría
- Instalar el gestor de entornos Anaconda [44]para Windows 10 en el PC de inferencia
- Crear un entorno llamado guAItaEnvironment en Anaconda, importando el fichero *guAIta_conda_env.yml*
    - Anaconda está preparado para crear un entorno de cero mediante un fichero yaml
- Descargar el software para inferencia (disponible en el repositorio Github)
- Crear un archivo .bat en Windows que ejecute el siguiente script
    - *call C:\Anaconda3\Scripts\activate.bat guAItaEnvironment*
        - Esta línea accede a un script de Anaconda llamado actívate.bat el cual recibe por parámetro el nombre del entorno que se quiere activar.
    - *cd C:\Carpeta_software_inferencia*
        - Esta línea se sitúa en la carpeta donde hemos copiado el software para inferencia
    - *python guAIta.py*
        - Mediante esta línea ejecutamos el script Python de *guAIta.py* que por defecto está preparado para ejecutar la inferencia en *batch*

[44] https://www.anaconda.com/

- Editar el archivo de configuración guAltaConfig.py y rellenar los campos requeridos para que el software de inferencia funcione correctamente. Los más destacados son la dirección de la carpeta Samba de la Raspberry y el *threshold* a aplicar.

Una vez realizados estos pasos, al ejecutar el archivo .bat creado se inicializará un entorno Anaconda en el que estarán ya instalado el *runtime* de Python, así como todas las librerías necesarias y se ejecutará por defecto la inferencia en modo *batch*.
Una práctica interesante es configurar una tarea programada de Windows que ejecute el .bat creado a la hora que se considere oportuno.
Para instalaciones en Linux, el procedimiento, es muy parecido puesto que Anaconda también está disponible para Linux y la creación del entorno se haría con el mismo fichero yaml.

Específicamente para la prueba de concepto en el observatorio, los primeros días se realizó un proceso iterativo para afinar el valor del *threshold*. Finalmente, el operador estableció un *threshold* de 0.85. Este valor se estableció en base a minimizar los falsos positivos y validando que, durante esos días, con ese *threshold* no se produjeron falsos negativos.
Es importante remarcar que el proceso de afinación del *threshold*, debería ser un proceso de constante revisión. Dado que el modelo parte de un entrenamiento con pocos casos positivos tenemos que asumir que es probable que en ocasiones se produzcan detecciones de meteoros con un *scoring* inferior a 0.85 y que el sistema no los notifique. Estos falsos negativos deberían introducirse en el dataset de training para posteriores iteraciones del modelo, con el objetivo de mejorar el proceso de detección.

4.2 Evaluación de resultados

Para evaluar los resultados de la prueba de concepto, se analiza la actividad del *pipeline* instalado en Pujalt durante un periodo de 13 días, comprendidos entre el 30/05 y el 11/06. La evaluación se ha realizado analizando visualmente las imágenes de estos días y, por lo tanto, está sujeta a posibles errores de clasificación manual. Se debe tomar esta evaluación como un ejercicio para mostrar una orden de magnitud.
En la tabla siguiente se puede observar un registro por fecha de: el numero total de imágenes, el numero de detecciones generadas por el *pipeline* (con un *threshold* al 0.85), el número de verdaderos positivos, el número de falsos positivos, el porcentaje de falsos positivos respecto al total de imágenes menos los verdaderos positivos, y finalmente una breve explicación de las condiciones meteorológicas de la noche.

Dia	Imgs.	Detecciones	V. Positivos	F. Positivos	% F. Pos	Condiciones
30/05/2021	1276	0	0	0	0,00%	Nublado toda la noche
31/05/2021	1008	38	18	20	2,02%	Nubosidad ocasional
01/06/2021	993	7	0	7	0,70%	Nublado toda la noche
02/06/2021	1056	45	13	32	3,07%	Nubosidad ocasional
03/06/2021	1197	2	0	2	0,17%	Nublado toda la noche
04/06/2021	982	4	0	4	0,41%	Nublado toda la noche

05/06/2021	1016	6	0	6	0,59%	Nublado toda la noche
06/06/2021	977	103	86	17	1,91%	Cielo claro
07/06/2021	975	127	95	32	3,64%	Cielo claro
08/06/2021	971	62	32	30	3,19%	Nubosidad ocasional
09/06/2021	968	59	24	35	3,71%	Nubosidad ocasional
10/06/2021	1083	34	7	27	2,51%	Nubosidad ocasional
11/06/2021	969	81	38	43	4,62%	Nubosidad ocasional

Tabla 6: Resultados Prueba Concepto
Fuente: Elaboración propia

A destacar, que claramente las condiciones meteorológicas afectan al rendimiento del modelo.
En noches muy nubladas, se producen pocas detecciones y en todas las ocasiones son falsos positivos. Es decir, el modelo responde bien a la situación de muchas nubes y prácticamente no genera detecciones porqué no se producen.

En noches con nubosidad ocasional aumentan el número de detecciones, y el porcentaje de falsos positivos (aunque se mantiene por debajo del 5% en todos los casos).
Se ha detectado que las nubes ocasionales pueden generar una mayor ratio de falsos positivos a causa del restado de imágenes con nubosidad, que, en ocasiones muy concretas, puede producir como resultado una estela parecida a un meteoro. Este es un aspecto que se debería mejorar de cara a futuras iteraciones, probablemente complementando el modelo actual con un modelo entrenado sin restado de imágenes.

En noches con cielo claro, el número de detecciones es muy elevado, pero se mantiene el porcentaje bajo de falsos positivos.

Otro aspecto para mejorar es que, en la actualidad, como se ha comentado, el modelo no es capaz de detectar cuando se trata de un meteoro o un satélite artificial. La estela que dejan es prácticamente idéntica. Existe un factor, ajeno al modelo, que podría facilitar dicha detección. Los satélites artificiales aparecen en imágenes consecutivas, mientras que los meteoros no.
Es importante recordar que las imágenes sobre las que trabaja el modelo son de 30 segundos de exposición. Por lo tanto, si una estela aparece en varias imágenes consecutivas es extremadamente improbable que se trate de un meteoro. Este punto se detallará en el apartado de conclusiones y trabajo futuro.

Durante los días de la prueba de concepto y específicamente las noches más claras, se realizaron multitud de detecciones, mayoritariamente estrellas fugaces y satélites artificiales, pero también se detectaron bólidos que se reportaron al SPMN. La mayoría de ellos al carecer de importancia científica no se publican en su registro.
Específicamente el bólido SPMN060621C [45] del día 06/06/2021 fue detectado por el *pipeline* de detección, y el Observatori de Pujalt lo notificó al SPMN que lo catalogó para posterior estudio.

[45] https://twitter.com/RedSpmn/status/1401925043899273228

En la siguiente imagen, en la primera línea, se puede observar el registro en la web de SPMN y la notificación que realizaron en su cuenta de Twitter. En la segunda línea se puede observar el registro de la detección en la web del proyecto (con un *scoring* de 0.98) y la detección realizada en el canal de Telegram del *pipeline*. (notar que en la web del SPMN la detección se muestra en UTC[46] mientras que el *pipeline* lo muestra en tiempo local, que se corresponde a +2 horas, y, por lo tanto, corresponde al día 07/06 a las 00:01:43)

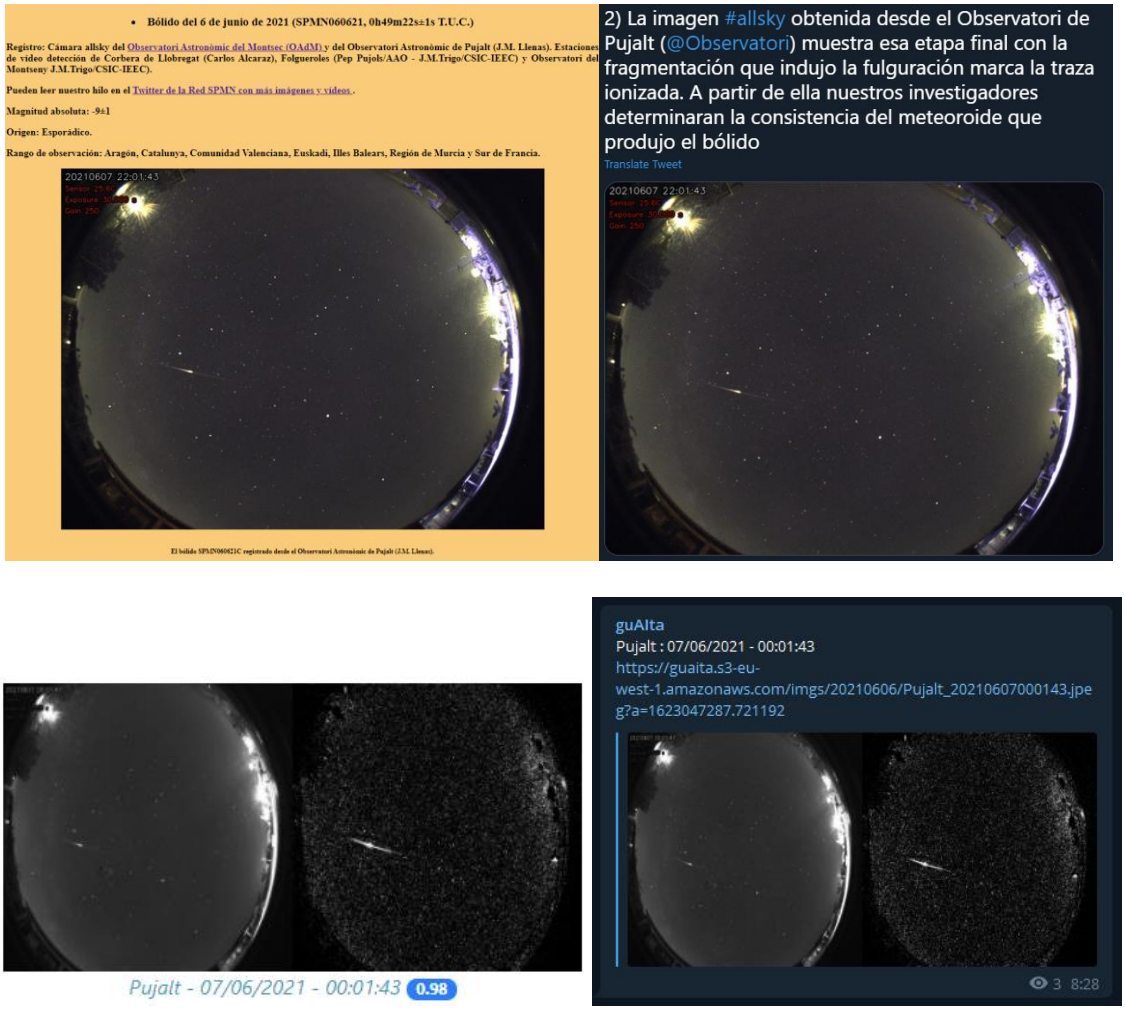


Figura 34: Bólido SPMN060621C
Fuente: spmn.uji.es y elaboración propia

[46] https://es.wikipedia.org/wiki/Tiempo_universal_coordinado

5. Conclusiones

Como conclusiones más destacadas, cabe remarcar que se ha podido validar la posibilidad de generar un modelo capaz de detectar meteoros en base a redes convolucionales pre-entrenadas y usando *transfer learning*.
Este aspecto es importante, porqué el hecho de usar redes pre-entrenadas, reduce los tiempos y recursos requeridos para el entrenamiento y, por lo tanto, facilita en gran manera el proceso de prueba y error que acompaña estos proyectos.
Otra conclusión interesante es que, gracias al avance en *frameworks* de IA, es factible afrontar proyectos en los que las condiciones iniciales no son del todo favorables, como en este caso, en el que no se disponía de un dataset de imágenes etiquetado y el número de casos positivos se reducía a menos de 20.
Mediante Fast.ai y sus funcionalidades de alto nivel en *transfer learning* y *data augmentation*, se ha podido generar un modelo capaz de generalizar y sin caer en el *overfitting*.
También es importante destacar que, mediante el pre-procesado de las imágenes, se pudo realzar las trazas que dejan los meteoros o satélites artificiales. El hecho de pre-procesar las imágenes permitió disponer de un dataset de casos positivos suficientemente grande para afrontar el proyecto.
Como conclusión importante sobre el pre-procesado y específicamente sobre el restado de imágenes consecutivas, cabe destacar que, en ocasiones muy concretas (y poco probables), el restado de imágenes consecutivas con presencia de nubes genera patrones que el modelo confunde con meteoros.
Otra conclusión importante, es asumir que el modelo que se ha generado permite detectar estelas en el cielo, y en ocasiones estas serán meteoros y en ocasiones no. Por lo tanto, será interesante aplicar ciertas medidas correctoras en el futuro, como por ejemplo parametrizar el software de inferencia para que no notifique detecciones consecutivas, asumiendo que probablemente serán producidas por satélites artificiales.
Otro punto que comentar es que el modelo generado, tiene un porcentaje de acierto elevado, pero dado que diariamente se procesan alrededor de 1.000 imágenes, es inevitable que se generen falsos positivos. Para minimizar este aspecto es importante seguir entrenando el modelo con los datos que se vayan recogiendo en adelante y además mejorar la inferencia mediante el uso de otros modelos más especializados. Como ya se ha comentado, una buena estrategia puede ser usar un modelo entrenado con imágenes sin restar y aplicar una lógica en base al *scoring* de ambos modelos para generar o no una detección. De esta manera se podrían corregir los efectos no deseados del restado de imágenes.

En general, las conclusiones son positivas. El *pipeline* funciona correctamente y el modelo permite simplificar las tareas del operador, pero es importante destacar que el *pipeline* está lejos de poder notificar automáticamente detecciones al SPMN o cualquier otra organización. El operador aún es necesario porqué, aunque se genere un porcentaje bajo de falsos positivos, estos existen.
Para poder llegar al punto en que el *pipeline* notifique automáticamente, se deberían realizar las mejoras comentadas en la inferencia y añadir más observatorios. Dado que los observatorios notifican de forma centralizada a la nube, un escenario futuro interesante, puede ser disponer de varios

observatorios y solo notificar una detección si todos ellos o la mayoría de ellos concluyen que se ha producido una detección.
En este aspecto, cabe remarcar que el diseño del *pipeline* se ha realizado para que el coste de incorporar más observatorios sea muy bajo.

Se considera que este trabajo debe tomarse como un punto de partida. Dada la baja cantidad de casos positivos que se disponía, este proyecto se encaró hacia capturar el máximo numero de meteoros posibles y enriquecer el dataset de casos positivos. Por este motivo se ha creado un sistema de notificación centralizado en la nube, para así, tener disponibles muchas más imágenes de meteoros, poder iterar sobre el modelo reentrenándolo y actualizar el software de inferencia con las nuevas versiones del modelo.
Se descartó en su día realizar directamente la inferencia en la nube porqué algunos observatorios pueden tener ciertos problemas de conectividad y para evitar el coste que podría suponer enviar a la nube más de 1.000 imágenes por noche y observatorio.

Por último, hay que destacar, que tal y como se ha comentado, el objetivo del trabajo se marcó en maximizar el *recall* de la clase meteoro, y sobre el entorno de test se consiguió un 0.98. Es decir, que cuando la imagen contiene un meteoro en un 98% de las veces el modelo lo detecta. Adicionalmente a este porcentaje, es importante recordar que se aplicaron técnicas de *class activation maps* sobre la clase meteoro, para así asegurarnos de que el modelo se fijaba en los elementos correctos de la imagen para afirmar que pertenecía a la clase meteoro.

Como puntos a mejorar en el *pipeline* y, por lo tanto, trabajo futuro, cabe destacar:

- Reentrenar el modelo con muchos más casos positivos.
- Aplicar mejoras en el software de inferencia para reducir los falsos positivos causados por satélites artificiales.
- Generar una versión de escritorio del software de inferencia. En la actualidad se ejecuta como script Python.
- Mejorar la API de comunicación usando un sistema de autenticación más robusto.
- Publicar el dataset en Kaggle para que la comunidad proponga mejoras.

Y como puntos a mejorar en cuanto a la implantación del proyecto, sería interesante poder desplegar el software de inferencia en otros observatorios y de esta manera facilitar el cálculo de la trayectoria de caída del meteoro y además, poder aplicar técnicas más interesantes de notificación automatizada de meteoros en base a consenso de varios observatorios.

6. Bibliografía

[1] Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner (1998). *Gradient-Based Learning Applied to Document Recognition.*
[2] Karen Simonyan, Andrew Zisserman (2015). *Very Deep Convolutional Networks for large-scale image recognition.*
[3] D.Cecil y M.Campbell-Brown (2020). *The application of convolutional neural networks to the automation of a meteor detection pipeline*. Pag 11
[4] De Cicco, Marcelo; Zoghbi, Susana; Stapper, Andres P.; Ordoñez, Antonio J.; Collison, Jack; Gural, Peter S.; Ganju, Siddha; Galache, Jose-Luis; Jenniskens, Peter (2018). *Artificial intelligence techniques for automating the CAMS processing pipeline to direct the search for long-period comets.*
[5] Yuri Galindo, Anna Carolina Lorena (2018). *Deep Transfer Learning for Meteor Detection.*
[6] Gural, Peter S (2019). *Deep learning algorithms applied to the classification of video meteor detections.*
[7] Gural, Peter S (2008). Algorithms and Software for Meteor Detection.
[8] Bolei Zhou et al. (2015). Learning Deep Features for Discriminative Localization
[8] https://towardsdatascience.com/how-transfer-learning-works-a90bc4d93b5e
[9] https://thenewstack.io/the-ultimate-guide-to-machine-learning-frameworks/
[10]https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78
[11] http://www.spmn.uji.es/
[12] https://www.fast.ai/
[13] https://www.imo.net/

Anexo 1: Estructura de una ResNet34

Se detalla a continuación la estructura de una ResNet34 mostrada en base al conjunto de capas de las que se compone.

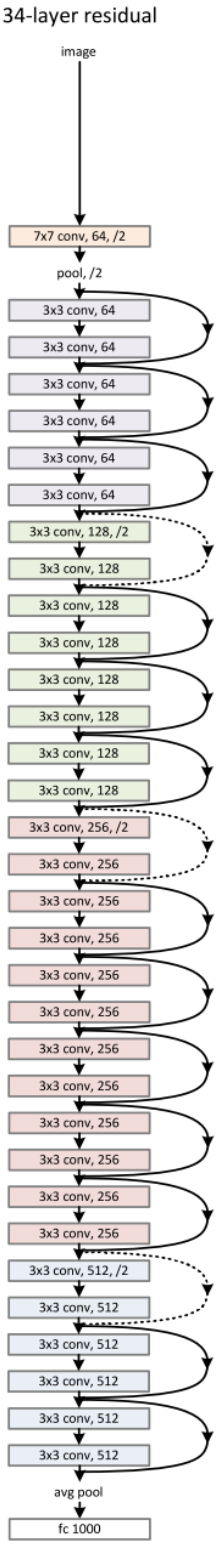


Figura 35: Estructura de una ResNet34
Fuente: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

A continuación, se muestra la definición en Fast.ai de la misma ResNet34

```
Sequential(
  (0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (4): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (5): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (2): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (3): BasicBlock(
        (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (6): Sequential(
```

```
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (3): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (4): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (5): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(7): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
```

```
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
   )
  )
  (1): Sequential(
   (0): AdaptiveConcatPool2d(
    (ap): AdaptiveAvgPool2d(output_size=1)
    (mp): AdaptiveMaxPool2d(output_size=1)
   )
   (1): Flatten(full=False)
   (2): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   (3): Dropout(p=0.25, inplace=False)
   (4): Linear(in_features=1024, out_features=512, bias=False)
   (5): ReLU(inplace=True)
   (6): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   (7): Dropout(p=0.5, inplace=False)
   (8): Linear(in_features=512, out_features=2, bias=False)
  )
)
```

Anexo 2: Proceso de entrenamiento de la ResNet18

Se detallan a continuación las 35 *epochs* del procedimiento de entrenamiento de la ResNet18, con los valores de las métricas para cada *epoch*.

epoch	train_loss	valid_loss	error_rate	accuracy	precision_score	f1_score	time
0	0.838610	0.571901	0.281150	0.718850	0.723393	0.713942	0:41
1	0.803993	0.560540	0.249201	0.750799	0.750238	0.749877	0:41
2	0.782298	0.528214	0.257188	0.742812	0.742992	0.741093	0:41
3	0.698753	0.507604	0.226837	0.773163	0.773163	0.773015	0:41
4	0.654818	0.461071	0.193291	0.806709	0.808311	0.805206	0:41
5	0.628972	0.429607	0.185304	0.814696	0.816974	0.813092	0:41
6	0.584568	0.452043	0.220447	0.779553	0.778983	0.779111	0:43
7	0.552143	0.425085	0.193291	0.806709	0.806176	0.806294	0:41
8	0.504984	0.435844	0.191693	0.808307	0.808307	0.808181	0:41
9	0.447361	0.366071	0.175719	0.824281	0.823795	0.823978	0:41
10	0.437134	0.359059	0.164537	0.835463	0.843530	0.832936	0:44
11	0.398034	0.344151	0.150160	0.849840	0.851551	0.848797	0:44
12	0.400914	0.363281	0.166134	0.833866	0.845921	0.833431	0:41
13	0.378998	0.308167	0.151757	0.848243	0.847817	0.847870	0:40
14	0.356866	0.296856	0.134185	0.865815	0.866439	0.865780	0:41
15	0.335868	0.324765	0.153355	0.846645	0.852948	0.846545	0:43
16	0.329427	0.281342	0.115016	0.884984	0.884582	0.884815	0:41
17	0.293908	0.292421	0.137380	0.862620	0.863808	0.862607	0:41
18	0.284020	0.282753	0.116613	0.883387	0.883129	0.883062	0:43
19	0.294557	0.261082	0.113419	0.886581	0.888582	0.885824	0:45
20	0.283444	0.247845	0.102236	0.897764	0.897943	0.897712	0:44
21	0.280663	0.271215	0.118211	0.881789	0.883700	0.881788	0:42
22	0.258704	0.254404	0.100639	0.899361	0.899153	0.899268	0:45
23	0.262213	0.250810	0.102236	0.897764	0.898987	0.897754	0:43
24	0.245186	0.269664	0.100639	0.899361	0.899153	0.899268	0:41
25	0.249133	0.241461	0.089457	0.910543	0.910431	0.910278	0:40
26	0.247837	0.236015	0.087859	0.912141	0.911933	0.912060	0:40
27	0.226847	0.241507	0.086262	0.913738	0.914132	0.913706	0:41
28	0.221255	0.236792	0.086262	0.913738	0.913355	0.913611	0:40
29	0.227504	0.228003	0.084665	0.915335	0.915619	0.915299	0:41
30	0.215469	0.225727	0.081470	0.918530	0.918324	0.918455	0:42
31	0.219962	0.226292	0.084665	0.915335	0.914950	0.915178	0:45
32	0.218590	0.223887	0.081470	0.918530	0.918168	0.918420	0:48
33	0.226887	0.223008	0.083067	0.916933	0.916842	0.916687	0:41
34	0.220581	0.223404	0.079872	0.920128	0.919792	0.920029	0:41

Tabla 7: Resultado del training del modelo ResNet18
Fuente: Elaboración propia

Se aporta también la gráfica con los valores de *loss* para *train* y *valid* para los distintos *batches* de imágenes proporcionados en el proceso de *train*:

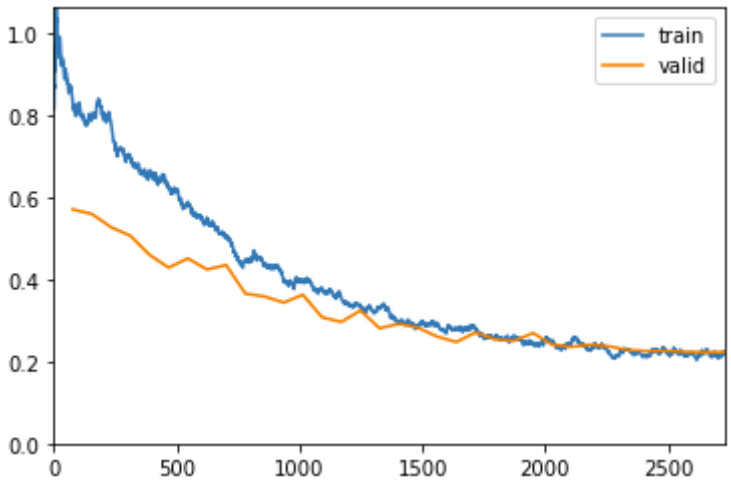


Figura 36: Gráfica de loss para train y validation (ResNet18)
Fuente: Elaboración propia