Trabajo Fin de Grado
Grado Universitario en Ingeniería de las Tecnologías de Telecomunicación

Segmentación de lesiones de psoriasis mediante el uso de Redes Neuronales

Autor: Rafael Díaz Castillo

Tutor: José Antonio Pérez Carrasco

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2024

Trabajo Fin de Grado
Grado Universitario en Ingeniería de las Tecnologías de Telecomunicación

Segmentación de lesiones de psoriasis mediante el uso de Redes Neuronales

Autor:
Rafael Díaz Castillo

Tutor:
José Antonio Pérez Carrasco
Profesor titular

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo Fin de Grado: Segmentación de lesiones de psoriasis mediante el uso de Redes Neuronales

Autor: Rafael Díaz Castillo

Tutor: José Antonio Pérez Carrasco

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

Agradecer a mis padres Ana María y Rafael, por todo su amor, comprensión y entendimiento durante todos estos años. Ayundandome a no rendirme y demostrándome los valores fundamentales de esta vida. Sin vosotros esto habría sido imposible. A mis abuelos por ser mi lugar seguro. Sus historias y enseñanzas siempre han sido una guía para mí. A mi pareja María, gracias por ayudarme en esta etapa de mi vida. A mi mascota Yukiki, por su lealtad y compañía, gracias.

Tambien agradecer, a mi tutor José Antonio, por su orientación, su dedicación y por confiar en mi. Sus consejos y apoyo han sido clave para alcanzar y desarrollar este proyecto. Y a todos los profesores que he tenido a lo largo de este camino, aportando su sabiduría.

Rafael Díaz Castillo

Sevilla, 2024

Resumen

En la actualidad, el avance en el uso de técnicas elaboradas a través de inteligencias artificiales y herramientas automáticas en el ámbito tecnológico es una realidad cada vez más evidente. Ayudando en tareas complicadas u obteniendo unos resultados con gran rapidez y eficacia.

En el entorno de la medicina un diagnóstico precoz puede suponer una gran ventaja en la lucha contra las patologías y enfermedades como la psoriasis, una enfermedad cutánea, que conformará la base de datos del proyecto.

Este Trabajo de Fin de Grado (TFG) se enfoca en la comparativa de técnicas de segmentación aplicadas a imágenes médicas, centrándose específicamente en dos variedades de psoriasis la *chronic plaque* y la *guttata*. Se aborda la investigación del funcionamiento de un método semiautomático de segmentación mediante el *software* Fiji y se contrasta con un enfoque automático basado en el uso de dos redes neuronales convolucionales actuales XNET y UNET.

El objetivo principal es realizar una comparativa exhaustiva de las técnicas de segmentación, evaluando su desempeño en términos de precisión, sensibilidad y especificidad. Este estudio contribuye a la comprensión de la aplicación de herramientas semiautomáticas y redes neuronales en el ámbito dermatológico, ofreciendo perspectivas valiosas para futuras investigaciones y aplicaciones clínicas en la identificación y tratamiento de la psoriasis.

Para terminar, se exponen los resultados obtenidos después de aplicar cada tecnología, así como la comparativa entre ellos dando una comprensión de la aplicación de redes neuronales convolucionales en el ámbito dermatológico y cómo puede ayudar en futuras aplicaciones.

Abstract

Currently, progress in the use of techniques developed through artificial intelligence and automatic tools in the technological field is an increasingly evident reality. Helping with complicated tasks or obtaining results very quickly and efficiently.
In the medical field, early diagnosis can be a great advantage in the fight against pathologies and diseases such as psoriasis, a skin disease, which will form the project's database.
This Final Degree Project (TFG) focuses on the comparison of segmentation techniques applied to medical images, specifically focusing on two varieties of psoriasis, chronic plaque and guttata. The investigation of the operation of a semi-automatic segmentation method using the Fiji software is addressed and it is contrasted with an automatic approach based on the use of two current convolutional neural networks XNET and UNET.
The main objective is to carry out a comprehensive comparison of the segmentation techniques, evaluating their performance in terms of precision, sensitivity and specificity. This study contributes to the understanding of the application of semi-automatic tools and neural networks in the dermatological field, offering valuable perspectives for future research and clinical applications in the identification and treatment of psoriasis.
Finally, the results obtained after applying each technology are presented, as well as the comparison between them, giving understanding convolutional neural networks in the dermatological field and how it can help in future applications.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

CNN	Red Neuronal Convolucional
GPU	Unidad de Procesamiento Grafico
CPU	Unidad Central de Procesamiento
CP	Psoriasis en Placas
ROC	Región de Convergencia
IA	Inteligencia Artificial
vp	Verdadero positivo
fp	Falso positivo
fn	Falso negativo
vn	Verdadero negativo
HDF5	Formato de datos jerárquicos versión 5
RELU	Unidad lineal rectificada
WEKA	Entorno para análisis del conocimiento de la Universidad de Waikato
RNN	Red Neuronal Recurrente
DNN	Red Neuronal Profunda
TFG	Trabajo Fin de Grado

1 INTRODUCCIÓN

La cueva a la que tememos entrar tiene el tesoro que buscamos.

- Joseph Campbell -

La necesidad del ser humano por buscar mejorar su calidad de vida y facilitar el desempeño de las tareas, hace que surja la motivación por buscar una automatización con el empleo de tecnologías innovadoras.

La inteligencia artificial es una de dichas innovaciones tecnológicas. Se considera rama de la tecnología que en poco tiempo ha experimentado un gran avance, sobretodo en lo que respecta en la optimización de procesos sin intervención humana, agilizando los procesos y reduciendo los posibles errores [1]. Aplicadas a ciertos ámbitos como puede ser el campo de la medicina, puede resultar muy provechoso en tareas repetitivas o que es necesario analizar un gran volumen de datos, como el diagnostico de una enfermedad y detectar a tiempo patologías que puedan verse agravadas.

Por ello se desea aplicar la inteligencia artificial en una de las tareas de prevención y diagnósticos, como es la segmentación de lesiones de tejidos. En el caso de este proyecto fin de grado, se ha optado por una aplicación en la detección de Psoriasis. "La Psoriasis es una enfermedad de la piel que causa un sarpullido con manchas rojas y escamosas que pican, esta enfermedad es crónica y no tiene cura"[2]. Aun no teniendo cura, con una detección rápida pueden paliarse sus efectos y reducir sus síntomas.

1.1 Objetivos

Con este proyecto se quiere establecer un estudio de varias redes neuronales convolucionales aplicadas a la segmentación de imágenes médicas, en concreto en dos tipos de psoriasis, conocidas como *guttata* y en placas. Para lograrlo se realizará el procedimiento descrito a continuación:

- La creación de una base de datos, compuesta por imágenes médicas de los dos tipos de psoriasis antes nombradas, así como de las máscaras de cada imagen. Para las máscaras se utiliza el programa de segmentación semiautomatica *Weka Trainable Segmentation*.
- La preparación de la base de datos con un aumento del dataset, mediante la aplicación de transformaciones geométricas, evitando posibles problemas con el entrenamiento de las redes neuronales convolucionales. Asi como el estudio del formato HDF5 para la formación de datasets.
- Descripción de un conjunto de redes neuronales existentes en las actualidades enfocadas a la segmentación de imágenes y utilizadas en el campo de la medicina.
- Aplicación de un código utilizando python y configurando unos parámetros para la realización del entrenamiento de las redes neuronales convolucionales XNET y UNET, se ejecutan varios entrenamientos de cada una para observar los distintos datos que se consiguen.
- Evaluación de los resultados obtenidos en ambas redes neuronales. Observando los parámetros de rendimientos, tales como la precisión o los derivados de utilizar la matriz de confusión, y analizando varios ejemplos conseguidos junto con las verdades fundamentales de UNET y XNET

respectivamente, se desea establecer una comparación del rendimiento entre ellas, así como las ventajas y desventajas que se presentan no solo en los resultados, sino que también en el proceso de aprendizaje y entrenamiento.

2 BASE DE DATOS

Pasan los años, y quedan la memoria y las fotos.

- Nestor Almendros -

Uno de los dos aspectos fundamentales, en conjunto con la capacidad computacional en el trabajo fin de grado ha sido la creación de una base de datos o dataset, es un punto a destacar debido a que es la fuente de información que se utiliza para poder entrenar la red convolucional, y, conseguir los datos de evaluación.

En general se usará una base de datos creada con imágenes de dos variedades distintas de psoriasis. La base de datos se conforma de imágenes con distintas resoluciones, todas ellas dotadas de unas máscaras de segmentación, organizadas mediante una jerarquía de directorios, para posteriormente ser implementado en el código de modelación del modelo de red convolucional. Previamente todos los datos se normalizarán en un tamaño 160 x 160. Finalmente se aplicará un aumento de la misma con el fin de conseguir mejores resultados a la hora de entrenar el modelo.

2.1. Obtención de imagenes y permiso de uso

La base de datos utilizada ha sido creada en base a las imágenes obtenidas de Dermnet [3] , un repositorio de imágenes médicas sobre lesiones cutáneas y dermatológicas. Este repositorio es de uso gratuito para fines educativos, no obstante se ha contactado a través de un formulario de la página web, para hacer la petición de los permisos de usos pertinentes, obteniendo una favorable respuesta.

2.2. Creacion de base de datos

La base de datos ha sido creada conforme a los objetivos de este proyecto, por este motivo y al no encontrar ninguna base predefinida por ninguna institución, con las máscaras ya creadas, se realizó desde cero de forma manual, acudiendo a la página Dermnet.org para escoger las imágenes y a partir de ahí con la utilización de un programa de tratamiento de imágenes, se realizó la segmentación de las imágenes.

En total las imágenes segmentadas han sido 77 imágenes, con un tamaño de 160 x 160. Debido al pequeño volumen de imágenes, se intenta no reducir la resolución en exceso. Con ello, se maximiza la extracción de detalles precisos, especialmente alrededor de estructuras más pequeñas, esencial en el tipo *guttata.*

2.2.1 Composición del dataset

Este dataset se diversificará en dos de los tipos de psoriasis más comunes, como son la psoriasis en placas y *guttata* o en gotas.

Se han utilizado un total de 26 imágenes corriespondientes a psoriasis en gotas y 51 de tipo placas, con sus respectivas capas de segmentación.

- Psoriasis en placas [4]: este tipo como la define mayoclinic.org, es la más común y se caracteriza por

producir manchas secas y elevadas en la piel que provocan picazón y están cubiertas de escamas. Normalmente aparecen en las articulaciones, así como cuero cabelludo y espalda.

[5]

Figura 1 – Psoriasis en placas

- Psoriasis *guttata*: es muy común en niños y jóvenes adultos y como su propio nombre nos indica se presenta en estas formas de goteo o patrón de pequeñas manchas en una región o zona del cuerpo.

[6]

Figura 2 – Psoriasis *guttata*

2.3. Jerarquia de directorios

La forma de ordenar los directorios, empieza con una carpeta raíz denominada Base de datos. En el directorio raíz se alojan las carpetas Guttata y CP haciendo referencia correspondientemente a los tipos *guttata* y *chronic plaque* de psoriasis, divididas en directorios distintos para un mejor balanceo de categorías a la hora de aumentar los datos. En cada carpeta simétricamente se alojan los directorios Images, donde se encuentran las imágenes originales a segmentar, y la carpeta Labels, que contiene las máscaras de segmentación de cada una de las imágenes.

Cada imagen tiene que corresponder al nombre, formato y resolución con respecto a su máscara de

segmentación, para evitar posibles errores de entrenamiento. A continuación se muestra un esquemático para su mejor comprensión.

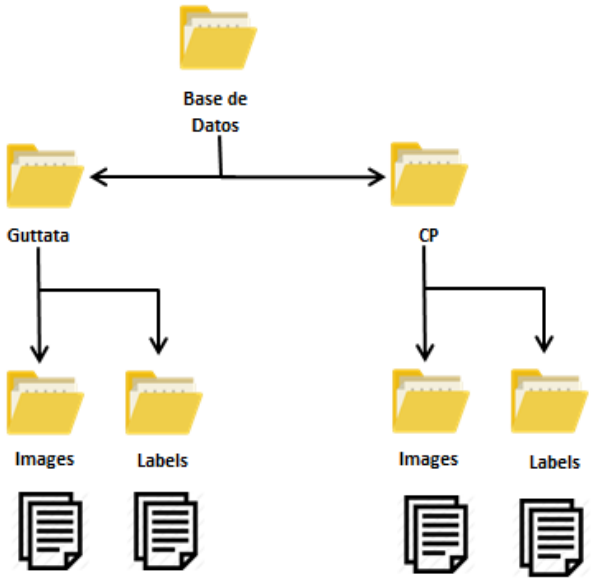


Figura 3 – Jerarquía de directorios de la base de datos

2.4. Segmentacion semiautomatica

La creación de las máscaras de segmentación semiautomática se ha realizado mediante el programa ya existente ImageJ [7], con ello se conforma la base de datos completamente. Existen diversas distribuciones del programa imagen, en este proyecto en concreto se ha usado la distribución Fiji, conocida ya su funcionamiento por usos anteriores, siendo una distribución estable y con una gran facilidad de ampliación de funciones mediante la instalación de paquetes externos.

Para el desempeño de la tarea se ha hecho uso de un plugin de segmentación semiautomática, conocido como Trainable WEKA Segmentation [8].

2.4.1 WEKA Segmentation

WEKA son las siglas de *Waikato Environment for Knowledge Analysis*, es un repositorio de software libre. Cuenta con una gran cantidad de herramientas de algoritmos enfocados al aprendizaje automático y la exploración de datos, y fue creado en la universidad de Waikato, Nueva Zelanda, desarrollado en lenguaje de programación Java.

Es un plugin que combina una colección de algoritmos de *machine learning* junto con herramientas de visualización para la segmentación a nivel de píxel de una serie de imágenes.

Al ser *software* de uso libre, cuenta con una gran colección de datos previos y muy buena portabilidad. Es fácil de manejar dado su entorno gráfico, bastante simple pero potente.

Un punto notorio de este plugin es que una vez entrenado una serie de clasificadores manualmente, estos pueden ser guardados para su posterior implementación en distintos set de imágenes, incluso, como punto de partida en la creación de unos nuevos.

2.4.2 Aplicación del software

Una vez comentado en el punto anterior el programa y plugin a utilizar en la segmentación semiautomática. Se pasa a realizar en este apartado una ejemplificación de uso del programa.

Tras iniciar Fiji, habrá una barra de herramientas como en la figura 4. En la cual se seleccionará plugins, Segmentation y Trainable WEKA Segmentation, mostrado en la figura 5, para ejecutar el plugin de WEKA y comenzar la segmentación.

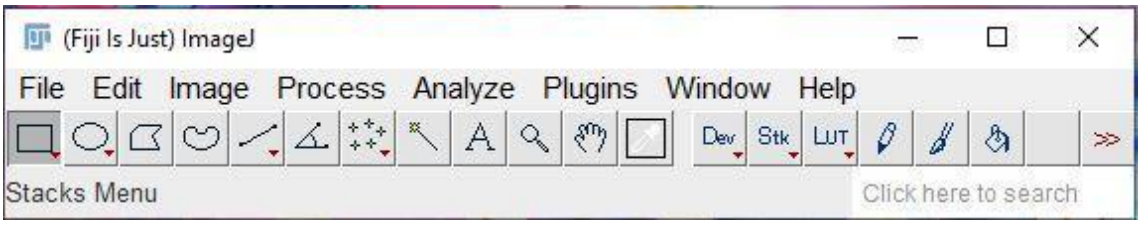


Figura 4 – Barra de herramientas

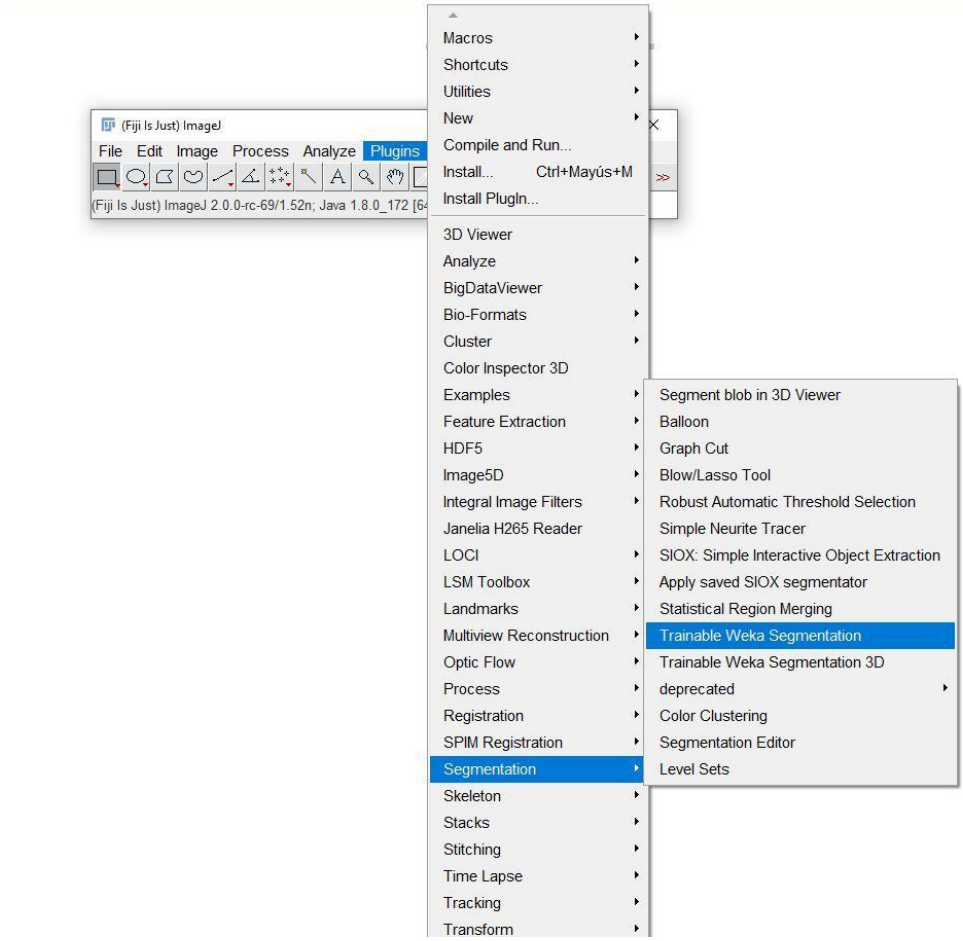


Figura 5 – Selección de Trainable Weka Segmentation

Antes de poder ejecutar WEKA, se requiere seleccionar la imagen o conjunto de imágenes a segmentar. Tras este paso WEKA se verá de la forma indicada a continuación.

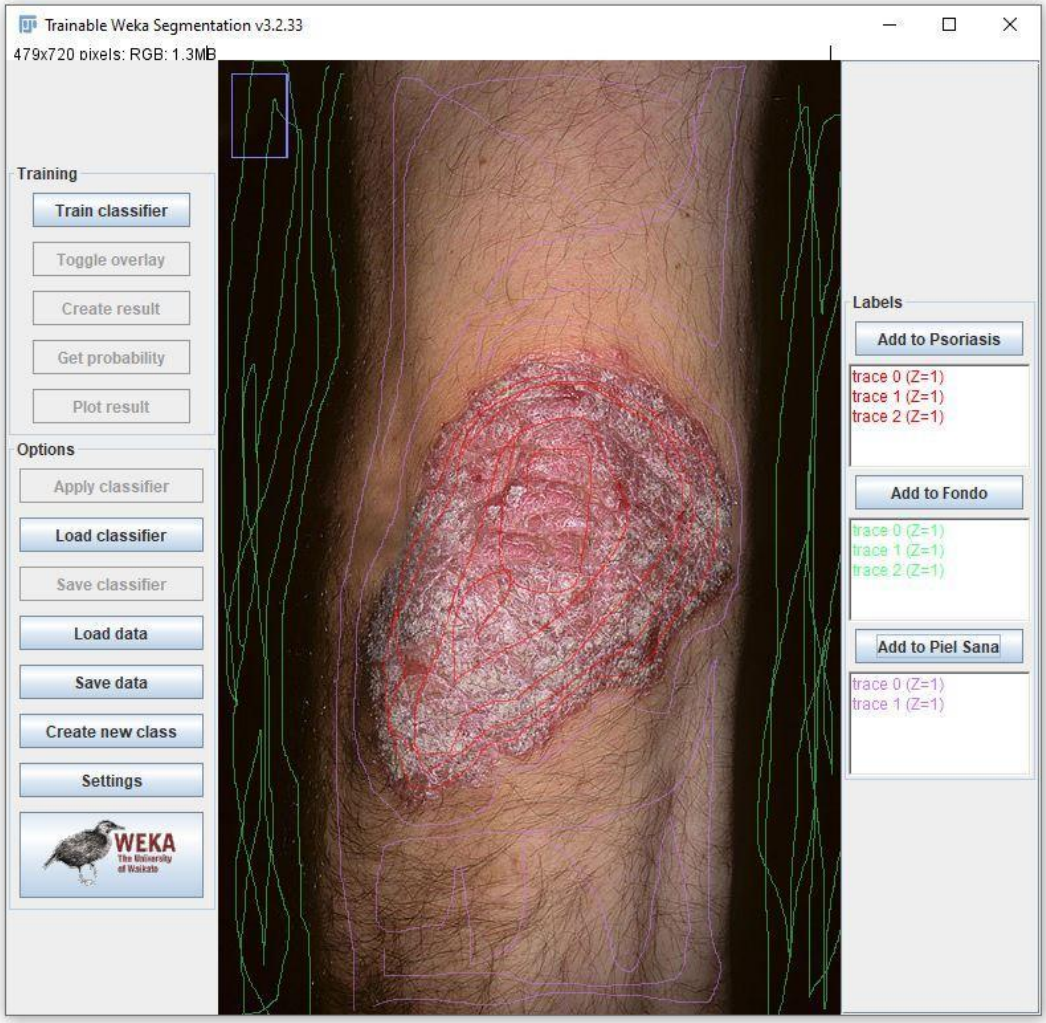


Figura 6 – Trainable Weka Segmentation

En la imagen anterior se aprecia todo el programa de WEKA iniciado, en la parte derecha, se encuentran definidos los clasificadores, por defecto solo vienen dos, pero en la parte izquierda hay una opción para crear nuevos. En el proyecto se diferencian tres partes, por lo que se creará una nueva capa para la piel sana.

Añadir que en la pestaña *settings* localizada a la izquierda se pueden modificar diversos parámetros para el entrenamiento, además de poder modificar el nombre de los clasificadores, resultando una tarea más sencilla a la hora de seleccionar manualmente las regiones de clasificación.

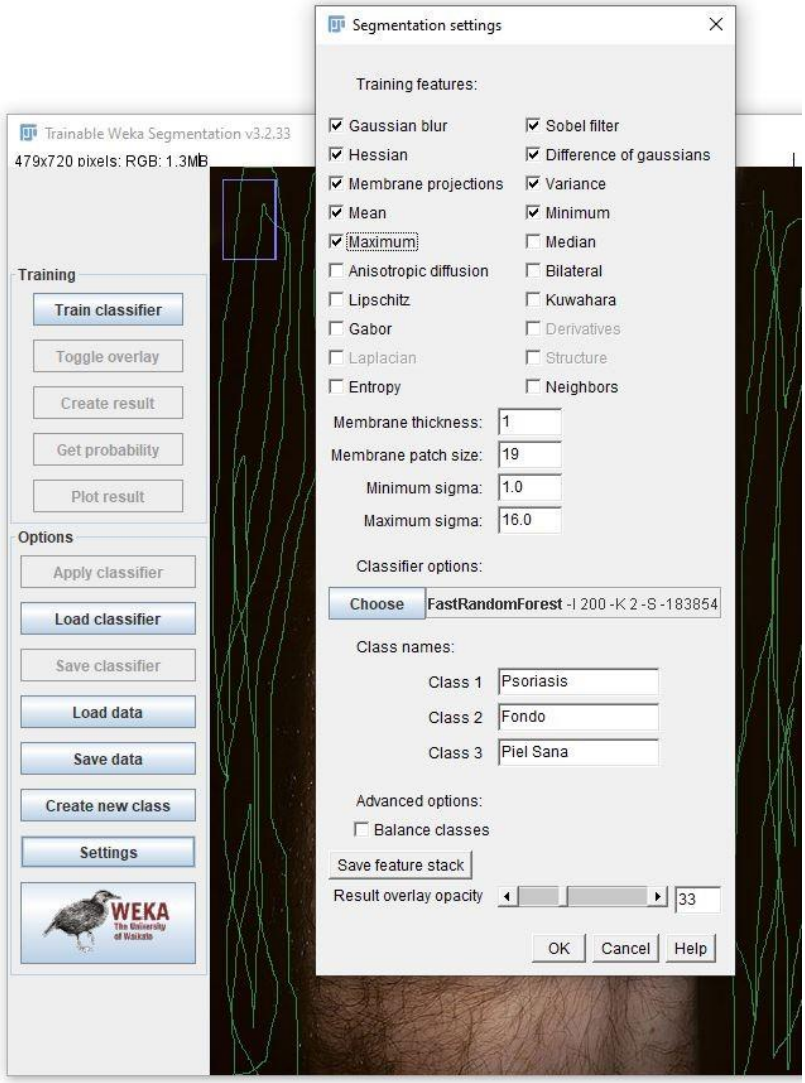


Figura 7 – Settings de parámetros de segmentación

En la pestaña *settings* se hizo una selección de características de entrenamiento, las cuáles fueron:

- *Gaussian blur*
- *Hessian*
- *Membrane projections*
- *Mean*
- *Maximum*
- *Minimum*
- *Variance*
- *Membrane thickness: 1*

- *Membrane patch size: 16*
- *Sigma*, este valor se configuró entre los valores 1 y 16.

Con la activación de estos parámetros se observa una mejora en la segmentación con respecto a las curvaturas y bordes de cada región que se pretende diferenciar.

Para configurar cada clasificador hay que dibujar trazos encima de la imagen indicando la zona a la que pertenece, una vez hecho el trazo manualmente. Habrá que añadir cada uno con la pestaña correspondiente del clasificador, haciendo click en los botones de la derecha, en los recuadros inferiores se muestran los trazos realizados. Con doble click se puede borrar cualquiera de ellos.

Una vez terminada la selección de todo lo destacable, se ejecuta el entrenamiento pulsando en el apartado superior izquierdo la opción *train classifier*.

Figura 8 – Selección de trazado

Tras el entrenamiento se puede formar la imagen segmentada mediante *create result*. Dando como resultado final el siguiente expuesto. Cabe a destacar que, si el resultado no es el esperado, se puede modificar y volver a entrenar.

Figura 9 – Resultado de segmentación semiautomática

Para finalizar, los resultados obtenidos de la segmentación se guardaran haciendo uso de la barra de Fiji. Se guarda en formato *jpeg*. La resolución es la misma que la imagen original. El nombre debe de coincidir.

3 DATA AUGMENTATION

El Hombre que mueve montañas empieza apartando piedras pequeñas.

- Confucio -

Un problema común en el desarrollo de la segmentación mediante el entrenamiento de una red convolucional en *deep learning*, son los relacionados con el *overfitting* y *underfitting* [9]. Estas son las principales causas que diferencian un resultado fiable a uno no deseado.

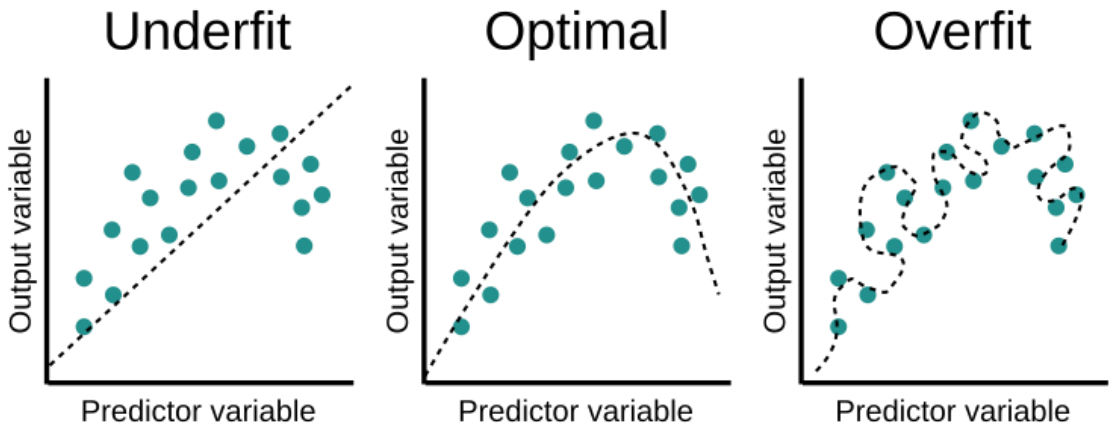
Para mitigar las posibles causas negativas es usual, realizar un incremento virtual del número de imágenes, mediante transformaciones o modificaciones de las imágenes. A esto se le conoce como Data Augmentation.

3.1 Overfitting y Underfitting

El *overfitting* y *underfitting* se dan principalmente tras un entrenamiento de una red neuronal muy simple o con pocos datos de entrenamiento. Aunque son problemas realcionados con el ajuste de la red neuronal, cada uno tiene un efecto distinto en los resultados.

En los casos con bases de datos compuestas por imágenes muy específicas o una red neuronal muy compleja. El modelo consigue captar unas características muy concretas y no sabiendo generalizar. Estas ocasiones se conocen como *overfitting*.

En cambio, el *underfitting*, se da cuando no se cuenta con un gran número de imágenes o la red neuronal es muy simple y a la hora del entrenamiento el modelo no consigue obtener las características de los datos proporcionados, estas características de predicción son necesarias para realizar las tareas de segmentación.



[10]

Figura 10 – Descripción gráfica de *underfitting*, *overfitting* y *fitting*

3.2 Tipos de transformaciones

Existen muchos tipos de transformaciones en el tratamiento de imágenes. El uso de dichas transformaciones tiene como finalidad aumentar de manera ficticia el volumen de imágenes que conforman la base de datos, con dicho aumento de volumen del dataset, se consigue fortalecer el programa frente a problemas de aprendizaje. Se procede a enumerar las más convenientes para el aumento del dataset, ya que el uso de ciertas transformaciones puede llegar a influir negativamente a la hora de entrenar el modelo de CNN.

Las transformaciones implementadas en el código usado, son obtenidas del paquete *Imgaug* [11], una librería de Python que contiene varias técnicas de generación de imágenes enfocadas a la creación de programas de *machine learning.*

3.2.1 Transformaciones geométricas y espaciales

Son aquellas en las que se modifica la imagen a nivel píxel, designando una nueva posición de los píxeles en la matriz de la imagen, modificando la geometría y la posición de los elementos en las imágenes. Entre las usadas se encuentran:

- *Translate X/Y*
- *Rotate*
- *Flipr*
- *Flipud*

Otra transformacion utilizada para terminar de enriquecer la variabilidad del dataset ha sido el Desenfoque de tipo Gaussiano.

3.3 Formato HDF5

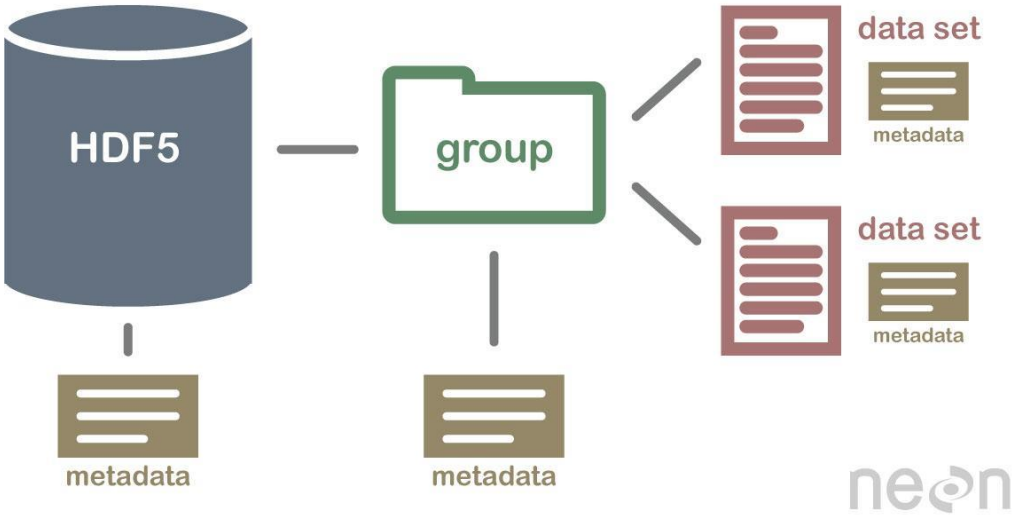
El código del proyecto en su mayoría está escrito en Python [12], un lenguaje creado a principios de la década de los 90 por Guido van Rossum en el centro Stichting Mathematisch Centrum, en los Países Bajos. Surgió como sucesor de varios lenguajes de programación existentes.

Se escoge Python [13], como lenguaje de programación frente a Matlab, por la sencillez de uso, el gran auge de su propia comunidad , y ser un lenguaje de programación que gracias a la existencia de sus numerosos paquetes resulta muy útil en la tarea de trabajar e implementar el *deep learning*.

El objetivo del script trabajado ha sido la de aumentar y balancear una base de datos de imágenes originales junto con sus respectivas máscaras de segmentación de entrada, con un tamaño de 160 x 160. Mediante una serie de modificaciones y transformaciones aplicadas, se consigue ampliar el dataset original y finalmente almacenarlo en formato HDF5 [14].

Hierarchical Data Format o HDF5 [15], es un formato de archivos de código libre para almacenar gran volumen de datos complejos y heterogéneos. Muy usado como almacenamiento de *big data*.

Organiza los datos de una manera similar a emplear directorios en el ordenador y permite en cada directorio tener tipos distintos de datos. Como particularidad adicional, permite añadir información con la inserción de metadatos, como por ejemplo la ruta de grupos o directorios que sigue el archivo HDF5 o el tipo de archivos que se encuentra en su interior.



[16]

Figura 11 – Arquitectura de formato HDF5

El dataset obtenido será utilizado en la parte de entrenamiento, validación del modelo obtenido de la red neuronal convolucional XNET y posteriormente en la parte de pruebas del modelo entrenado.

3.4 Implementación del código

A lo largo de esta sección se pretende desarrollar la organización del código así como dar una definición del uso empleado para cada script que conforman la parte de aumento de base de datos. No obstante, se detalla en más profundidad todo los parámetros y ejecución por partes del código en el anexo.

3.4.1 Jerarquía de directorios

Para simplificar la detección de posibles errores y facilitar el manejo del código, se han creado distintos archivos de programación, conteniendo cada uno partes diferenciadas para cumplir con el objetivo del apartado 3.

La carpeta contenedora es llamada *Augmentation*, y contiene cuatro archivos de código, entre ellos el más importante es *augmentation.py*, ya que se puede considerar el main del conjunto.

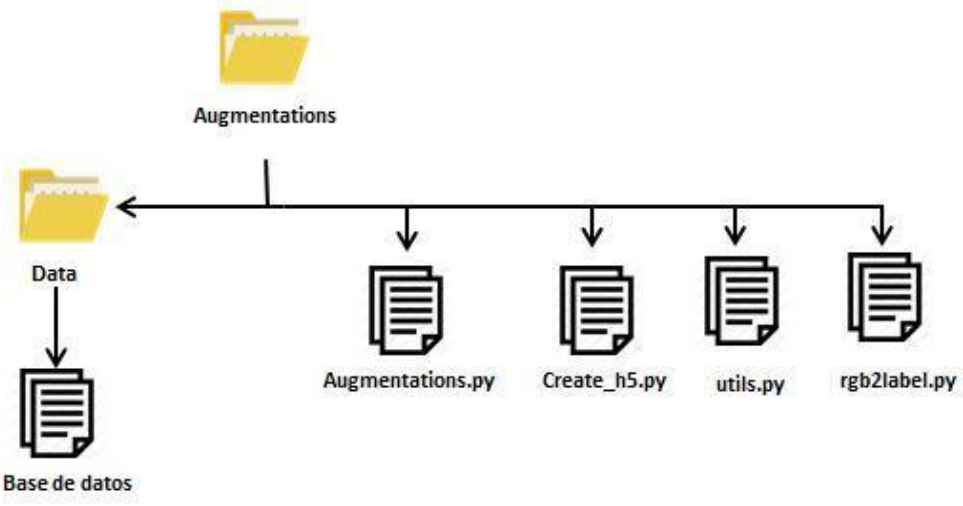


Figura 12 - Jerarquía de directorios de la parte de aumento de datos

Como se puede observar en la figura 12, se describe la jerarquía de directorios y archivos completos que componen esta parte del proyecto. Data hace referencia a la carpeta contenedora de la base de imágenes médicas de psoriasis destinadas a ser aumentadas. La organización de la carpeta Data viene descrita en el subapartado creación del apartado base de datos de este documento.

A continuación se detalla a la finalidad de cada fichero de programación:

- **Augmentations.py:** Este código de python se encarga de preprocesar imágenes médicas para su uso en el entrenamiento de modelos de aprendizaje automático. Realiza tareas como dividir los datos en conjuntos de entrenamiento, validación y test, aplicar un aumento de los datos mediante el uso de transformaciones, para enriquecer el conjunto de entrenamiento, y finalmente, crear un archivo HDF5 para almacenar las imágenes y máscaras de segmentación procesadas. El objetivo es preparar los datos para entrenar la red convolucional.

- **Create_h5.py:** Contiene una serie de funciones utilizadas para escribir datos en un archivo HDF5. Los datos incluyen imágenes, máscaras de segmentación y nombres de archivo, que se dividen en tres conjuntos de entrenamiento, test y validación. Los datos se almacenan en el archivo HDF5 con atributos y conjuntos de datos específicos para cada tipo de información. Antes de escribir los datos, realiza algunas operaciones como la normalización de las imágenes entre otros. Luego, guarda los datos en el archivo HDF5.

- **Utils.py:** se describen cuatro funciones en su interior utilizadas tanto en la parte de aumento de datos como en la de entrenamiento. Para esta parte en concreto se utiliza la función *balanced_test_val_split()***,** esta función tiene como objetivo conseguir un equilibrio en la cantidad de datos de cada clase en el conjunto que se usará en la creación del modelo entrenado de la red convolucional, para reducir los problemas relacionados con el overfitting.

- **rgb2label.py:** Se realiza una serie de tratamientos a las imágenes y sus máscaras de segmentación para conseguir a nivel pixel informaciones de colores y clasificación de los mismos que componen la base de datos, realizándose un etiquetado.

3.4.2 Resultados de ejecución del programa

Una vez ejecutado el script *augmentation.py* se obtiene el dataset conformado en formato HDF5 listo para ser utilizado en el entrenamiento y validación del modelo.

A Continuación se muestra un ejemplo de ejecución del programa en Anaconda, que es una de las distribuciones de existencia de Python.

```
Python 3.7.7 (default, Mar 23 2020, 17:31:31)
Type "copyright", "credits" or "license" for more information.

IPython 7.22.0 -- An enhanced Interactive Python.

In [1]:         '/Users/TedyRobin/Desktop/Augmentations/
augmentations.py'       ='/Users/TedyRobin/Desktop/Augmentations'
Using TensorFlow backend.
/Users/TedyRobin/opt/anaconda3/envs/XNETT/lib/python3.7/site-packages/
tensorflow/python/framework/dtypes.py:516: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future
version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])

Comprobando si las mascaras de segmentacion y los datos coinciden en
la carpeta CP ...
Comprobando si el numero de mascaras de segmentacion coincide con el
numero de archivos de imagenes de datos...
Los nombres de las mascaras y los datos en la carpeta CP coinciden
perfectamente, se encontraron 51 imagenes.
Comprobando si las mascaras de segmentacion y los datos coinciden en
la carpeta GUTATE ...
Comprobando si el numero de mascaras de segmentacion coincide con el
numero de archivos de imagenes de datos...
Los nombres de las mascaras y los datos en la carpeta GUTATE coinciden
perfectamente, se encontraron 26 imagenes.
/Users/TedyRobin/opt/anaconda3/envs/XNETT/lib/python3.7/site-packages/
rasterio/__init__.py:219: NotGeoreferencedWarning: Dataset has no
geotransform set. The identity matrix may be returned.
  s = DatasetReader(path, driver=driver, sharing=sharing, **kwargs)
(Categoria b'psoriasis-') procesando imagen 46/47, aumentando imagen
21/21Imagen con Problema
(Categoria b'psoriasis-') procesando imagen 14/15, aumentando imagen
20/20Finalizando el aumento del Dataset
(1034,)
Guardando el archivo hdf5 en CP_GUTATE_s160_a1000.hdf5 ...


Figures now render in the Plots pane by default. To make them also
appear inline in the Console, uncheck "Mute Inline Plotting" under the
Plots pane options menu.


In [2]:
```

Figura 13 – Ejecución de Anaconda del aumento de datos

Se observa, como antes de iniciar el proceso de aumento, de la parte de entrenamiento y la parte de validación, se hace la comprobación de que, tanto los directorios de imagen y máscaras de segmentación de CP (*Chronic Plaque*) como de *guttata*, coinciden en número de imágenes. Al igual que tienen nombres cada par de imagen/mascara, entre otras comprobaciones.

Después realiza los aumentos de cada parte del dataset correspondiente. Por último guarda todas estas imágenes transformadas en su correspondiente dataset en formato HDF5.

4 SEGMENTACION USANDO DEEP LEARNING

Una herramienta no es más que la extensión de la mano de un hombre, y una máquina no es más que una herramienta compleja. Y quien inventa una máquina aumenta el poder del hombre y el bienestar de la humanidad.

- Henry Ward Beecher -

El proceso de la segmentación automática que se desarrolla en el TFG, se realiza mediante la aplicación de las técnicas conocidas como *deep learning*. Para desarrollar la identificación de imágenes y otro tipo de tareas automatizadas, se construyen unos modelos computacionales capaces de entrenar al ordenador a reconocer patrones y realizar predicciones acertadas. A estos modelos se les conoce como redes neuronales.

Las redes neuronales son una herramienta fundamental que permiten, en el ámbito de la computación, un avance en procesamiento y reducción de complejidad en ciertas problemáticas surgidas a lo largo del tiempo, el objetivo de este proyecto es usarlas para poder realizar tareas de clasificación y segmentación de manera más sencilla y automatizada.

Su funcionamiento se basa análogamente en el del cerebro humano, estableciendo periodos de aprendizaje y periodos de validación y de memorización de los patrones de generalización obtenidos en el primer periodo nombrado. Se conforman de capas de nodos, que siguiendo con la analogía expuesta se comparan a las neuronas.

Existen varios tipos de redes neuronales artificiales según su complejidad (básicas o redes neuronales profundas) o dependiendo de si son con o sin retroalimentación CNN frente a las RNN.

Una breve aclaración es que para que una DNN se pueda considerar como tal, debe de contar con una arquitectura con más de tres capas de nodos, si no solo se denomina red neuronal básica.

Para cumplir con el objetivo del proyecto, se hará uso de las redes neuronales del tipo convolucionales.

4.1 Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales son un tipo de arquitectura de red neuronal diseñada para analizar imágenes y reconocer patrones visuales. Su nombre proviene de la operación matemática clave que utilizan para ser desarrolladas, conocida como "convolución". Las CNN son ampliamente utilizadas en aplicaciones de visión y como algoritmos de *deep learning* si son lo suficientemente complejas debido a su eficacia [17] [18].

Estas redes constan de varias capas, incluyendo capas de convolución y agrupación, que trabajan juntas, para extraer características significativas de las imágenes, mediante herramientas de convolución.

Estas características se utilizan para clasificar las imágenes y realizar tareas de procesamiento de imágenes. La idea central detrás de las CNN es reducir la complejidad de los datos al eliminar detalles innecesarios y resaltar patrones importantes.

En resumen, las CNN son una herramienta fundamental en la visión por computadora que permite a estas comprender y procesar imágenes de manera efectiva.

4.1.1 Estructura de las Redes Neuronales Convolucionales

Generalmente, las redes neuronales convolucionales trabajan mediante una composición de capas, cada una con su funcionamiento, interconectadas entre sí, como se ha introducido en la sección anterior. En este apartado se desarrollan varios tipos de capas y partes que normalmente componen la arquitectura CNN, para su mejor comprensión en futuros apartados.

Codificador

El codificador [19] es una parte fundamental de un sistema de procesamiento de datos. Su tarea principal consiste en extraer las caracterisitcas que definen los datos de entrada, como una imagen en este caso, utilizando capas de agrupación y convoluciones. En este proceso se transforma la entrada en una representación más compacta y significativa.

Decodificador

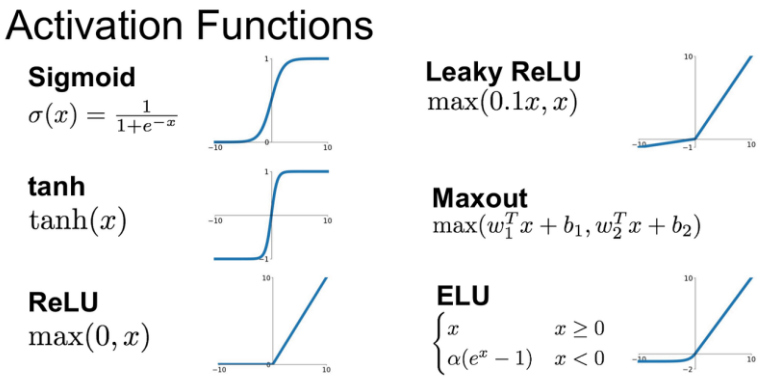
El decodificador es la contraparte del codificador y punto clave en la recuperación de la información original. Su función es aumentar partiendo de las caracteristicas conseguidas, los datos codificados para restaurarla al tamaño y formato originales de la entrada. Esto se logra mediante la aplicación de técnicas de aumento de resolución (*up-sampling*) y convoluciones, permitiendo así la recuperación de la información original a partir de la representación compacta generada por el codificador.

Capa de activación

La capa de activación es una de las capas más importantes que conforman la CNN, se aplica en cada neurona a la salida. Esta es la responsable de que la CNN pueda crear patrones y relaciones entre los datos. En ella se crean relaciones no lineales entre los datos de entrada y de salida, para que la red neuronal pueda abordar cada vez una mayor complejidad.

Existen varios tipos de funciones de activación, como pueden ser la función escalón, la función tangente hiperbólica, la función sigmoidal o la función rectificado o comúnmente conocida como RELU.

RELU o Función Rectificadora será la empleada para activar XNET, la CNN empleada en el proyecto, y es una de las más populares en el campo de las redes neuronales. Se encarga de eliminar los datos negativos de entrada, dándoles un valor nulo, solo devuelve los valores positivos.



[20]

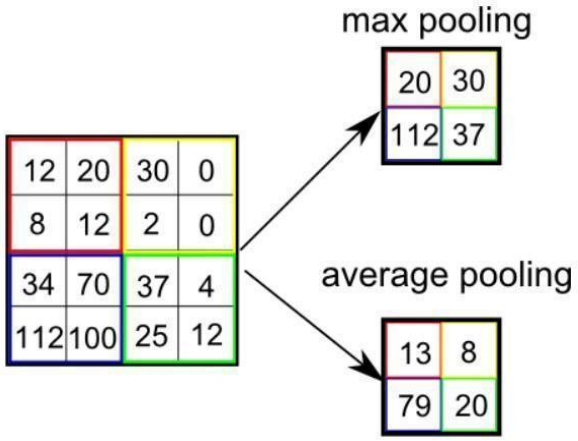
Figura 14 – Funciones de Activación

Pooling layer

Pooling layer estudia el contenido de la imagen por regiones, para obtener la información que mejor representa cada sección.

Con esta reducción se consigue obtener una minimización en la potencia necesaria de procesamiento, aligerando la carga del ordenador con todas las ventajas que conlleva, generalizando aún más las características obtenidas de los datos introducidos al segmentar.

El *max pooling*, realiza una división de la imagen en regiones. Devuelve el valor máximo de la sección que se trabaja en el momento y cuenta con una mayor protección frente al ruido.



[21]

Figura 15 – Maxpooling

Batch normalization

La normalización por lotes, es utilizada para regular el modelo estableciendo una media cero y varianza 1 a la entrada de cada neurona. Se aplica cada vez que se desarrolla la convolución y antes de la activación.

Con esta capa se obtiene una serie de mejoras sustanciales en el modelo en la parte de entrenamiento, haciendo este más robusto, con un aprendizaje más veloz y menos problemas derivados en sobreajuste.

Dropout

El *Dropout* es una técnica, que establece de forma aleatoria, la desactivación de un cierto número de neuronas. Desconectando dichas neuronas, se consigue que no aporten ayuda en la obtención de datos a las neuronas cercanas. Al eliminar una parte de información y no tenerla en su totalidad, se fortalece a las neuronas activas en la generalizacion de patrones.

Llegando a parecer contradictorio por el hecho de que se elimina una parte de información, es una clave importante para evitar el *overfitting*.

Capa de Clasificación

Tras el aprendizaje del modelo se establece en la última capa de la CNN una capa de clasificación. Esta es la encargada de, una vez obtenida todas las características de los datos, crear un vector con todas sus respectivas probabilidades de imagen de entrada.

Hay dos tipos definidos de capas de clasificación, cada una con sus ventajas y desventajas.

La denominada completamente conectada o abreviada FC, conecta todas las partes de la capa anterior con la capa siguiente. Para producir el vector de probabilidades, se conecta a una activación del tipo *softmax*,

encargada de dar el valor de dichas probabilidades. Este tipo de clasificación es muy buena para aprender fronteras de decisión complejas, pero como necesita de muchos parámetros, puede llevar al sobreajuste.

La capa GAP o promedio global, que surge como solución a los problemas de sobre ajustes y reducción de parámetros del anterior tipo. Su objetivo se obtiene al hacer un valor promedio de cada clase característica. Su mayor desventaja es que reducen también su aprendizaje de las características más complejas.

4.2 Tipos de Redes Neuronales Convolucionales

Tras realizar una introducción de las redes convolucionales y explicar su funcionamiento interno, cabe destacar que el desarrollo de las redes convolucionales desde cero es una tarea muy compleja.

Existen una gran diversidad de redes neuronales desarolladas según el número de capas nodales a usar, los tipos de conexiones entre ellas o las finalidad con la que se puede desarrollar.

Actualmente, se trabaja con alguna de las redes neuronales ya existentes que hayan sido probadas y demostrada su alta capacidad de acierto y precisión.

En el proyecto se utilizaran dos redes neuronales diferentes, no obstante, la explicación se centrará en tres de las CNN más populares en el ámbito de segmentación y clasificación de imágenes, que más se podría asemejar a la finalidad del trabajo.

Aún así existen diversas investigaciones que tratan el tema con más profundidad. En el articulo '*Review of deep learning: concepts, CNN architectures, challenges, applications, future directions'* de los autores Laith Alzubaidi, Jinglan Zhang y Amjad J. Humaidi publicado en 2021 [22], se proporciona una revisión exhaustiva sobre el *deep learning*, sus aplicaciones en diferentes dominios, además de los avances en técnicas y redes como las CNN

4.2.1 XNET

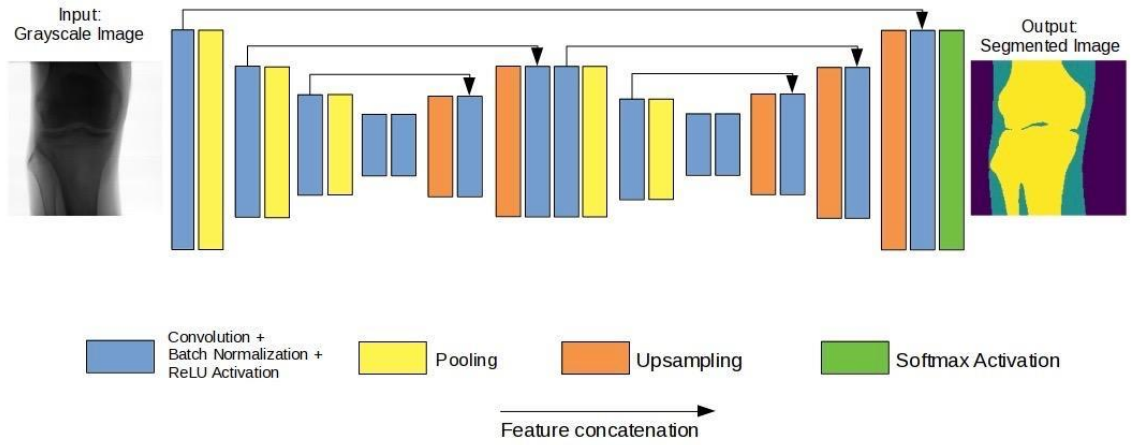
Es la red neuronal principal que ha sido utilizada en la elaboración del proyecto.

Es una red neuronal convolucional creada para la segmentación de imágenes de carácter médico, sobre todo en los últimos trabajos ha sido utilizada en el tratamiento de imágenes de rayos X, diferenciando regiones óseas y tejidos blandos. Se ha consolidado como una CNN robusta y eficaz en estos objetivos.

Como punto principal la diferencia al resto de CNN, es que está pensada para entrenarla con pequeñas bases de datos, un punto fuerte es su reducción de falsos positivos en regiones de tejidos blandos.

Dado que no se cuenta con una base de datos muy amplia, como se comentó en el apartado uno, el uso de XNET ayuda a reducir el impacto de dicho punto débil del proyecto, siendo por lo tanto el motivo principal de seleccionar esta CNN.

Al igual que todas las CNN ya definidas hasta ahora en dicho ámbito, XNET cuenta con una estructura codificador-decodificador.



[23]

Figura 16 - Arquitectura de XNET

Una de las características diferenciadoras de XNET con respecto a otras redes neuronales convolucionales en su arquitectura, es el incremento del número de etapas de reducción de tamaño para extraer las características que componen a las imágenes.

Para lograrlo se establece una configuración codificador-decodificador en sucesión, guardando los mapas de características del codificador con su posterior uso en los decodificadores para la creación de mapas de características más densas.

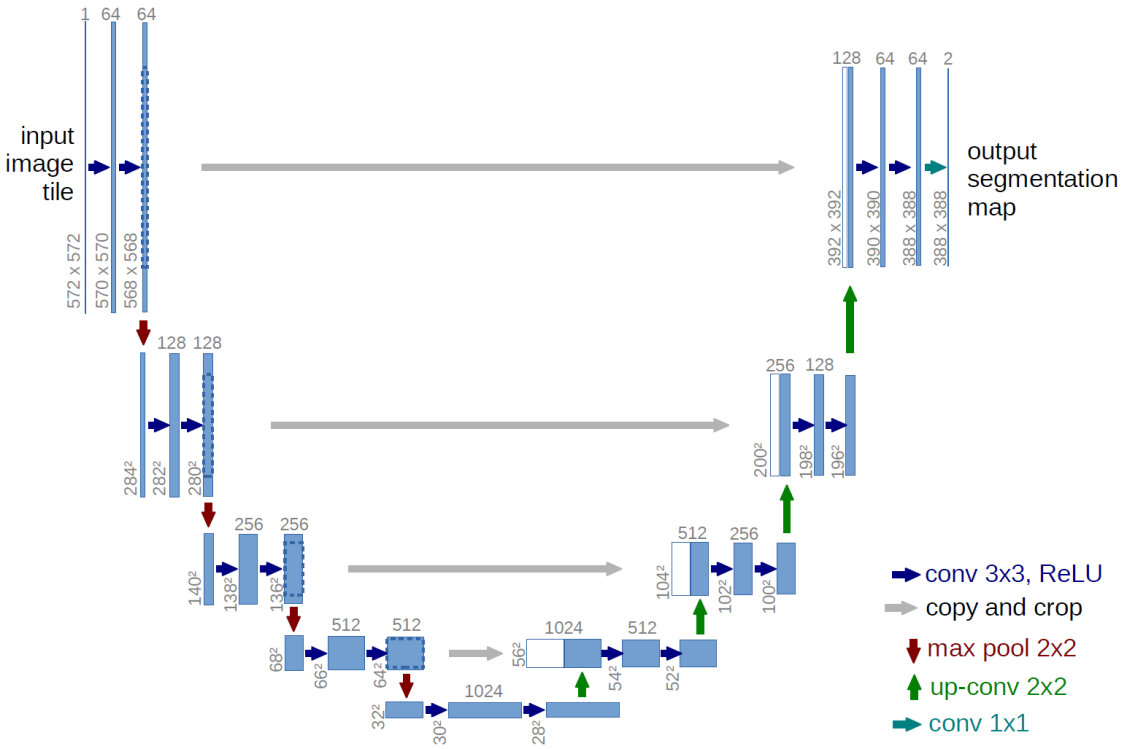
Cada capa convolucional utilizada en su estructura se acompaña de capas de activación de tipo ReLu y el decodificador aumenta el tamaño de la imagen utilizando el método de vecinos más cercanos.

El procesamiento y aprovechamiento de los mapas de características generados por el codificador se efectúa a través de la transferencia de filtros entre capas que tienen dimensiones equivalentes. Esto implica que el modelo tiene una menor probabilidad de desprender la información que ha adquirido previamente. En consecuencia, se reduce la posibilidad de perder detalles precisos una vez que la imagen o datos han sido comprimidos. Esta técnica de compartir filtros preserva la información importante y ayuda a mantener la calidad de la representación original.

4.2.2 UNET

Desarrollada por Olaf Ronneberger, Philipp Fischer, Thomas Brox en el año 2015, recibiendo varios premios.

UNET [24], es una de las arquitecturas de CNN con un gran impacto para la segmentación de imágenes, en concreto en el ámbito de la biomedicina. Siendo una de las más precisas y completas, ya que cuenta con un gran nivel de conectividad entre sus capas.



[25]

Figura 17 - Arquitectura de UNET

Como se puede observar en la imagen descriptora de su arquitectura es una CNN con estructura del tipo codificación-decodificación. Las capas de color azul hacen referencia a una capa multicanal. En la parte izquierda se refleja la parte codificadora, en la que se comprimen las imágenes mediante una serie de convoluciones y tratamientos, bajando cada vez a una capa de menor nivel, obteniendo un mapa de características de las imágenes de entrada.

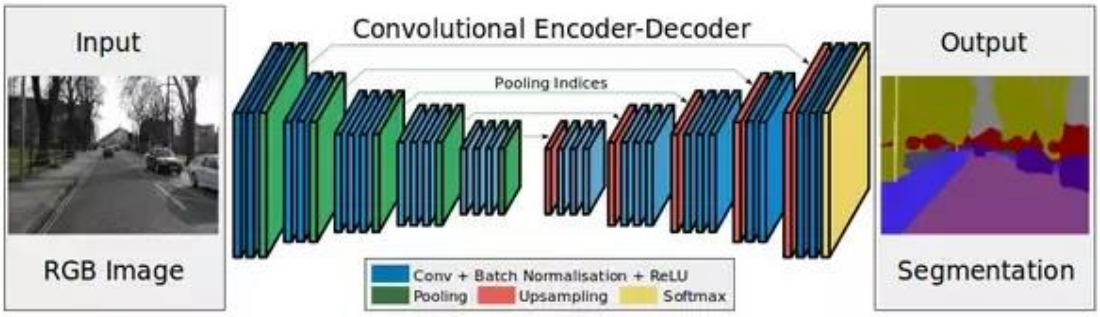
Una vez llegue a la capa más pequeña pasará a su parte de decodificadora que realizará operaciones complementarias a las aplicadas en la parte de codificación obteniéndose la segmentación semántica de la imagen.

4.2.3 SEGNET

SEGNET [26], es una red neuronal convolucional creada por Vijay Badrinarayanan, Alex Kendall, Roberto Cipolla en el año 2015, enfocada a la segmentación semántica de imágenes.

Al igual que UNET es una CNN creada con una arquitectura codificador-decodificador. Su parte codificadora cuenta con 13 capas convolucionales, siendo idéntica en composición a su antecesora, la red CNN VGG16.

Su diferencia innovadora se encuentra en la decodificación, en la que se amplían sus mapas de baja resolución, mediante el uso de los índices de agrupación o *pooling*, obtenidos en la parte del codificador. Teniendo en cuenta esta reutilización de parámetros, no existe la necesidad de conseguir un aumento de resolución. Consiguiendo una gestión eficiente de recursos como la memoria y la minimización de tiempos computacionales.



[27]

Figura 18 – Arquitectura de SEGNET

5 RESULTADOS EXPERIMENTALES

La ciencia más úti es aquella cuyo fruto es el más comunicable.

- Leonardo Da Vinci -

El entrenamiento de las redes neuronales explicadas en el punto anterior da consigo una serie de modelos entrenados, estos son utilizados para realizar la tarea de segmentación. Estos modelos se obtienen tras la realización de varios experimentos ejecutando el código, en el proceso se realizan modificaciones de los parámetros y configuraciones iniciales para optimizar los resultados.

En este caso se han escogido tanto XNET como UNET, para desarrollar los modelos entrenados, con estos se realiza una comparativa del comportamiento y rendimiento de cada una, observando las diferencias que puedan existir en sus resultados y cuáles son sus ventajas o desventajas.

Para la comparativa de ambos modelos se utilizan tanto una serie de parámetros de medición como los resultados prácticos de la segmentación en un conjunto de test de imágenes, previamente desarrollado, localizado en el dataset creado anteriormente.

5.1 Parámetros para la medición del rendimiento

Para comprobar la robustez y fiabilidad del modelo entrenado, en el desarrollo de redes neuronales se utilizan una serie de métricas de rendimiento [28], para poder evaluar los resultados obtenidos durante el proceso y conocer si hay algún método de mejora.

Aunque existen diversos tipos de mediciones, a continuación se desarrollan las que han sido utilizadas en este caso en concreto.

Verdaderos Positivos

Aquellos casos que se predice que el valor es positivo y realmente lo es.

Verdaderos Negativos

El valor predecido como un valor negativo es realmente negativo.

Falso Positivo

Engloba casos de predicción de valores positivos y que en realidad no lo son.

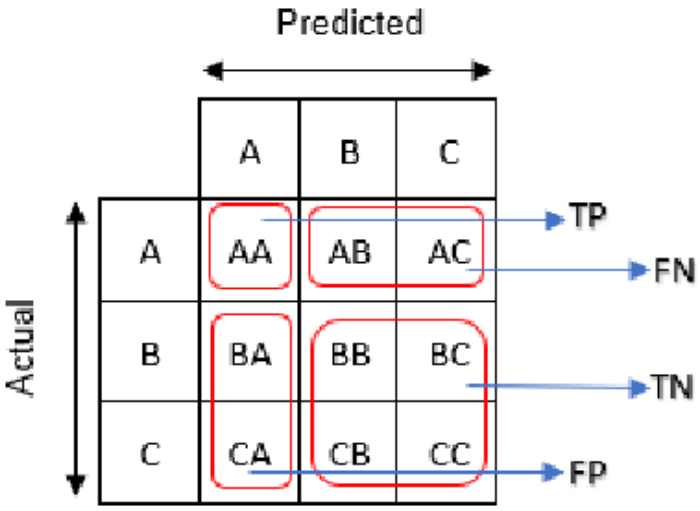
Falso Negativo

Casos que la predicción tomó como valor negativo y en la realidad el valor es positivo.

La matriz formada por los cuatro valores anteriores es conocida como matriz de confusión. Su estructura es una tabla con filas que indican el valor actual de cada clase y columnas que indican el valor predecido mediante el modelo generado por la red neuronal de cada clase correspondiente. Esta matriz es una herramienta esencial, aplicándola se consigue nuevos parámetros aplicados a la métrica del rendimiento.

Como en este proyecto se utilizan tres clases para segmentar las imágenes, la matriz de confusión será conformada por una dimensión 3 x 3. En la siguiente figura se muestra la conformidad de la matriz de confusión para la dimensión del proyecto, también se muestra la forma de hallar los valores de los anteriores

parámetros definidos, haciéndose referencia de sus siglas en ingles, TP (*true positive*), FP (*false positive*), TN (*true negative*) y FN (*false negative*).



[29]

Figura 19 – Matriz de Confusión de tamaño 3X3

A partir de la matriz de confusión se pueden obtener nuevos parámetros de medición.

Exactitud

Es un valor que especifica el porcentaje de acierto obtenido a la hora de realizarse una predicción. Se divide la suma de todos los casos verdaderos, tanto negativos como positivos, entre el número de casos totales.

$$\frac{(vp + vn)}{(vp + vn + fp + fn)}$$

Precisión

Al realizar una serie de mediciones repetidas, puede surgir una cierta variabilidad en los valores obtenidos. La precisión mide esa dispersión de los valores, cuanto menor sea la dispersión, aumentará la precisión.

$$\frac{vp}{(vp + fp)}$$

Sensibilidad

Indica el porcentaje de casos positivos que realmente han sido detectados en el uso del modelo.

$$\frac{vp}{(vp + fn)}$$

Especificidad

Porcentaje de los casos negativos que se han clasificado correctamente como tales.

$$\frac{vn}{(vn + fp)}$$

5.2 Resultados de las distintas Redes Neuronales

En la elaboración del proyecto se ha trabajado con dos redes neuronales, XNET y UNET. Inicialmente se deseaba comprobar el funcionamiento de una única red neuronal, pero en búsqueda de diferentes perspectivas de resultados, se ha optado por el uso de ambas para su posterior analisis y comparación.

Para la obtención de los resultados se ha trabajado con un dataset de psoriasis que incluye imágenes médicas de dos tipos, en concreto *chronic plaque* y *guttata*. Las imágenes que conforman la base de datos utilizan el espacio de color RGB. La base de datos ha sido dividida en tres partes con distintas proporciones.

Teniendo un total de 77 imágenes originales, la división de la base de datos antes de establecer el aumento ha sido la siguiente:

- Se ha destinado un 60% para la parte de entrenamiento, 47 imágenes. Tras varias ejecuciones se observa unos mejores resultados que estableciendo otros porcentajes distintos.
- Un 20% para la parte de validación, 15 imágenes de la base de datos original.
- De las 77 imágenes, 15 son para la parte de tests, que equivale al 20% de la base de datos original.

Una vez equilibrada la base de datos, se aplicaron los correspondientes aumentos a las partes necesarias. Las imágenes han tomado un tamaño de 160 x 160, como se indicó en el apartado creación de base de datos.

Durante el entrenamiento del modelo se han utilizado 20 épocas, las épocas indican el número de veces que se repite el entrenamiento. Una mayor repetición consigue un modelo mejor configurado, pero sobrepasar también el número de épocas puede dar problemas de *overfitting*.

A continuación se procede a evaluar con el mismo dataset el entrenamiento y rendimiento de cada red neuronal. Se usa para ello los parámetros de rendimiento, también se muestran imágenes prácticas de la segmentación del modelo generado. Para comparar las imágenes de las segmentaciónes obtenidas, se utiliza una *Ground Truth* creada de forma original, sin supervisión medica, pero teniendo en cuenta los conocimientos obtenidos a través de varios repositorios médicos, como Dermnet.org entre otros.

Finalmente, se realiza un pequeño análisis de las direfencias encontradas en la ejecución de XNET y UNET y valoración de las ventajas y desventajas de cada uno respectivamente. Para fortalecer el proyecto, se han añadido varios trabajos adicionales relacionados a la segmentación de psoriasis mediante CNN. Los resultados expuestos en estos trabajos serán comparados con los propios obtenidos, estableciendo las similitudes y diferencias que se puedan observar.

5.2.1 Resultados obtenidos mediante XNET

En este punto se procede a aportar los parámetros usados para evaluar la red XNET. Para obtener unos datos precisos, se ha entrenado la red neuronal 3 veces distintas, cada entrenamiento compuesto por 20 épocas como se comentó en la anterior sección, todo ello utilizando el dataset creado conforme se explicó en los apartados anteriores.

La siguiente tabla muestra la matríz de confusión para cada uno de los modelos ejecutados:

	Verdaderos Positivos	Verdaderos Negativos	Falsos Positivos	Falsos Negativos
1	0.89	0.81	0.11	0.18
2	0.88	0.87	0.12	0.11
3	0.88	0.84	0.12	0.15

Tabla 1 - Matriz de Confusión XNET

Los parámetros de rendimiento del entrenamiento de XNET son los siguientes:

	Exactitud	Precision	Sensibilidad	Especifidad
1	0.85	0.89	0.83	0.88
2	0.88	0.88	0.89	0.88
3	0.86	0.88	0.85	0.88
Media	0.87	0.88	0.86	0.88
Desviación Estándar	±0.01	±0.00	±0.02	±0.00

Tabla 2 - Parámetros de medición del rendimiento XNET

En la ejecución del entrenamiento se obtiene la curva de aprendizaje. En la figura se muestra, en color azul, como en la sucesión de épocas, tanto el error, como las perdidas de aprendizaje de la parte de entrenamiento van decreciendo en forma de una curva suave. Para reducir los picos observados en las curvas de validación, en color naranja, sería necesario una mayor cantidad de imágenes, de esta forma se suavizarían dichas curvas. En líneas generales se puede observar que la red neuronal obtiene un comportamiento deseado de aprendizaje.

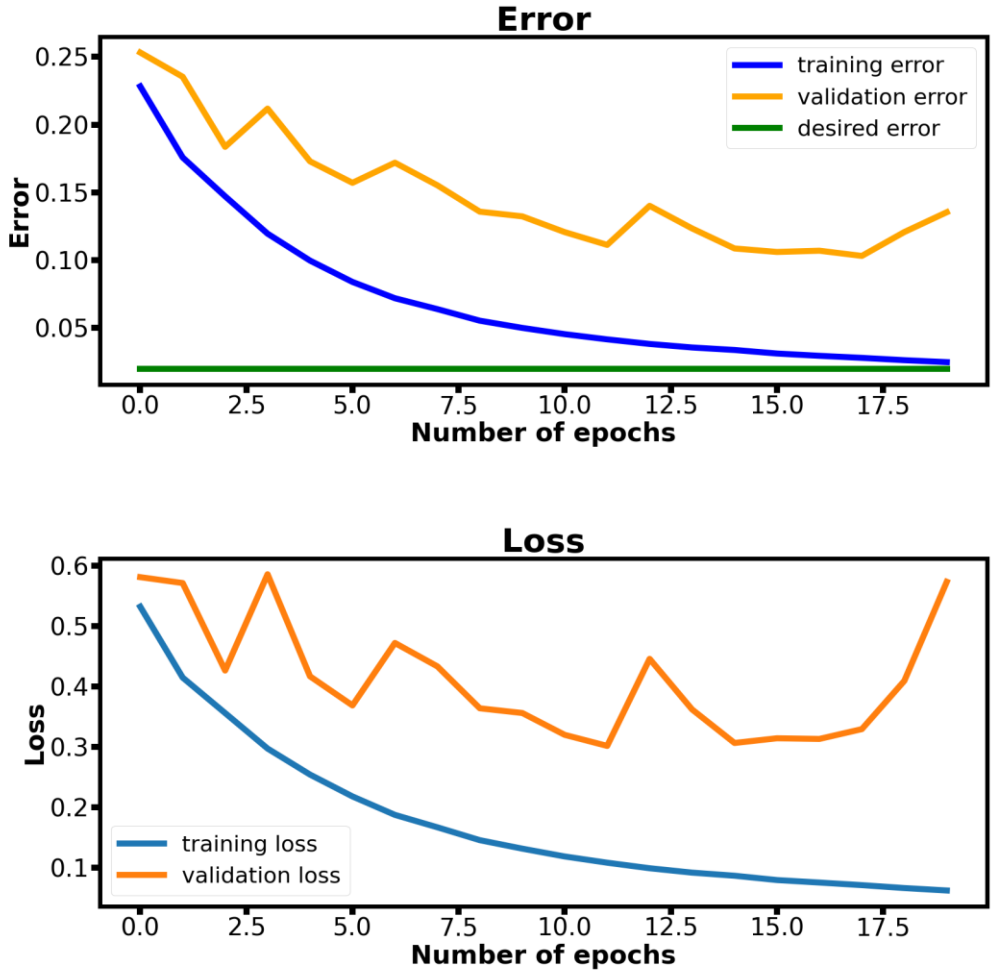


Figura 20 – Curva de aprendizaje de XNET

Como medida adicional del rendimiento, la siguiente figura muestra la gráfica de la Curva de Región de Convergencia, ROC [30]. Representa la comparativa de los falsos positivos y verdaderos positivos en función del umbral de clasificación.

Se oberva que las tres curvas que representan fondo, piel sana y psoriasis, se alejan de la diagonal y con una AUC, o área bajo la curva cercanas a un valor de 1, lo que indica que el modelo tiene un buen nivel de discriminación de cada clase. Por otra parte se aprecia que la curva con mejor resultado es la correspondiente al fondo en las imágenes, esto es debido a que el límite entre piel y fondo tiene un mayor contraste y es más fácil de discriminar.

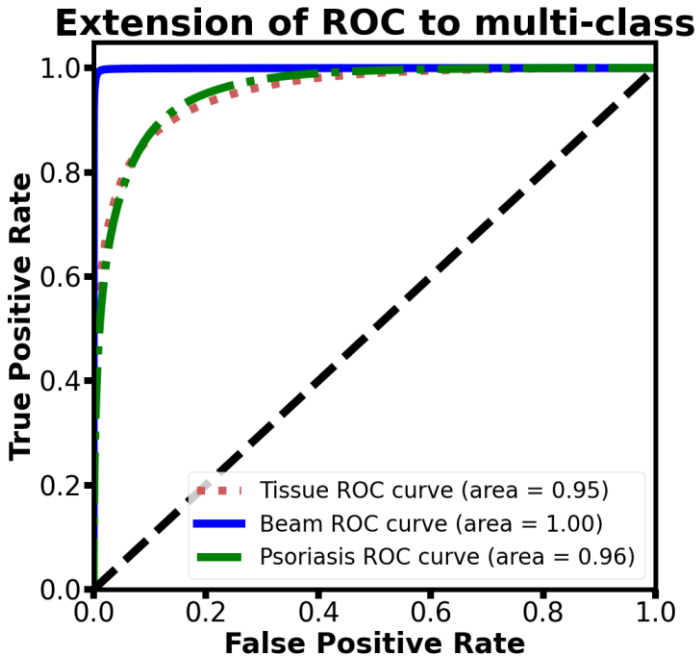


Figura 21 – Curva ROC de XNET

Para finalizar, se muestran una serie de ejemplos prácticos obtenidos con los modelos entrenados de XNET. Estas imágenes corresponden al entrenamiento número 2 ejecutado, dado que fue el modelo conseguido con una mayor exactitud en general.

Las dos primeras imágenes dentro de la figura 22, correspondiente a una psoriasis de tipo *chronic plaque*, muestra en líneas generales una segmentación bastante conseguida. Se puede apreciar que el fondo de la imagen con respecto a la piel, esta perfectamente diferenciado, además las placas de psoriasis está correctamente localizada.

Algo a destacar en ciertas partes del delineado de la psoriasis, es que en zonas del borde en la cual se encuentra menos marcada la placa o donde hay un menor contraste, el modelo lo toma como piel sana, dando lugar a un falso negativo. Añadir que en la figura 22, en la segunda imagen, se observa un error de segmentación en la separación entre los dedos del pie, provocados por la sombra que se crea en esa zona, el cambio de contraste se toma como una posible lesión, cuando esta no lo es.

Con Respecto a las imagenes 3 y 4 en las que se hay una psoriasis del tipo *gutatta*, al haber un menor numero de imágenes en el dataset de este tipo, al modelo le cuesta un poco mas realizar este tipo de segmentación.

Las lesiones se identifican correctamente en la posición en las que se encuentran en la imagen, pero solo localiza las zonas en las que se encuentran una mayor concentración de lesiones, realizando una agrupación del vecindario cercano, tomándolo como placas consistentes de un mayor tamaño y no dispersas.

Como conclusión el resultado obtenido en *chronic plaque* es más fiable y robusto que el de tipo *gutatta*, pero teniendo en cuenta el número de imágenes que componen el dataset, se ha obtenido un resultado suficiente.

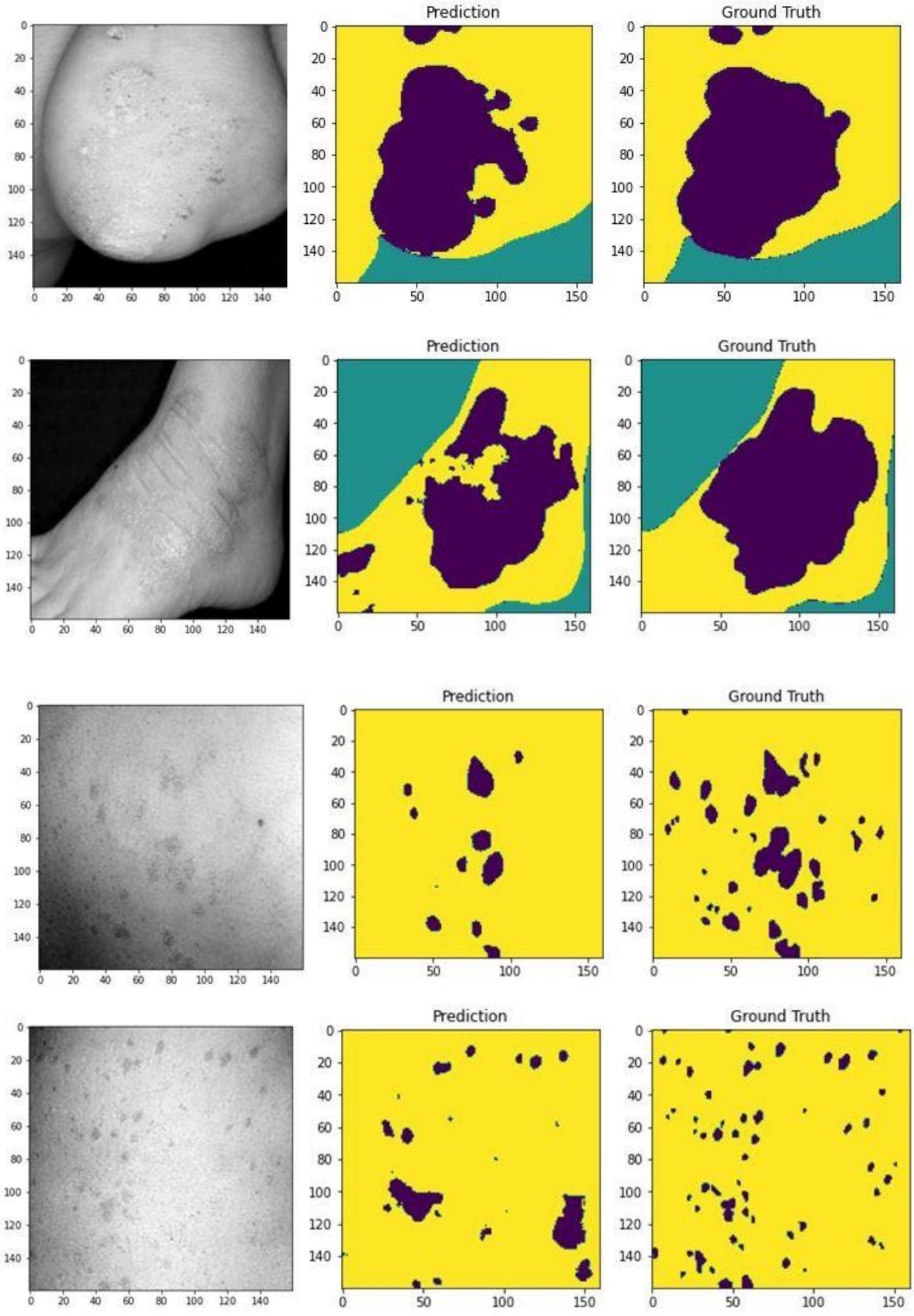


Figura 22 – Muestras, predicciones y ground truth de XNET

La precisión indica en qué partes de la imagen que se ha clasificado como psoriasis, existe verdaderamente dicha lesión en la piel. Las pérdidas son todos aquellos parámetros que no se han captado. Como se puede observar en la gráfica la precisión del entrenamiento es mucho mayor que en la validación debido a la proporción de las imágenes con las que cuenta cada una de las partes. En total se ha conseguido una precisión de 88% aproximadamente.

5.2.2 Resultados obtenidos mediante UNET

Al igual que en el caso anterior, el modelo ha sido entrenado 4 veces, utilizando el mismo dataset. Para una correcta comparación de ambas redes neuronales, se ha utilizado el mismo procedimiento de ejecución y configuración de parámetros.

A continuación se procede a mostrar las tablas de métrica obtenidas con UNET. La tabla 1, muestra la matriz de confusion para cada uno de los modelos ejecutados:

	Verdaderos Positivos	Verdaderos Negativos	Falsos Positivos	Falsos Negativos
1	0.91	0.68	0.09	0.30
2	0.88	0.82	0.12	0.17
3	0.88	0.84	0.12	0.14
4	0.88	0.86	0.12	0.12

Tabla 3 – Matriz de Confusión UNET

Los parámetros de rendimiento del entrenamiento de UNET son los siguientes:

	Exactitud	Precision	Sensibilidad	Especifidad
1	0.80	0.91	0.75	0.88
2	0.85	0.88	0.84	0.87
3	0.87	0.88	0.86	0.88
4	0.88	0.88	0.88	0.88
Media	0.85	0.89	0.83	0.88
Desviación Estándar	±0.03	±0.01	±0.05	±0.00

Tabla 4 – Parámetros de medición del rendimiento UNET

Con respecto a XNET, una diferencia notoria en el apredizaje, es que UNET va obteniendo una mejora del entrenamiento mas lenta, tanto las perdidas y exactitud del entrenamiento como el periodo de validación son peores, no obstante el modelo, conforme se produce cada época va mejorando gradualmente.

En el modelo XNET esto no ocurre así, este comienza con un valor muy superior al de UNET, pero el modelo solo mejora hasta la época 10, a partir de ahí, el resultado del parámetro *val_loss* que hace referencia a las

pérdidas en el periodo de validación no mejora. Al no mejorar este parámetro, el código está programado para que no se guarden los cambios realizados en el modelo.

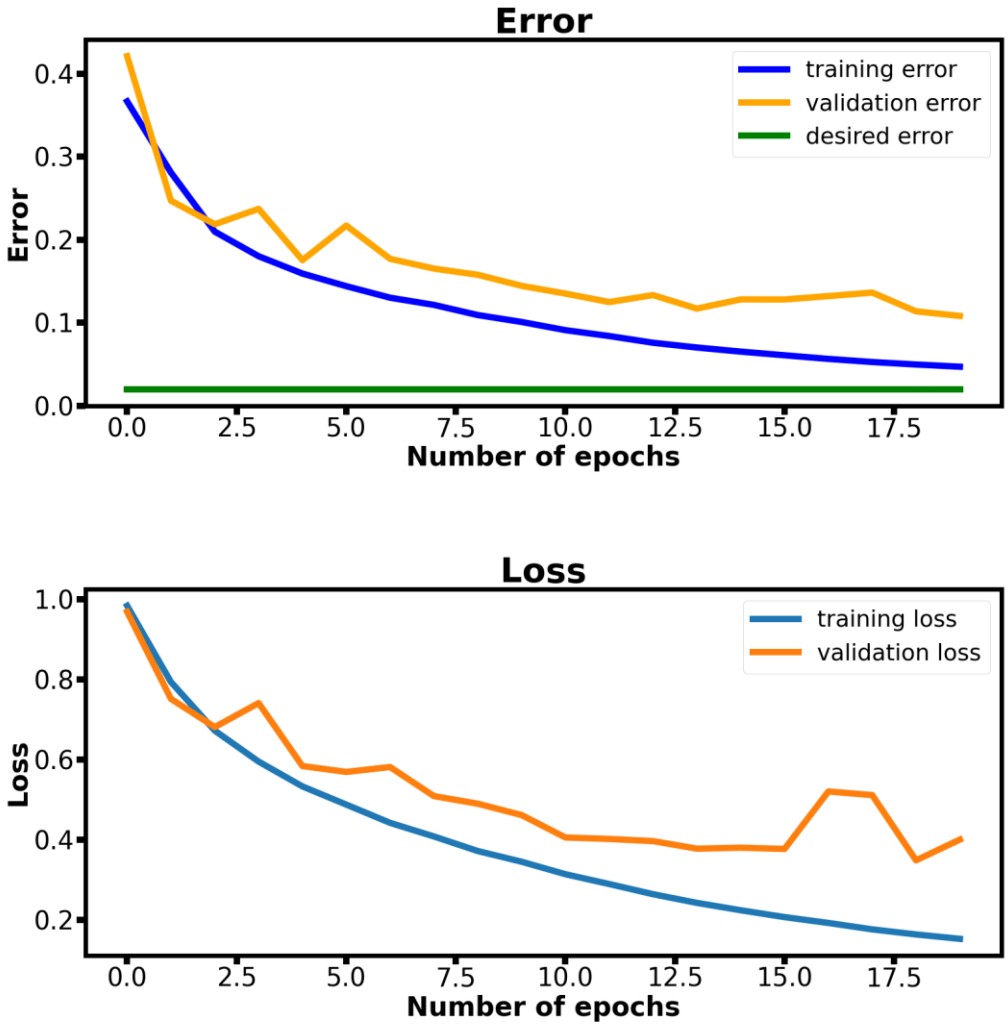


Figura 23 – Curva de aprendizaje de UNET

En la curva ROC no hay una gran diferencia con respecto a la conseguida en XNET. Lo más destacable es el empeoramiento de 0.01 en el área de psoriasis, pasando a tener un área de 0.96.

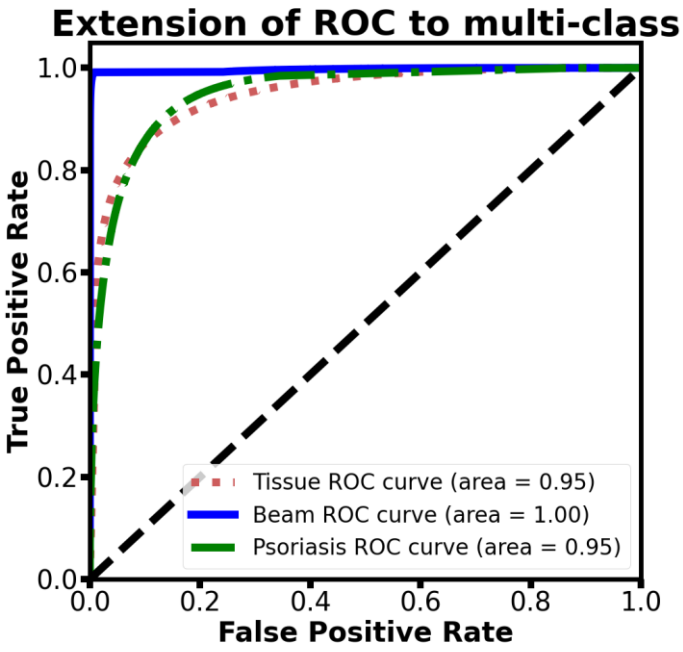


Figura 24 – Curva ROC de UNET

Al igual que en el caso anterior, se exponen una serie de figuras con casos prácticos de la segmentación realizada con el modelo entrenado, utilizando el modelo obtenido en la ejecución del código número 4, siendo el modelo con una mayor exactitud general.

En los dos primeros ejemplos de psoriasis *chronic plaque* se aprecian unos muy buenos resultados. El modelo localiza casi en su totalidad las distintas regiones a segmentar, cometiendo pequeños errores, como se pueden observar. En la primera predicción, la lesión que se localiza en la parte superior derecha no llega a ser encontrada. Otro error cometido a destacar es la perteneciente al segundo ejemplo, donde el modelo tiende a distorsionar la lesión en la parte superior del talón dando con ello un falso positivo.

En estos dos primeros casos de la figura 25, en comparación con el modelo de XNET, no se comete el fallo correspondiente al bajo contraste o regiones que por las características de las imágenes originales estan menos marcadas. Como punto negativo, en este modelo el límite de los bordes de cada región no se encuentra tan delineado y afinado como en XNET.

El caso 3 y 4 correspondiente a una psoriasis del tipo *guttata*, se aprecia un comportamiento del modelo muy parecido al modelo XNET. Llega a identificar zonas de lesión, pero solo en zonas que existen una agrupación de lesiones más pequeñas a muy cercana distancia. Al igual que la comparativa anterior, el delineado parece ser menos preciso, pero consigue abarcar una mayor región de acierto en lo que respecta a la lesión.

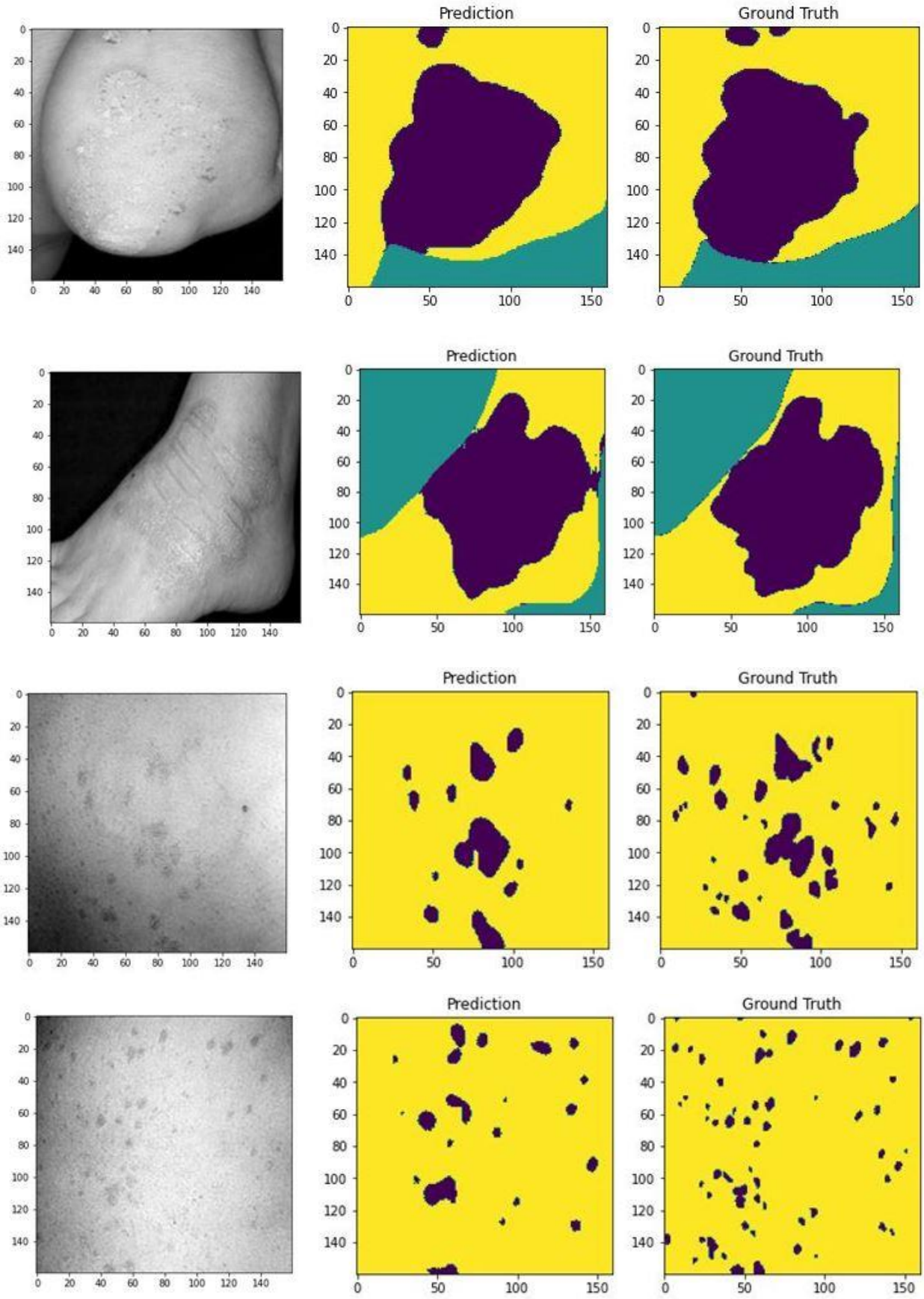


Figura 25 – Muestras, predicciones y ground truth de UNET

5.2.3 Comparación de resultados con otros estudios externos

Para validar y fortalecer los resultados obtenidos en este TFG, se han incorporado los resultados de dos artículos externos.

Primer artículo

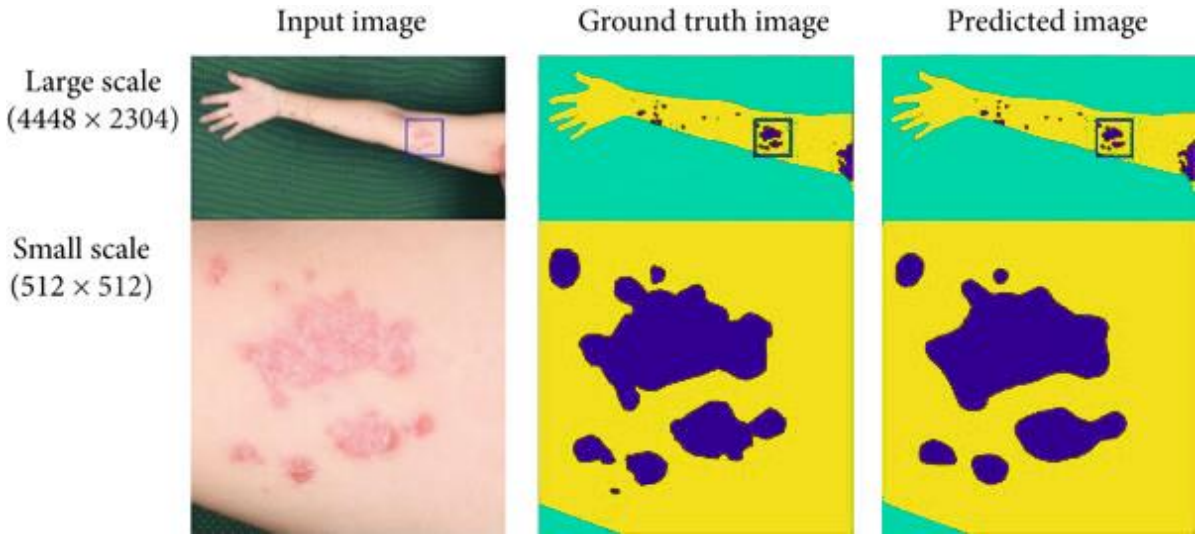
En el artículo '*Measurement of Body Surface Area for Psoriasis Using U-net Models'* [31], escrito por los autores Yih-Lon Lin, Adam Huang, Chung-yi Yang y Wen-yu Chang. Se utiliza la arquitectura UNET en la segmentación de imágenes *chronic plaque*, la segmentación es enfocada a imágenes de alta resolución.

En el trabajo se utiliza una base de datos compuesta por 255 imágenes de distintas zonas del cuerpo, cada imagen de un tamaño de 512 x 512. La división del dataset original para la ejecución del código es la siguiente

- 165 imágenes para entrenamiento
- 41 imágenes para la parte de validación
- Las 49 imágenes restantes son destinadas a la parte de tests

Aplicando el posterior aumento a las partes necesarias del dataset original y estableciendo un entrenamiento de 70 epocas, se obtuvo por parte del equipo una exactitud de 0.97 con una desviación estándar de ±0.07. El entrenamiento se repitió elevando el tamaño de las imágenes a 4300 x 2300 aproximadamente, en ese caso se apreció un aumento en el parámetro de la exactitud de casi 1, mejorando los resultados obtenidos.

A continuación se muestra los resultados prácticos obtenidos en el artículo:



[31]

Figura 26 – Muestras, ground truth y predicciones del artículo 1

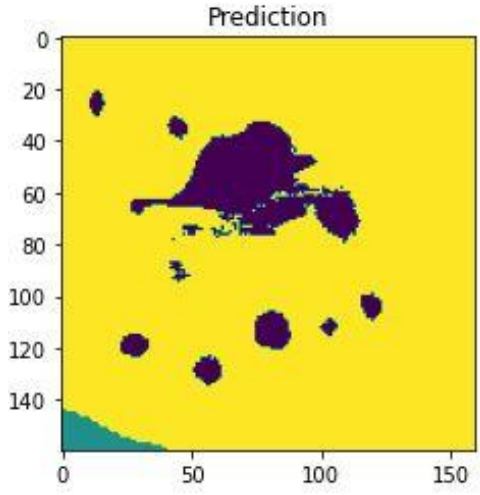


Figura 27 – Resultado UNET de este TFG aplicando la imagen del articulo 1

La figura 26 muestra los resultados prácticos obtenidos de la segmentación de distintos tamaños, correspondiente a una psoriasis de tipo *chronic plaque.*

En comparación con los visualizados en la figura 25, la figura 26 corresponde al tamaño de imagen 512 x 512, se observa una segmentación muy buena en líneas generales. Se aprecia que el fondo consigue diferenciarlo perfectamente y localiza con mínimos errores las distintas regiones a segmentar. Como diferencias destacables las placas de psoriasis tienen un mejor delineado en la figura 26, además se aprecia una corrección de las distorsiones de las lesiones, reduciendo el número de falsos positivos, como se mostraba en el talón del pie de la figura 25.

La figura 27 muestra el resultado obtenido aplicando el propio modelo de UNET de este TFG, a la imagen de entrada de la figura 26. Se aprecia tal y como se ha comentado anteriormente un peor delineado de las lesiones, incluso se puede observar, que lesiones como la situada en la parte inferior derecha de la segmentación en la figura 27, llegan a deformarse separándose en varias lesiones más pequeñas, dando a lugar un menor acierto.

En cuanto a los valores de exactitud de este estudio externo, con respecto al 0.85 de exactitud media obtenida en este TFG, se puede observar un aumento considerable del valor. Esto es debido a la diferencia en el número de imágenes que compone la base de datos y el tamaño de las mismas, ya que este estudio externo cuenta con una base de datos casi 4 veces superior y un tamaño de imagen de 512 x512 frente al 160x160 aquí utilizado.

Las 70 épocas durante el entrenamiento, compensan el aumento tanto de tamaño como el número de imágenes comentado anteriormente.

Segundo artículo

El segundo articulo '*Image segmentation with a convolutional neuronal network without pooling layers in dermatological disease diagnostics systems'* [32] publicado por Polyakova M. V., tiene como objetivo mejorar la segmentación de imágenes de lesiones de psoriasis mediante una CNN basada en la arquitectura UNET, sin implementar las capas de *Pooling layer*. Con ello pretende mejorar la segmentación de lesiones pequeñas y de formas complejas, como es el caso de las imágenes de psoriasis de tipo *guttata*. Los resultados indican una mejora en la segmentación de imágenes de psoriasis, especialmente en los bordes de las lesiones, aplicando el modelo UNET modificado, frente a la arquitectura UNET original.

La base de datos en este caso estaba compuesta por 50 imágenes originales de tamaño 256 x 256 de los tipos *chronic plaque* y *guttata*. Estas imágenes contenían únicamente áreas de piel sana y lesiones de psoriasis, sin incluir fondos.

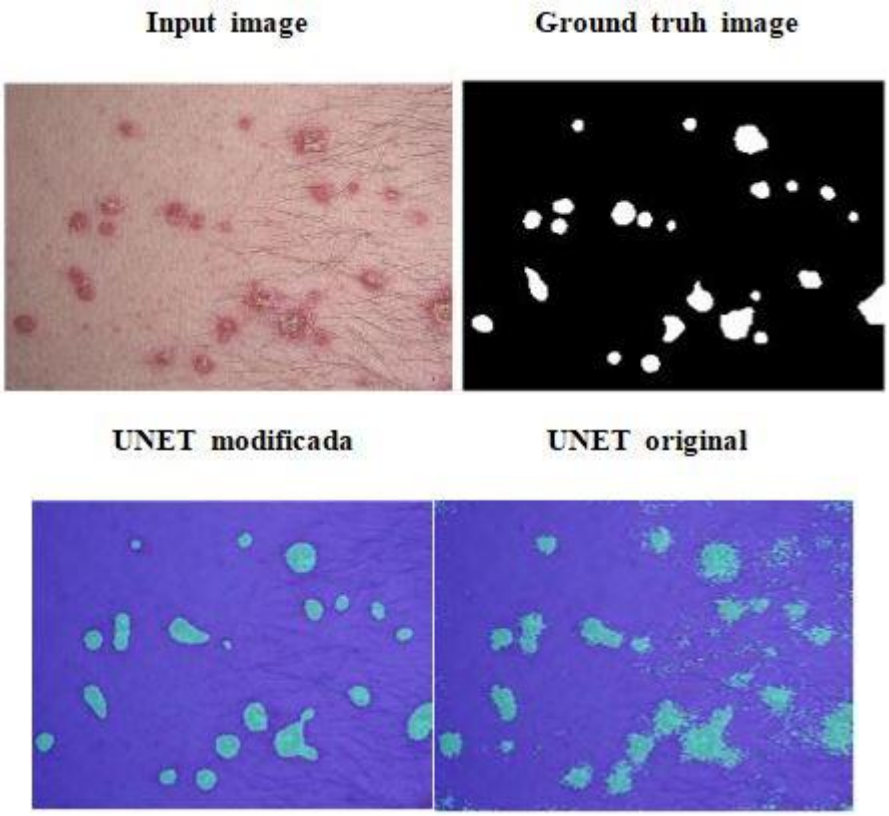
La ejecución de los modelos, a diferencia del primer articulo y este propio TFG, es que se hizo un

entrenamiento por separado para cada tipo de psoriasis. Centrando el entrenamiento en cada clase de lesión por separado, se consigue aumentar el rendimiento del modelo, llegando a obtener los siguientes valores de la exactitud:

- UNET modificado chronic plaque 0.99
- UNET modificado guttata 0.93
- UNET chronic plaque 0.98
- UNET guttata 0.90

El valor de la exactitud alcanzado, aplicando la separación comentada anteriormente para cada clase de lesión, demuestra que ayuda a elevar el nivel de rendimiento, al igual que queda consolidado que es mejor UNET modificado en lo que respecta a la segmentación de psoriasis.

Al igual que en el primer articulo, se procede a mostrar un ejemplo de segmentaciones prácticas obtenidas tanto por UNET original como el modelo modificado por Polyakova para el tipo *guttata*.



[32]

Figura 28 – Muestras, ground truth y predicciones del artículo 2

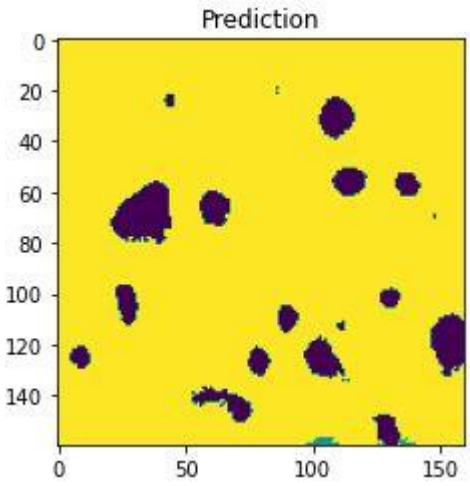


Figura 29 – Resultado UNET de este TFG aplicando la imagen del articulo 2

Como se puede observar, el delineado conseguido con la arquitectura modificada, así como la localización de todas las pequeñas lesiones está más perfeccionado que ejecutando el entrenamiento con UNET original. Otro punto a destacar es la reducción de ruido que se aprecia. En comparación al desarrollado en este TFG, UNET modificada detecta todas las lesiones por pequeñas que sean, en cambio en la figura 25, en lo que se refiere a los casos *guttata* se observa que no llega a detectar todas las regiones de lesión, en concreto, las más pequeñas. Puede deberse también al bajo contraste que existe entre las regiones de lesión y la piel sana.

Un punto en común que se aprecia es, el caso de que haya varias pequeñas lesiones cercanas con una leve separación. En ambos casos, tanto en la figura 25 como en la figura 28, se realiza una unión de todas ella en una lesión de mayor tamaño dando lugar a posibles errores de diagnósticos.

La figura 29 muestra el resultado obtenido aplicando el propio modelo de UNET de este TFG, a la imagen de entrada de la figura 28. Al comparar la segmentación de UNET modificada del segundo artículo, con la realizada en la figura 29 se confirma lo antes comentado. La mayoría de las lesiones en la figura 29 son detectadas y situadas, pero al encontrarse varias lesiones pequeñas muy próximas suelen formar una lesión única de mayor tamaño.

Estas comparaciones no solo subraya la coherencia en ciertos aspectos de las conclusiones posteriormente expuestas, sino que revela áreas clave donde se requieren ajustes y análisis adicionales.

5.3 Conclusiones

Una vez expuestas las métricas del rendimiento y otras gráficas características, así como comentado los resultados prácticos de ambas redes neuronales. Se puede llegar a las siguientes conclusiones.

- Con una exactitud entorno al 0.88 en ambas redes neuronales, se demuestra que son capaces de segmentar tanto el fondo de la imagen, como la piel y las zonas de psoriasis de una forma muy acertada.

- Tanto XNET como UNET, realizan una segmentación muy parecida en cuanto a precisión y acierto se refiere. XNET se caracteriza por delinear mejor las regiones, en cambio UNET es capaz de reducir los falsos negativos en zonas con poco contraste.

- Al haber descartado varios entrenamientos anteriores a los expuestos, se concluye que tanto el tamaño de las imágenes utilizados en la base de datos como la cantidad de estas, marcan una gran diferencia en los resultados finales del entrenamiento.

6 DISCUSIONES

Durante el desarrollo del trabajo, se han presentado desafíos significativos que han impactado en la eficiencia y la efectividad de las investigaciones.

La limitación de recursos de *hardware* y la disponibilidad de un conjunto de datos relativamente reducido, ha incrementado considerablemente la carga computacional y ha demandado una configuración más precisa.

El primer obstáculo que se ha encontrado ha sido relacionado con la tarjeta gráfica utilizada, una NVIDIA GTX 1070, creada en 2016. En la actualidad, esta tarjeta se considera relativamente lenta en comparación con las opciones más recientes. Esta limitación ha generado tiempos de espera prolongados al probar diferentes configuraciones y parámetros de ejecución, dificultando la optimización del modelo. La importancia del *hardware* utilizado es crucial, ya que cada entrenamiento o prueba ha requerido entre 7 y 15 horas seguidas.

Mejorar o actualizar el equipo podría reducir considerablemente estos tiempos y permitir una mejor optimización del modelo. Además, aunque se profundice en los conocimientos de redes convolucionales, el proceso de prueba y error para ajustar los parámetros sigue siendo un factor importante a considerar.

El segundo punto a comentar radica en el tamaño limitado del conjunto de imágenes originales. Se ha logrado corregir problemas de *overfitting*, aplicando un aumento de las imágenes de manera artificial y usando un tamaño de 160 x 160, más alta de la deseada en un principio. Pero lo ideal sería contar con una mayor cantidad de imágenes originales para mejorar aún más la generalización de los modelos. El aumento de datos y su resolución ha sido efectivo, pero con ello se ha incrementado considerablemente la carga computacional.

A pesar de estos desafíos, se considera que han sido cuestiones superables en el proceso de desarrollo de este proyecto. La experiencia ha permitido comprender la importancia de contar con recursos de hardware adecuados y la necesidad de acceder a conjuntos de datos más grandes y variados. Tras dedicarle tiempo y llevar a cabo diversas estrategias, se ha conseguido alcanzar resultados que se aproximan al objetivo deseado.

Como posibles mejoras a aplicar en un futuro, sería interesante, repetir los experimentos utilizando otros espacios de color como CIE L*a*b*. Tambien sería interesante, una vez conseguido el modelo entrenado, este pueda ser usado en un análisis en tiempo real y ver estos resultados en el mismo momento en el que se trata al paciente, reduciendo los tiempos de actuación. Añadir que al ser una herramienta tan potente, no solo es aplicable en el campo de la medicina, si no que puede ser expansible a otros ámbitos y usos.

BIBLIOGRAFÍA Y REFERENCIAS

[1] «Ventajas y desventajas de la inteligencia artificial», Inesdi. Accedido: 4 de junio de 2024. [En línea]. Disponible en: https://www.inesdi.com/blog/ventajas-y-desventajas-de-la-inteligencia-artificial/

[2] A. Menter *et al.*, «Joint AAD-NPF guidelines of care for the management and treatment of psoriasis with biologics», *J. Am. Acad. Dermatol.*, vol. 80, n.º 4, pp. 1029-1072, abr. 2019, doi: 10.1016/j.jaad.2018.11.057.

[3] «DermNet | Dermatology Resource». Accedido: 1 de junio de 2024. [En línea]. Disponible en: https://dermnetnz.org/

[4] «Mayoclinic.org», mayoclinic.org. Accedido: 1 de junio de 2024. [En línea]. Disponible en: https://www.mayoclinic.org/es/diseases-conditions/psoriasis/symptoms-causes/syc-20355840

[5] «Chronic plaque psoriasis images — DermNet». Accedido: 4 de junio de 2024. [En línea]. Disponible en: https://dermnetnz.org/images/chronic-plaque-psoriasis-images

[6] «Guttate psoriasis images | DermNet». Accedido: 4 de junio de 2024. [En línea]. Disponible en: https://dermnetnz.org/topics/guttate-psoriasis-images

[7] «ImageJ User Guide - IJ 1.46r». Accedido: 4 de junio de 2024. [En línea]. Disponible en: https://imagej.net/ij/docs/guide/index.html

[8] I. Arganda-Carreras *et al.*, «Trainable Weka Segmentation: a machine learning tool for microscopy pixel classification», *Bioinformatics*, vol. 33, n.º 15, pp. 2424-2426, ago. 2017, doi: 10.1093/bioinformatics/btx180.

[9] A. Addagatla, «Investigating Underfitting and Overfitting», Geek Culture. Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://medium.com/geekculture/investigating-underfitting-and-overfitting-70382835e45c

[10] J. Rathod, «Underfitting, Overfitting, and Regularization», Jash Rathod. Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://jashrathod.github.io/2021-09-30-underfitting-overfitting-and-regularization/

[11] «imgaug — imgaug 0.4.0 documentation». Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://imgaug.readthedocs.io/en/latest/

[12] «History and License», Python documentation. Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://docs.python.org/3/license.html

[13] L. J. C. G, «Aprende Python desde el nivel más básico». Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://entrenamiento-python-basico.readthedocs.io/es/3.7/

[14] «The HDF5® Library & File Format», The HDF Group. Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://www.hdfgroup.org/solutions/hdf5/

[15] «HDF5: The HDF5 Data Model and File Structure». Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://docs.hdfgroup.org/hdf5/v1_14/_h5_d_m__u_g.html#subsec_data_model_intro

[16] «Hierarchical Data Formats - What is HDF5? | NSF NEON | Open Data to Understand our Ecosystems». Accedido: 4 de junio de 2024. [En línea]. Disponible en: https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5

[17] P. Juyal y A. Kundaliya, «Multilabel Image Classification using the CNN and DC-CNN Model on Pascal VOC 2012 Dataset», en *2023 International Conference on Sustainable Computing and Smart Systems (ICSCSS)*, jun. 2023, pp. 452-459. doi: 10.1109/ICSCSS57650.2023.10169541.

[18] «¿Qué es una red neuronal? | IBM». Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://www.ibm.com/es-es/topics/neural-networks

[19] «Segmentación semántica de imágenes con Deep Learning». Accedido: 2 de junio de 2024. [En línea].

Disponible en: https://blog.damavis.com/segmentacion-semantica-de-imagenes-con-deep-learning/

[20] U. Udofia, «Basic Overview of Convolutional Neural Network (CNN)», DataSeries. Accedido: 5 de junio de 2024. [En línea]. Disponible en: https://medium.com/dataseries/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17

[21] P. Potrimba, «What is a Convolutional Neural Network?», Roboflow Blog. Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://blog.roboflow.com/what-is-a-convolutional-neural-network/

[22] L. Alzubaidi *et al.*, «Review of deep learning: concepts, CNN architectures, challenges, applications, future directions», *J. Big Data*, vol. 8, n.º 1, p. 53, mar. 2021, doi: 10.1186/s40537-021-00444-8.

[23] «XNet/Images/architecture.jpg at master · JosephPB/XNet · GitHub». Accedido: 5 de junio de 2024. [En línea]. Disponible en: https://github.com/JosephPB/XNet/blob/master/Images/architecture.jpg

[24] «Ronneberger et al. - 2015 - U-Net Convolutional Networks for Biomedical Image.pdf». Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://arxiv.org/pdf/1505.04597

[25] O. Ronneberger, P. Fischer, y T. Brox, «U-Net: Convolutional Networks for Biomedical Image Segmentation», en *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, vol. 9351, N. Navab, J. Hornegger, W. M. Wells, y A. F. Frangi, Eds., en Lecture Notes in Computer Science, vol. 9351. , Cham: Springer International Publishing, 2015, pp. 234-241. doi: 10.1007/978-3-319-24574-4_28.

[26] «Badrinarayanan et al. - 2016 - SegNet A Deep Convolutional Encoder-Decoder Archi.pdf». Accedido: 2 de junio de 2024. [En línea]. Disponible en: https://arxiv.org/pdf/1511.00561

[27] «Papers with Code - SegNet Explained». Accedido: 5 de junio de 2024. [En línea]. Disponible en: https://paperswithcode.com/method/segnet

[28] J. I. B. Arce, «La matriz de confusión y sus métricas – Inteligencia Artificial –», Juan Barrios. Accedido: 5 de junio de 2024. [En línea]. Disponible en: https://www.juanbarrios.com/la-matriz-de-confusion-y-sus-metricas/

[29] «Figure 7. Confusion Matrix 3x3», ResearchGate. Accedido: 5 de junio de 2024. [En línea]. Disponible en: https://www.researchgate.net/figure/Confusion-Matrix-3x3_fig3_365099641

[30] J. A. Martínez Pérez y P. S. Pérez Martin, «La curva ROC», *Med. Fam. SEMERGEN*, vol. 49, n.º 1, ene. 2022, doi: 10.1016/j.semerg.2022.101821.

[31] Y.-L. Lin, A. Huang, C.-Y. Yang, y W.-Y. Chang, «Measurement of Body Surface Area for Psoriasis Using U-net Models», *Comput. Math. Methods Med.*, vol. 2022, n.º 1, p. 7960151, 2022, doi: 10.1155/2022/7960151.

[32] M. V. Polyakova, «IMAGE SEGMENTATION WITH A CONVOLUTIONAL NEURAL NETWORK WITHOUT POOLING LAYERS IN DERMATOLOGICAL DISEASE DIAGNOSTICS SYSTEMS», *Radio Electron. Comput. Sci. Control*, n.º 1, Art. n.º 1, feb. 2023, doi: 10.15588/1607-3274-2023-1-5.

ANEXO

Durante el desarrollo del trabajo, se han presentado desafíos significativos que han impactado en la eficiencia y la efectividad de las investigaciones.

Aumento de la base de datos

En este apartado se va a definir y aclarar el código encargado en la parte de aumento de la base de datos. Este se conforma de varios archivos en lenguaje de programación Python.

Se han creado todos los valores iniciales para que sean compatibles al dataset utilizado en el proyecto, de lo contrario sería inviable realizar la tarea.

Se ha adaptado la función *balanced_test_val_split()* del archivo útil.py para que concuerde con la jerarquía de directorios que se proporciona.

Se han modificado partes de código y añadido otras nuevas. En concreto el archivo Augmentation.py se le ha añadido el aumento de las imágenes destinadas a la validación, ya que al tener una base de datos pequeña al aplicar la división para entrenar el modelo, la parte de validación era muy poco notoria, causando problemas considerables de *overfitting*.

Por último, se completa el codigo con comentarios a las diversas partes para aclarar que está sucediendo en cada momento, facilitando sus posteriores modificaciones o implementaciones.

AUGMENTATIONS.PY

Es el archivo principal que desarrolla todas las operaciones para conseguir el aumento del dataset de imágenes de psoriasis.

Inicialmente se realizan los *imports* de las librerías útiles y archivos adicionales creados con funciones que posteriormente serán usadas, entre ellas se pueden observar bibliotecas como *imgaug* explicada en una sección anterior del trabajo. *Keras, numpy o sklearn* son librerías añadidas entre otras.

Una vez realizado todos los *imports* necesarios para el correcto funcionamiento, se desarrolla la primera parte del código, en la cual se definen los parámetros a utilizar en el proceso de aumento.

Entre los parámetros se encuentra *main_path* que define el directorio raíz. *data_to_add,* define las carpetas con los tipos de psoriasis. *Image_size train_size* y *n_clases* indican la resolución de imagen, el número de imágenes que se cogen del dataset original para la parte de entrenamiento y el número de clasificadores que contienen las imágenes respectivamente.

Una aclaración que hay que realizar es que, el tamaño de la imagen, implementada por el parámetro definido *image_size* tiene un valor de 160. Además debe cumplir una condición extra, y es que estas resoluciones utilizadas deben ser múltiplos de 32, observandose resultados disconformes en el entrenamiento y errores varios a la hora de ejecutarlo en caso contrario

Examples_per category y *examples_per_category_val*, serán respectivamente el aumento del número de imágenes de entrenamiento y de validación que se va a hacer sobre las imágenes implementadas.

```
from __future__ import division
from sklearn.model_selection import train_test_split
import os, sys
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
import imgaug as ia
from imgaug import augmenters as iaa
from imgaug import parameters as iap
import create_h5
import cv2
import glob
from random import shuffle
import h5py
import argparse
from keras.utils import to_categorical
import random
from keras.preprocessing.image import ImageDataGenerator
from utils import random_crop
from utils import shuffle_together
from utils import balanced_test_val_split
import sys
import time


import matplotlib.pyplot as plt


# ****************** PARAMETROS ************************#


main_path = "Data"
data_to_add = ['CP' , 'GUTATE']            #directorio de la imagenes Chronic Plaques y Gutate respectivamente
hdf5_path = "final"
EXAMPLES_PER_CATEGORY = 1000               #numero de imagenes parte de entrenamiento
EXAMPLES_PER_CATEGORY_VAL=300              #numero de imagenes parte validacion
image_size = 128                           #resolucion de la imagen de entrada
train_size = 0.4                           #division en porcentaje del dataset de entrada#al principio 0.7 es mejor 0.6
n_classes = 3                              #numero de clasificadores que existen para las imagenes



# configuracion nombre de salida del dataset creado en HDF5
hdf5_name = '_'.join(data_to_add)

if(EXAMPLES_PER_CATEGORY == 0):
    hdf5_name = hdf5_name + '_s' + str(image_size) + '.hdf5'

else:
    hdf5_name =  hdf5_name +'_s'+str(image_size)+'_a'+ str(EXAMPLES_PER_CATEGORY)+ '.hdf5'
```

El código a continuacion es el encargado del balance del dataset en la división que se utiliza en la red neuronal para su entrenamiento, para ello se utiliza la función *balanced_test_val_split()* creada en el archivo *util.py*

Se divide el conjunto de datos en tres partes: entrenamiento, validación y prueba utilizando la función.

```
# ****************** BALANCE SET EN PARTES TRAIN/VAL/TEST *********************#

# Con la ayuda de la funcion balanced_test_val_split() desarrollada en utils.py
#se consigue un reparto equitativo de la simagenes en sets de entrenamiento validacion y test

images_train, labels_train, body_train, filenames_train, images_test, labels_test, body_test, \
filenames_test, images_val, labels_val, body_val, filenames_val = \
balanced_test_val_split(main_path, data_to_add, image_size, train_size, n_classes)
```

Definida las distintas partes, se procede a realizar el aumento de las imágenes para la parte de entrenamiento.

Se utiliza la función *unique()* para conseguir que el número de aumentos por cada imagen del set sea un número semejante, aportando equilibrio extra al dataset.

Seguidamente se realiza la definición de las transformaciones que serán aplicadas en las imágenes originales de la base de datos. Las transformaciones utilizadas han sido definidas anteriormente en el proyecto, además en el siguiente código se han definido los valores necesarios para poder realizar cada transformación, sus máximos y mínimos, porcentajes y cualquier otro valor indispensable. Los valores escogidos son:

translate_max = 0.01 //valor máximo de desplazamiento de la imagen

rotate_max = 15 // valor de rotación de la imagen en grados

shear_max = 2 //valor para el cizallamiento

```
# ****************** AUMENTO PARTE ENTRENAMIENTO *********************#


# Se decide el numero de aumentos por cada imagen para conseguir un dataset de entrenamiento equitativo

unique, counts = np.unique(body_train, return_counts=True)
unique_per_category = dict(zip(unique, counts))
augmentations_per_category = dict(unique_per_category)
for key in unique_per_category:
    augmentations_per_category[key] = int(EXAMPLES_PER_CATEGORY/unique_per_category[key])


# Parametros de configuracion para las transofrmaciones aplicadas
translate_max = 0.01
rotate_max = 15
shear_max = 2

# Definicion de las transformaciones aplicadas posteriormente

affine_trasform = iaa.Affine( translate_percent={"x": (-translate_max, translate_max), # Desplazamientos positivos y negativos en el eje X, marcados por el valor translate_max
                                                 "y": (-translate_max, translate_max)}, # Desplazamientos positivos y negativos en el eje Y, marcados por el valor translate_max
                              rotate=(-rotate_max, rotate_max), # Rotaciones desde -rotate_max hasta rotate_max en grados
                              shear=(-shear_max, shear_max), # Acizallamiento marcados por el valor negativo y positivo shear_max en grados
                              order=[1], # usa la tecnica del pixel vecino mas proximo, por defecto
                              cval=125, # si el modo es constante, usar en este caso un valor entre 0 y 255
                              mode="reflect",
                              name="Affine",
                              )


spatial_aug = iaa.Sequential([iaa.Fliplr(0.5), iaa.Flipud(0.5), affine_trasform])

other_aug = iaa.SomeOf((1, None),
       [
           iaa.OneOf([
               iaa.GaussianBlur((0, 0.4)), # desenfoca las imagenes con un valor entre 0 y 1
               iaa.ElasticTransformation(alpha=(0.5, 1.5), sigma=0.25), # se aplica poca transformacion elastica

           ]),

       ])
```

```
#Aplicacion de las transformaciones a la parte de imagenes y mascaras de etiquetas destinadas al entrenamiento

augmentator = [spatial_aug,other_aug]
total_images=sum(augmentations_per_category[k]*unique_per_category[k] + unique_per_category[k] for k in augmentations_per_category)
images_aug = np.zeros((total_images,images_train.shape[1],images_train.shape[2],images_train.shape[3]))
labels_aug = np.zeros((total_images,labels_train.shape[1],labels_train.shape[2],labels_train.shape[3]))
bodypart = np.empty((total_images),dtype = 'S10')
filenames_aug = np.empty((total_images),dtype = 'S60')

images_aug[:images_train.shape[0],...] = images_train
labels_aug[:images_train.shape[0],...] = labels_train/255
bodypart[:images_train.shape[0],...] = body_train
filenames_aug[:images_train.shape[0],...] = filenames_train

# Bucle para las distintas imagenes
counter = images_train.shape[0]
counter_block = 0

error_images = [] #lista para imagenes con error de dimension
```

Se procede a realizar el bucle para aplicar las transformaciones, si por cualquier motivo se produjera un error, se ha configurado un método de detección de errores, para que se notifique.

Al ser unas transformaciones definidas pero aleatorias, para que las mismas transformaciones sean aplicadas a cada imagen con su respectiva mascara se añaden varias líneas de código al principio del bucle, evitando un entrenamiento de la red convolucional totalmente inválido.

```
for i, (k, v) in enumerate(augmentations_per_category.items()):

    indices = np.array(np.where(body_train == k )[0])
    # Numero de aumentos por imagen
    N = int(v)

    for j in indices:
        for l in range(N):
            clear_output(wait=True)
            # Se mantiene los prarametros aleatorios de las transofrmaciones para aplicarlos tambien a sus respectivas mascaras de segmentacion.
            spatial_det = augmentator[0].to_deterministic()
            other_det = augmentator[1]


            # se realiza una captura de imagenes que puedan dar fallos de dimension durante la aplicacion de las tranformaciones
            try:
                images_aug[counter,...] = spatial_det.augment_image(images_train[j])

                labels_aug[counter,...] = spatial_det.augment_image(labels_train[j])

                img_crop, label_crop = random_crop(images_aug[counter,...],labels_aug[counter,...],0.1,0.4)
                images_aug[counter,...] = other_det.augment_image(img_crop )
                labels_aug[counter,...] = to_categorical(np.argmax(label_crop,axis=-1))

                bodypart[counter] = k

                # Guardado de imagenes aumentadas aÃ±adiendo el prefijo aug_
                filenames_aug[counter] = b'aug_' + filenames_train[j]
                sys.stdout.write('(Categoria %s) procesando imagen %i/%i, aumentando imagen %i/%i'%(k,counter_block,
                                                                                    body_train.shape[0],
                                                                                    l+1, N))

                sys.stdout.flush()
                time.sleep(0.5)
                counter +=1

            except Exception as e:
                print(f'Error en la dimension en la imagen  {filenames_train[j]}')
                error_images.append(filenames_train[j])

        counter_block +=1

#imprime la imagen erronea

print('Imagen con Problema')
for error_image in error_images:
    print(error_image)
```

Como ya se ha comentado varias veces a lo largo del proyecto, la base de datos obtenida es pequeña y a la hora de hacer la división de las tres partes del set conocida, el número destinado a la validación crea en un principio problemas de *overfitting* en el momento de ejecución de la red neuronal.

Para solucionarlo se añade un aumento de las imágenes de la parte de validación, semejante a la del entrenamiento.

```
# ****************** AUGMENTATIONS PARTE VALIDACION *********************#


# Al ser una base de datos muy pequeÃ±a, se realiza tambien un aumento en la parte de las imagenes destinadas a la validacion del programa


# Se decide el numero de aumentos por cada imagen para conseguir un dataset de validacion equitativo
unique_val, counts_val = np.unique(body_val, return_counts=True)
unique_per_category_val = dict(zip(unique_val, counts_val))
augmentations_per_category_val = dict(unique_per_category_val)
for key in unique_per_category_val:
    augmentations_per_category_val[key] = int(EXAMPLES_PER_CATEGORY_VAL/unique_per_category_val[key])

# Parametros de configuracion para las transofrmaciones aplicadas
translate_max = 0.01
rotate_max = 15
shear_max = 2

affine_trasform = iaa.Affine( translate_percent={"x": (-translate_max, translate_max), # Desplazamientos positivos y negativos en el eje X, marcados por el valor translate_max
                                                 "y": (-translate_max, translate_max)}, # Desplazamientos positivos y negativos en el eje Y, marcados por el valor translate_max
                              rotate=(-rotate_max, rotate_max), # Rotaciones desde -rotate_max hasta rotate_max en grados
                              shear=(-shear_max, shear_max), # Acizallamiento marcados por el valor negativo y positivo shear_max en grados
                              order=[1], # usa la tecnica del pixel vecino mas proximo, por defecto
                              cval=125, # si el modo es constante, usar en este caso un valor entre 0 y 255
                              mode="reflect",
                              name="Affine",
                              )


spatial_aug = iaa.Sequential([iaa.Fliplr(0.5), iaa.Flipud(0.5), affine_trasform])


other_aug = iaa.SomeOf((1, None),
        [
            iaa.OneOf([
                iaa.GaussianBlur((0, 0.4)), # desenfoca las imagenes con un valor entre 0 y 1
                iaa.ElasticTransformation(alpha=(0.5, 1.5), sigma=0.25), # se aplica poca transformacion elastica

            ]),

        ])
```

```
augmentator_val = [spatial_aug,other_aug]

total_images_val=sum(augmentations_per_category_val[k]*unique_per_category_val[k] + unique_per_category_val[k] for k in augmentations_per_category_val)
images_val_aug = np.zeros((total_images_val,images_val.shape[1],images_val.shape[2],images_val.shape[3]))
labels_val_aug = np.zeros((total_images_val,labels_val.shape[1],labels_val.shape[2],labels_val.shape[3]))
bodypart_val = np.empty((total_images_val),dtype = 'S10')
filenames_val_aug = np.empty((total_images_val),dtype = 'S60')

images_val_aug[:images_val.shape[0],...] = images_val
labels_val_aug[:images_val.shape[0],...] = labels_val/255
bodypart_val[:images_val.shape[0],...] = body_val
filenames_val_aug[:images_val.shape[0],...] = filenames_val

# Bucle para las distintas imagenes
counter_val = images_val.shape[0]
counter_block_val = 0
for i, (k, v) in enumerate(augmentations_per_category_val.items()):

    indices_val = np.array(np.where(body_val == k )[0])
    # Numero de aumentos por imagen
    N = int(v)

    for j in indices_val:
        for l in range(N):
            clear_output(wait=True)
            # Se mantiene los prarametros aleatorios de las transofrmaciones para aplicarlos tambien a sus respectivas mascaras de segmentacion.
            spatial_det = augmentator_val[0].to_deterministic()
            other_det = augmentator_val[1]

            images_val_aug[counter_val,...] = spatial_det.augment_image(images_val[j])

            labels_val_aug[counter_val,...] = spatial_det.augment_image(labels_val[j])
            img_crop_val, label_crop_val = random_crop(images_aug[counter_val,...],labels_aug[counter_val,...],0.1,0.4)
```

```
            images_val_aug[counter_val,...] = other_det.augment_image(img_crop_val )
            labels_val_aug[counter_val,...] = to_categorical(np.argmax(label_crop_val,axis=-1)) #aÃ±adio numclasses del final
            bodypart_val[counter_val] = k

            # Guardado de imagenes aumentadas aÃ±adiendo el prefijo aug_
            filenames_val_aug[counter_val] = b'aug_' + filenames_val[j]
            sys.stdout.write('(Categoria %s) procesando imagen %i/%i, aumentando imagen %i/%i'%(k,counter_block_val,
                                                                                                body_val.shape[0],
                                                                                                l+1, N))

            sys.stdout.flush()
            time.sleep(0.5)
            counter_val +=1
        counter_block_val +=1
```

En la parte final del código, se crea un dataset en formato HDF5. Haciendo uso de la función *write_h5()*, creada en el archivo *create_h5.py,* se consigue el objetivo de crear un dataset aumentado definido, listo para ser usado en el entrenamiento de los modelos XNET y UNET.

El archivo HDF5 contiene las imágenes aumentadas y sus máscaras complementarias divididas en entrenamiento, validación y test.

```
# ****************** CREACION BASE DE DATOS FINAL TRAS AUMENTO EN FORMATO HDF5  *********************#


images_aug, labels_aug, bodypart, filenames_aug = shuffle_together(images_aug, labels_aug, bodypart, filenames_aug)

images_test, labels_test, body_test, filenames_test = shuffle_together(images_test, labels_test, body_test, filenames_test)

images_val_aug, labels_val_aug, body_val_aug, filenames_val_aug = shuffle_together(images_val_aug, labels_val_aug, bodypart_val, filenames_val_aug)

print('Finalizando el aumento del Dataset ')

create_h5.write_h5(hdf5_path + hdf5_name, images_aug, labels_aug, bodypart,filenames_aug, images_test, labels_test/255,body_test,filenames_test,\
        images_val_aug, labels_val_aug,bodypart_val ,filenames_val_aug)


print('Guardando el archivo hdf5 en %s ...'%hdf5_name)
```

UTIL.PY

Define varias funciones utilizadas en *augmentation.py* y en el archivo de entrenamiento de XNET, *Train.py*. Una de las mayores modificaciones ha sido adaptar la función *balanced_test_val_split()* para que funcione con la base de datos de psoriasis, como el cambio de formato de las imágenes y sus máscaras, que de entrada son en formato *.jpg*, entre otros cambios. Además adaptar librerías que estaban obsoletas a las nuevas como son el cambio *scipy* por *rasterio* para el tratamiento de imágenes.

El script consta de tres funciones diferenciadas definidas a continuación:

- ***balanced_test_val_split()*:** Esta función tiene como objetivo dividir un conjunto de datos en conjuntos de entrenamiento, prueba y validación, garantizando que haya un equilibrio en la cantidad de datos de cada clase en el conjunto.

```python
import numpy as np
import random
import rasterio
import glob
import cv2
from random import shuffle
import os
import scipy.misc
import rgb2label as gen_label

from skimage.transform import resize
from skimage import data
import matplotlib.pyplot as plt

# Archivo que cuenta con una serie de funciones utiles para el desempeÃ±o del aumento de la base de datos y el entrenamiento del modelo


### Funcion BALANCED_TEST_VAL_SPLIT() ###
"""
   Divide un conjunto de datos en conjuntos de entrenamiento, prueba y validacion,
   garantizando que haya un equilibrio en la cantidad de datos de cada clase a la hora del entrenamiento del modelo.
"""
def balanced_test_val_split(main_path, data_to_add, image_size, train_size, n_classes):
    images_found = []
    labels_found = []
    for category in data_to_add:

        print('Comprobando si las mascaras de segmentacion y los datos coinciden en la carpeta %s ...'%category)
        data_path =os.path.join( main_path , category, 'Images')
        data_path += os.sep + '*.jpg'
        labels_path = os.path.join(main_path, category, 'Labels')
        labels_path += os.sep + '*.jpg'

        images = glob.glob(data_path)
        labels =  glob.glob(labels_path)
        assert len(labels) != 0
        print('Comprobando si el numero de mascaras de segmentacion coincide con el numero de archivos de imagenes de datos...')

        # Comprueba que el numero de etiquetas corresponde con el numero de imagenes
        assert len(labels) == len(images)

        # Comprueba que la mascara y las imagenes tienen el mismo nombre
        label_filename = []
        img_filename = []

        for (i, img) in enumerate(images):
            label_filename.append(labels[i].split(os.sep)[-1].split('.')[0].replace('onehot', ''))
            img_filename.append(img.split(os.sep)[-1].split('.')[0]  )


        label_filename = sorted(label_filename)
        img_filename = sorted(img_filename)

        for i in range(len(label_filename)):

            assert label_filename[i] == img_filename[i]
            images_found.append(  os.path.join(main_path , category, 'Images') + os.sep + img_filename[i] + '.jpg')
            labels_found.append(  os.path.join(main_path , category, 'Labels') + os.sep + label_filename[i] + '.jpg')


        print('Los nombres de las mascaras y los datos en la carpeta %s coinciden perfectamente, se encontraron %d imagenes. '%(category, len(img_filename)))
```

```python
    #Baraja imagenes y mascaras de segmentacion
    c = list(zip(images_found,labels_found))
    shuffle(c)
    images, labels = zip(*c)

    # Lee y guarda todas las imagenes + mascara de segmentacion
    images_read = np.zeros((len(images),image_size,image_size,1),dtype=np.float32)
    labels_read = np.zeros((len(labels), image_size, image_size,3),dtype=np.uint8)
    bodyparts = np.empty((len(images)),'S10')
    split_names = np.empty((len(images)),'S50')
    for i in range(len(images)):
        filename = images[i]
        img = rasterio.open(filename)
        img = img.read(1)
        images_read[i,...,0] = cv2.resize(img, (image_size, image_size), interpolation=cv2.INTER_AREA)
        label_filename = labels[i]
        labels_images = cv2.imread(label_filename)

        labels_read[i,...] = resize(labels_images, (image_size,image_size,3))
        labels_read[i,...] = labels_images
        plt.imshow(labels_read[0])
        labels_read[i,...] = np.uint8(labels_read[i,...])
        labels_read[i,...] = 255*gen_label.get_categorical_label(labels_read[i,...], n_classes)
        plt.imshow(labels_read[7])


        bodypart = filename.split(os.sep)[-1].split('_')[0].lower()
        split_names[i] = filename.split(os.sep)[-1].split('.')[0].lower()
        bodypart = ''.join(i for i in bodypart if not i.isdigit())
        bodypart = bodypart.split('.')[0]
        bodypart = bodypart.encode("ascii", "ignore")
        bodyparts[i] = bodypart


    unique, counts = np.unique(bodyparts, return_counts=True)
    unique_per_category = dict(zip(unique, counts))

    indices = np.arange(images_read.shape[0])
```

```
    # Construccion equilibrada de un set de pruebas y validacion
    one_per_class = []
    for i in unique_per_category:
        split_category = np.where(bodyparts==i)[0].tolist()
        #coge una de cada parte para conformar el set
        chosen_one_per_class = random.choice(split_category)
        indices_to_remove = np.argwhere( indices ==chosen_one_per_class)[0].tolist()
        indices = np.delete(indices, indices_to_remove)
        one_per_class.append(chosen_one_per_class)

    bodyparts_cut = bodyparts[indices]
    unique, counts = np.unique(bodyparts_cut, return_counts=True)
    unique_per_category = dict(zip(unique, counts))

    extra_need = int((1-train_size)*len(images)) - len(one_per_class)

    counter = 0
    test_extra = []
    while ( counter < extra_need ):
        keys = list(unique_per_category.keys())
        np.random.shuffle(keys)
        for bodypart in keys:
            if ( counter >= extra_need):
                break
            if( unique_per_category[bodypart] == 1 or unique_per_category[bodypart] == 0):
                continue

            #Consigue muestra aleatorias de cada parte
            bodypart_indices = np.where(bodyparts[indices] == bodypart)[0].tolist()
            bodypart_choice = random.choice(indices[bodypart_indices])
            test_extra.append(bodypart_choice)
            #Elimina las repeticiones
            unique_per_category[bodypart] -= 1
            remove_bodypart_index = np.argwhere( indices == bodypart_choice)[0].tolist()
            indices = np.delete(indices, remove_bodypart_index )
            counter += 1

    test_indices = (np.concatenate((one_per_class,test_extra))).astype(int)

    images_train = images_read[indices,...]
    body_train = bodyparts[indices]
    split_names_train = split_names[indices]
    labels_train = labels_read[indices,...]

    random.shuffle(test_indices)
    images_test = images_read[test_indices[:int(len(test_indices)/2)],...]
    body_test = bodyparts[test_indices[:int(len(test_indices)/2)]]
    split_names_test = split_names[test_indices[:int(len(test_indices)/2)]]
    labels_test = labels_read[test_indices[:int(len(test_indices)/2)],...]

    images_val = images_read[test_indices[int(len(test_indices)/2):],...]
    body_val = bodyparts[test_indices[int(len(test_indices)/2):]]
    split_names_val = split_names[test_indices[int(len(test_indices)/2):]]
    labels_val = labels_read[test_indices[int(len(test_indices)/2):],...]

    # Comprobacion de que no hay perdida de imagenes en el proceso
    assert (images_train.shape[0] + images_test.shape[0] + images_val.shape[0]) == len(images)

    return images_train, labels_train, body_train, split_names_train, images_test, labels_test, body_test,\
    split_names_test, images_val, labels_val, body_val, split_names_val
```

- ***shuffle_together_simple()*:** esta función tiene como objetivo mezclar las imágenes y máscaras de segmentación juntas.
- ***shuffle_together()*:** su objetivo es mezclar las imágenes, etiquetas y nombres de archivo juntos.
- ***random_crop()*:** se encarga de recortar aleatoriamente una imagen y su etiqueta asociada, manteniendo una proporción entre los valores que determinan el tamaño del recorte.

```
### Funcion SHUFFLE_TOGETHER_SIMPLE() ###
"""
    Baraja (mezcla) las imÃ¡genes, etiquetas
    y partes del cuerpo juntas.
"""
def shuffle_together_simple(images, labels, bodyparts):

    c = list(zip(images,labels, bodyparts))
    shuffle(c)
    images, labels, bodyparts = zip(*c)
    images = np.asarray(images)
    labels = np.asarray(labels)
    bodyparts = np.asarray(bodyparts)

    return images, labels, bodyparts

### Funcion SHUFFLE_TOGETHER() ###
"""
   Baraja (mezcla) las imÃ¡genes, etiquetas,
   partes del cuerpo y nombres de archivo juntos.
"""
def shuffle_together(images, labels, bodyparts, filenames):

    c = list(zip(images,labels, bodyparts,filenames))
    shuffle(c)
    images, labels, bodyparts, filenames = zip(*c)
    images = np.asarray(images)
    labels = np.asarray(labels)
    bodyparts = np.asarray(bodyparts)
    filenames = np.asarray(filenames)

    return images, labels, bodyparts, filenames

### Funcion RANDOM_CROP() ###
"""
   Recorta aleatoriamente una imagen y su etiqueta asociada manteniendo una proporciÃ³n
   entre los valores que determinan el tamaÃ±o del recorte.
"""
def random_crop(x, y, permin, permax):
```

```
    h, w, _ = x.shape
    per_h = random.uniform(permin, permax)
    per_w = random.uniform(permin, permax)
    crop_size = (int((1-per_h)*h),int((1-per_w)*w))

    rangew = (w - crop_size[0]) // 2 if w>crop_size[0] else 0
    rangeh = (h - crop_size[1]) // 2 if h>crop_size[1] else 0
    offsetw = 0 if rangew == 0 else np.random.randint(rangew)
    offseth = 0 if rangeh == 0 else np.random.randint(rangeh)
    cropped_x = x[offseth:offseth+crop_size[0], offsetw:offsetw+crop_size[1], :]
    cropped_y = y[offseth:offseth+crop_size[0], offsetw:offsetw+crop_size[1], :]
    resize_x = cv2.resize(cropped_x, (h, w), interpolation=cv2.INTER_CUBIC)
    resize_y = cv2.resize(cropped_y, (h, w), interpolation=cv2.INTER_NEAREST)
    if cropped_y.shape[-1] == 0:
        return x, y
    else:
        return np.reshape(resize_x,(h,w,1)), resize_y
```

CREATE_H5.PY

Este código define la función *Write_H5()*. Esta función es utilizada para crear un archivo HDF5 tras el aumento de la base de datos. Almacena los datos relacionados con las imágenes y máscaras de clasificación para su uso en el entrenamiento de las redes neuronales. Los datos se dividen en los tres conjuntos.

Tras obtener las imágenes y las máscaras de clasificación por la entrada de la función, se crea el archivo HDF5 que aloja toda la información.

Se definen también una serie de atributos como son los valores máximos y mínimos y la resolución de las imágenes.

```
from __future__ import division
import os, sys
import numpy as np
import cv2
import glob
from random import shuffle
from IPython.display import clear_output
import h5py
from sklearn.model_selection import train_test_split


### Funcion WRITE_H5() ###
"""
    Crea un archivo HDF5 para almacenar datos de imagenes junto con etiquetas y otra informacion
    en las partes diferenciadas de entrenamiento, prueba y validacion

    Parametros de entrada:
    hdf5_name: El nombre del archivo HDF5 que se va a crear
    images_train, labels_train, body_train, file_train: Datos de entrenamiento como son imagenes y mascaras de clasificacion
    images_test, labels_test, body_test, file_test: Datos de prueba como son imagenes y mascaras de clasificacion
    images_val, labels_val, body_val, file_val: Datos de validacion como son imagenes y mascaras de clasificacion

    Devuelve:
    El dataset HDF5 creado
"""

def write_h5(hdf5_name,images_train,labels_train,body_train, file_train, images_test,labels_test,body_test ,\
        file_test,images_val,labels_val,body_val,file_val):

    # Crea un archivo HDF5
    hdf5_file = h5py.File(hdf5_name, mode='w')

    # Atributos
    hdf5_file.attrs['image_size'] = images_train.shape[2]
    hdf5_file.attrs['max_value'] = 1.
    hdf5_file.attrs['min_value'] = 0.
    print(body_train.shape)
```

Posteriormente se crea la base de datos que se aloja en el fichero HDF5, diferenciando las tres partes existentes e incluyendo las imágenes y máscaras de clasificación, al igual que sus atributos en cada parte correspondiente.

Cada imagen se centra en cero y se normaliza.

Creado y configurado el archivo HDF5 se cierra y finaliza el código.

```
    # Base De Datos creada en HDF5
    hdf5_file.create_dataset("train_img", images_train.shape, np.float64)
    hdf5_file.create_dataset("train_label", labels_train.shape, np.int)
    hdf5_file.create_dataset("train_bodypart", body_train.shape, 'S10')
    hdf5_file.create_dataset("train_file", file_train.shape, 'S60')

    hdf5_file.create_dataset("test_img", images_test.shape, np.float64)
    hdf5_file.create_dataset("test_label", labels_test.shape, np.int)
    hdf5_file.create_dataset("test_bodypart", body_test.shape, 'S10')
    hdf5_file.create_dataset("test_file", file_test.shape, 'S60')

    hdf5_file.create_dataset("val_img", images_val.shape, np.float64)
    hdf5_file.create_dataset("val_label", labels_val.shape, np.int)
    hdf5_file.create_dataset("val_bodypart", body_val.shape, 'S10')
    hdf5_file.create_dataset("val_file", file_val.shape, 'S60')

    categories = ['train','test','val']
    images_split = [images_train, images_test, images_val]
    labels_split =  [labels_train, labels_test, labels_val]
    bodys_split = [body_train, body_test, body_val]
    names_split = [file_train, file_test, file_val]
    for j  in range(len(images_split)):
        for i in range(images_split[j].shape[0]):
            clear_output(wait=True)

            # Centra la imagen en cero
            img = images_split[j][i,...] - np.mean(images_split[j][i,...])

            # Se efectua una Normalizacion despues del aumento de la base de datos
            img = (img-np.min(img))/(np.max(img) - np.min(img))

            # Almacena los datos en el archivo HDF5
            hdf5_file[categories[j] + '_img'][i, ...] = img
            hdf5_file[categories[j] + "_label"][i, ...] = labels_split[j][i,...]
            hdf5_file[categories[j] + "_bodypart"][i] = bodys_split[j][i]
            hdf5_file[categories[j] + "_file"][i] = names_split[j][i]


    hdf5_file.close()
```

Entrenamiento de XNET y UNET

Al igual que se desarrolló en el aumento de la base de datos, se realiza una explicación de la sección de la programación enfocada al entrenamiento y obtención de los resultados.

Han sido modificados y creados varios archivos, ya que el código estaba adaptado a una base de datos de distintas regiones de huesos, por lo que las funciones que trataban el dataset de esta forma se han adaptado al nuevo dataset.

El procedimiento de entrenamiento en cada red neuronal es exactamente el mismo. La única diferencia a destacar es el cambio del archivo *XNET.py* por el de *UNET.py*, notificándolo como parámetro en el script *GENERATE_PARAMETER.py*.

XNET.PY

Parte esencial y en la que está enfocado todo el proyecto. Este código define la arquitectura de la red convolucional XNET, mediante el uso de *Keras*, biblioteca característica de Python, utilizada para tareas de *machine learning*.

Al ser XNET una arquitectura predefinida, no se ha modificado nada de ella, porque su modificación resulta una tarea muy compleja, objetivo no buscado en este proyecto.

Inicia el código con la importación de diversas funciones y clases incluidas en la biblioteca de *Keras*, esenciales para la construcción y entrenamiento del modelo definido.

Entre ellos se importa *Model* para realizar la arquitectura de la red y conectarlo entre ellos. Se añaden también las funciones de las capas y nodos que componen generalmente una CNN, como son la convolución, activación o la normalización de lotes entre otras.

También se incluyen retornos de llamadas a función. Por ejemplo, *Model checkpoint*, permite guardar el modelo en puntos de control durante el entrenamiento, *learningRateScheduler* se usa para adaptar la tasa de aprendizaje mientras se desarrolla el entrenamiento de la red.

```
from keras.models import Model
from keras.layers import Input, Concatenate, Conv2D, MaxPooling2D, UpSampling2D, Dropout, Cropping2D
from keras.layers import BatchNormalization, Reshape, Layer
from keras.layers import Activation, Flatten, Dense
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.metrics import categorical_accuracy
from keras import backend as K
from keras import losses
```

Realizadas las importaciones, el siguiente paso es la definición de la función *Model* que contiene la arquitectura. Como parámetros de entrada incluye *input_shape*, que define la forma de las imágenes. *Classes*, que da el número de clases que se van a predecir. El tamaño de *kernel* y profundidad de filtro.

Dentro de la función se pasa a crear la estructura codificador y decodificador de la red.

Comienza el codificador que reduce la resolución espacial de la imagen a medida que pasa a través de capas convolucionales, capas de *batch normalization* y funciones de activación ReLu. Esto se hace en los tres primeros nodos. Los nodos 4 y 5 son capas convolucionales extras.

```
def model(input_shape=(128,128,3), classes=3, kernel_size = 3, filter_depth = (64,128,256,512,1024)):

    img_input = Input(shape=input_shape)

    # Encoder
    conv1 = Conv2D(filter_depth[0], (kernel_size, kernel_size), padding="same")(img_input)
    batch1 = BatchNormalization()(conv1)
    act1 = Activation("relu")(batch1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(act1)
    #100x100

    conv2 = Conv2D(filter_depth[1], (kernel_size, kernel_size), padding="same")(pool1)
    batch2 = BatchNormalization()(conv2)
    act2 = Activation("relu")(batch2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(act2)
    #50x50

    conv3 = Conv2D(filter_depth[2], (kernel_size, kernel_size), padding="same")(pool2)
    batch3 = BatchNormalization()(conv3)
    act3 = Activation("relu")(batch3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(act3)
    #25x25

    #Flat
    conv4 = Conv2D(filter_depth[3], (kernel_size, kernel_size), padding="same")(pool3)
    batch4 = BatchNormalization()(conv4)
    act4 = Activation("relu")(batch4)
    #25x25

    conv5 = Conv2D(filter_depth[3], (kernel_size, kernel_size), padding="same")(act4)
    batch5 = BatchNormalization()(conv5)
    act5 = Activation("relu")(batch5)
    #25x25
```

Continúa iniciando el decodificador con los nodos 6 y 7 aplicando capas *upsampling*, que amplifican la salida de una capa, para aumentar la resolución de la imagen.

Los nodos 8 y 9 hacen referencia a la repetición del codificador, propiedad por la que se define la red XNET.

```
#Up
up6 = UpSampling2D(size=(2, 2))(act5)
conv6 = Conv2D(filter_depth[2], (kernel_size, kernel_size), padding="same")(up6)
batch6 = BatchNormalization()(conv6)
act6 = Activation("relu")(batch6)
concat6 = Concatenate()([act3,act6])
#50x50

up7 = UpSampling2D(size=(2, 2))(concat6)
conv7 = Conv2D(filter_depth[1], (kernel_size, kernel_size), padding="same")(up7)
batch7 = BatchNormalization()(conv7)
act7 = Activation("relu")(batch7)
concat7 = Concatenate()([act2,act7])
#100x100

#Down
conv8 = Conv2D(filter_depth[1], (kernel_size, kernel_size), padding="same")(concat7)
batch8 = BatchNormalization()(conv8)
act8 = Activation("relu")(batch8)
pool8 = MaxPooling2D(pool_size=(2, 2))(act8)
#50x50

conv9 = Conv2D(filter_depth[2], (kernel_size, kernel_size), padding="same")(pool8)
batch9 = BatchNormalization()(conv9)
act9 = Activation("relu")(batch9)
pool9 = MaxPooling2D(pool_size=(2, 2))(act9)

#25x25

#Flat
conv10 = Conv2D(filter_depth[3], (kernel_size, kernel_size), padding="same")(pool9)
```

En los nodos 10 y 11 se aplican dos capas de convolución adicionales seguidas de *batch normalization* y ReLu.

Tras los dos nodos anteriores y hasta el nodo 14 comienza la parte de decodificación del modelo, que

reconstruye la imagen a su resolución original, mediante capas de *upsampling* y unión de activaciones.

Como último nodo, el número 15 es el encargado de realizar la tarea de clasificación mediante una capa de convolución denominada *conv15,* que reduce el número de canales a clases de clasificación. Se sigue de una capa de activación *Softmax* que produce la salida final de la red dando una probabilidad para cada clase clasificadora.

Una vez definida toda la estructura del modelo, este se construye mediante la llamada de la función definida anteriormente, dando como parámetro la capa de entrada y la capa de salida.

Al final la función *Model* devuelve el modelo ya creado.

```
batch10 = BatchNormalization()(conv10)
act10 = Activation("relu")(batch10)
#25x25

conv11 = Conv2D(filter_depth[3], (kernel_size, kernel_size), padding="same")(act10)
batch11 = BatchNormalization()(conv11)
act11 = Activation("relu")(batch11)
#25x25

#Encoder
up12 = UpSampling2D(size=(2, 2))(act11)
conv12 = Conv2D(filter_depth[2], (kernel_size, kernel_size), padding="same")(up12)
batch12 = BatchNormalization()(conv12)
act12 = Activation("relu")(batch12)
concat12 = Concatenate()([act9,act12])
#50x50

up13 = UpSampling2D(size=(2, 2))(concat12)
conv13 = Conv2D(filter_depth[1], (kernel_size, kernel_size), padding="same")(up13)
batch13 = BatchNormalization()(conv13)
act13 =  Activation("relu")(batch13)
concat13 = Concatenate()([act8,act13])
#100x100

up14 = UpSampling2D(size=(2, 2))(concat13)
conv14 = Conv2D(filter_depth[0], (kernel_size, kernel_size), padding="same")(up14)
batch14 = BatchNormalization()(conv14)
act14 = Activation("relu")(batch14)
concat14 = Concatenate()([act1,act14])
#200x200

conv15 = Conv2D(classes, (1,1), padding="valid")(concat14)


reshape15 = Reshape((input_shape[0]*input_shape[1],classes))(conv15)
act15 = Activation("softmax")(reshape15)

model = Model(img_input, act15)

return model
```

GENERATE_PARAMETER.PY

En el siguiente script se realiza la tarea de definir y guardar una lista de parámetros en formato JSON, creando un archivo de texto. Este fichero de texto formado, será utilizado en el archivo principal *train.py*, para establecer la configuración de la clase interna *trainingclass*. Con este procedimiento se facilita la configuración y ejecución de entrenamientos de modelos de aprendizaje automático. Pueden ser modificados estos parámetros según sus necesidades y preferencias antes de ejecutar el entrenamiento del modelo.

Se inicia con un conjunto de importaciones de librerías, en este script en concreto enfocadas al manejo de archivos, directorios y estructuras de datos.

Seguidamente se inicia un conjunto de parámetros utilizados en la instancia de la clase entrenamiento. Todos ellos han sido modificados conforme las necesidades de la base de datos propia utilizada en el entrenamiento. Los parámetros son los siguientes:

- ***params_name***: El nombre del archivo que contendrá los parámetros en formato JSON.
- ***save_folder***: El nombre de la carpeta que contiene los archivos resultantes del entrenamiento.
- ***model***: El nombre del archivo que contiene la definición de la red neuronal convolucional.
- ***name***: Un nombre para el modelo resultante del entrenamiento.
- ***lrate***: La tasa de aprendizaje (*learning rate*) utilizada durante el entrenamiento.
- ***reg***: El valor de regularización, que parece estar en el rango de 0 a 0.1.
- ***batch_size***: El tamaño del lote (*batch size*) utilizado en el entrenamiento.
- ***kernel_size***: El tamaño del *kernel* en las capas de convolución.
- ***filter_list***: Una lista que define el número de filtros en cada capa convolucional.
- ***loss***: La métrica empleada para calcular la pérdida durante el entrenamiento.
- ***data***: El nombre del archivo que contiene los datos de entrenamiento.

Como aclaración el parámetro más modificado en el archivo a lo largo del proyecto, ha sido *no_epochs*, que indica las repeticiones que realiza el modelo para aprender. Por temas relacionados con el dataset de entrada y el rendimiento del *hardware*, es un valor que se ha reducido bastante. No obstante, es compensado con otros parámetros para conseguir unos buenos resultados.

```
import numpy
import json
import os
import sys

###### DEFINICION DE PARAMETROS ######


params_name = "parameters.txt" #Nombre del fichero de parametros
save_folder = "Model0" #carpeta que contiene los archivos tras el entrenamiento


model = "XNet.py" #archivo que contiene la CNN
name = "DLs196_64" #nombre que obtiene el modelo resultante del entrenamiento

lrate = 0.000100 #parametro learning rate
reg = 0.000      #valor para regular entre 0 y 0.1
batch_size = 2   #era 5 #internet dice que sea de multiplos de 2 mientras mas bajo mejor , se puede compensar con el learning rate
kernel_size = 3
filter_list =  [64,128,256, 512, 1024]


loss = "categorical_crossentropy" #Metrica empleada para las perdidas durante el entrenamiento


data = "finalCT_GUTATE_s160_a1000.hdf5"


#Por tema relacionados con el tamaÃ±o del dataset empleado en el proyecto y el rendimiento del hardware utilizado
#Se ha modificado el parametro no_epochs numero de ciclos de entrenamiento del modelo

no_epochs = 20 #numero de ciclos de entrenamiento del modelo
duplicate = True #estaba en true al principio
```

Para la creación del archivo JSON se define una estructura diccionario denominada *D*, en ella se incluyen todos los parámetros anteriormente declarados.

```
#Diccionario con el conjunto de parametros para la creacion del fichero JSON

d = {"name":name,
     "model_path": model,
     "data_path": data,
     "save_folder": save_folder,
     "kernel_size": kernel_size,
     "batch_size": batch_size,
     "filters": filter_list,
     "lrate": lrate,
     "reg":reg,
     "loss": loss,
     "no_epochs": no_epochs,
     "duplicate": duplicate}
```

El código verifica si ya existe un archivo con el nombre *params_name* (que contiene los parámetros). Si existe, se solicita una confirmación para borrar el archivo existente. Si el usuario confirma, el archivo existente se elimina. Si el usuario no confirma, el script se cierra, mediante la ejecución de un *sys.exit*.

Finalmente, el código utiliza la función *json.dump* para escribir el diccionario *D* en un archivo con el nombre *params_name*. Esto crea un archivo JSON que contiene todos los parámetros necesarios para el entrenamiento de un modelo de aprendizaje automático.

```
#Estructura para decidir si se quiere borrar el archivo de parametros existente
#Mediante la introduccion de y para caso afirmativo y n para el negativo

if (os.path.isfile(params_name)):
    confirm_metada = input("Aviso! desea borrar el archivo de parametros ya existente? (y/n) ")
    if(confirm_metada == "y"):
        os.remove(params_name)
    else:
        sys.exit()

# Escritura del fichero de parametros

with open(params_name, 'w') as fp:
    json.dump(d, fp)
```

RGB2LABEL.PY

En este fichero se declaran tres funciones para el tratamiento de las etiquetas, colores y máscaras. A continuación se define el desempeño de cada función:

- ***get_mask_from_color()***: tiene como objetivo crear una máscara que especifique la ubicación de un cierto tipo de color en la imagen, mediante la comparación de los valores de píxeles con el color especificado. Como parámetros de entrada se obtiene la imagen en cuestión y el color específico a identificar.
- ***get_012_label()***: en función de los colores determinados por cada píxel de una imagen, se asigna un etiquetado 012. Los parámetros de entrada serán la imagen, el número de colores y los colores a designar.
- ***get_categorical_label()***: mediante el uso de las dos funciones anteriores se calcula una etiqueta clasificadora. Como parámetros de entrada se obtiene la imagen a tratar y el número de clases existentes.

```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
import glob
from keras.utils import to_categorical


### Funcion GET_MASK_FROM_COLOR() ###
"""
    Devuelve una mascara de tipo boolean del misma resolucion que la imagen obtenida por entrada, con
    valores True en las posiciones de pixeles que coincide con el color de entrada

    Parametros de entrada:
    image: La imagen de entrada
    color: El color que se desea buscar en la imagen, representado como una lista de tres valores RGB

    Devuelve:
    Una mascara booleana con valores True donde se encuentra el color especificado en la imagen
"""

def get_mask_from_color( image, color ):
    rows, columns, channels = image.shape  # Obtiene las dimensiones de la imagen (filas, columnas, canales)
    total_pixels = rows * columns          # Calcula el numero total de pÃ xeles en la imagen
    image_flat = image.reshape(total_pixels, channels)
    color_array = np.array([color,] * total_pixels)    # Compara los valores de los pixeles con el color especificado
    channels_mask = np.isclose(image_flat, color_array, atol = 100)

    # Combina las mascaras de canales individuales
    mask = np.logical_and(channels_mask[:,0], channels_mask[:,1])
    mask = np.logical_and(mask, channels_mask[:,2])
    return mask.reshape(rows,columns)
```

```
### FUNCION GET_012_LABEL() ###
"""
    Devuelve una etiqueta 0, 1, 2 para 3 colores de una imagen

    Parametros de entrada:
    image: Imagen de entrada
    n_colors: Numero de colores a etiquetar pudiendo ser dos o tres
    colors: Una lista de los colores a etiquetar

    Devuelve:
    Una matriz de etiquetas 0, 1, 2 correspondientes a los colores en la imagen
"""

def get_012_label(image, n_colors = 3, colors = [[0,0,255], [0,255,0], [255,0,0]]):

     # Inicializa una matriz de etiquetas 0 con las mismas dimensiones que la imagen.
    label_012 = np.zeros((image.shape[0], image.shape[1]))
    # Si se requieren 2 colores, etiqueta con 1 los pixeles del tercer color en la lista.
    if(n_colors == 2):
        mask = get_mask_from_color(image, colors[2])
        label_012[mask] = 1
     # Si se requieren 3 colores, etiqueta con 1 los pixeles del segundo color y con 2 los pixeles del tercer color en la lista
    elif(n_colors == 3):
        mask = get_mask_from_color(image, colors[1])
        label_012[mask] = 1
        mask = get_mask_from_color(image, colors[2])
        label_012[mask] = 2
    # en caso de no implementarse ningun color
    else:
        print("Numero de colores no implementado")
        return False

    return label_012


### FUNCION GET_CATEGORICAL_LABEL() ###
"""
    Calcula la etiqueta 012 y la etiqueta de clasificacion de una imagen

    Parametros de entrada:
    image: Imagen de entrada
    n_classes: Numero de clases

    Devuelve:
    Una etiqueta de clasificacion calculada a partir de la etiqueta 012
"""

def get_categorical_label(image, n_classes = 3):

    label_012 = get_012_label(image, n_classes)
    return to_categorical(label_012, n_classes)
```

TRAIN.PY

Script principal en la parte de entrenamiento. Se desarrolla el entrenamiento y validación de la CNN, creando un modelo útil para la tarea de segmentación y clasificación de imágenes de psoriasis.

A la vez, en este script se obtienen unos datos informativos de los resultados obtenidos como son la exactitud conseguida por el modelo, los parámetros que miden el rendimiento, la curva Región de Convergencia conocida como ROC y la curva de aprendizaje. A continuación se explica el código escrito en el fichero.

Como en todos los ficheros de código de este proyecto, se añaden las bibliotecas y funciones externas necesarias para que el script sea funcional.

Tras las importaciones, se realiza la búsqueda del fichero de texto *parameter.txt*, que como fue comentado anteriormente guarda en su interior todos los parámetros de configuración del modelo a entrenar. Una vez encontrado el archivo, se almacenan los parámetros en la variable *params*.

La última tarea antes de iniciar el proceso de entrenamiento es la comprobación y creación de la carpeta que guarda todos los resultados obtenidos tras el entrenamiento. Esta tarea se realiza mediante una estructura condicional. Si la carpeta existe, para poder continuar hay que confirmar su eliminación, en caso contrario la ejecución se detiene.

```python
import numpy as np
import TrainingClass
import json
import os
import sys
from PostProcessing import PostProcessing
import glob
import shutil
import webbrowser
import time

param_files = glob.glob("parameters.txt")
print("Se entrenara utilizando este archivo de parametros:\n")
print(*param_files, sep = "\n")


for file in param_files:
    params = json.load(open(file,'r'))

    save_folder = params["save_folder"]
    # Comprobacion de la existencia de la carpeta de resultados, en caso de que exista
    # se pregunta para eliminarla, si no se confirma el programa se detiene
    if(os.path.isdir(save_folder)):
        rm_folder = input("Aviso, Carpeta en existencia! Desea Eliminar? (y/n) ")
        if(rm_folder == "y"):
            shutil.rmtree(save_folder)
        else:
            sys.exit()

    os.mkdir(save_folder)
```

Tras realizar las configuraciones previas pertinentes, comienza el entrenamiento. Se instancia un objeto de la clase *TrainingClass* con los parámetros cargados y se procede a entrenar el modelo utilizando el método *fit()*.

```python
    # Inicia el proceso de entrenamiento utilizando la clase TrainingClass con los parametros obtenidos previamente
    training = TrainingClass.TrainingClass(**params)
    try:
        training.fit()
    except:
        print("\n Finalizando... \n")

    print("Ejecutando analisis posterior al entrenamiento...\n")

    h5_files = np.sort(glob.glob(os.path.join(params["save_folder"], "*.h5")))
    try:
        # Inicia el analisis posterior al entrenamiento, para ello se usa la clase PostProcessing
        pp = PostProcessing( h5_files[-1], params["data_path"], device = "gpu")
    except:
        print("No se ha entrenado nada?")
        continue
```

Finalizado el entrenamiento, se realiza un análisis posterior mediante el uso de la clase *postproccesing* que incluye varias medidas de rendimiento. Se crea un archivo de texto llamado *results.txt* en la carpeta de resultados y se escriben métricas de rendimiento, incluyendo precisión, verdaderos positivos, falsos positivos y otras métricas. Además se dibujan dos gráficas, que en el siguiente párrafo se concretan.

```
# Calcula la precision global de acierto y el numero de parametros entrenables
pfile = open(os.path.join(params["save_folder"],  "results.txt"), "w")
pfile.write("Rendimiento general: \n")
accuracy_test, trainable_count = pp.evaluate_overall(device = "gpu")
pfile.write("Acierto: {} \nParametros entrenables: {} \n\n".format(round(accuracy_test,2)*100, trainable_count) )

pfile.write("Verdaderos Positivos and Falsos Positivos:\n")
tp, fp = pp.tpfp()
pfile.write(" TP: {} \n FP {} \n\n ".format(round(tp,2)*100, round(fp,2)*100))

# Realiza el analisis con un umbral del 90% y 99%
pfile.write("Umbral 90% \n")
thresh90 = pp.thresholding(0.9)
tp90, fp90 = pp.tpfp(thresh90)
pfile.write(" TP90: {} \n FP90 {}\n \n ".format(round(tp90,2)*100, round(fp90,2)*100))

pfile.write("Umbral 99% \n")
thresh99 = pp.thresholding(0.99)
tp99, fp99 = pp.tpfp(thresh99)
pfile.write(" TP99: {} \n FP99 {}\n \n ".format(round(tp99,2)*100, round(fp99,2)*100))

pfile.close()

# Generar la curva de aprendizaje
lc_fig, lc_ax = pp.learning_curve(os.path.join(params["save_folder"], params["name"] + ".csv"))
lc_fig.savefig(os.path.join(params["save_folder"], "learning_curve.png"))
```

Las cuatro últimas líneas de código, hacen referencia a la creación de dos imágenes en formato *.png* de dos gráficas, la primera correspondiente a la curva de aprendizaje del modelo, y la segunda a la curva ROC.

```
# Genera la curva ROC
rc_fig, rc_ax = pp.ROC_curve()
rc_fig.savefig( os.path.join(params["save_folder"] , "roc_curve.png") )
```

TRAININGCLASS.PY

Fichero que alberga la definición de la clase *trainingclass*, junto con todas las funciones que la componen, utilizada en el script principal para poder entrenar la red. Dado que cuenta con una gran cantidad de funciones se realiza una breve aclaración de la tarea que cumple cada una de ellas.

- ***_init_()***: Da valores a los parámetros de configuración, obtenidos del archivo *parameters.txt* creado en *Generate_parameters.py*.
- ***load_data()***: Carga de la base de datos desde un archivo de tipo HDF5.
- ***write_data()***: Escribe los metadatos en un fichero de texto, con todos los datos que configuran el entrenamiento.
- ***generator()***: Este Generador es usado para pasar los datos al algoritmo de entrenamiento. Se dividen los datos de entrenamiento en lotes. Esta función permite el entrenamiento cuando todos los datos no pueden ser procesados en la memoria RAM.

```python
import tensorflow
import h5py
import numpy as np
import os
import sys
import signal
import shutil
import importlib.util
import time

from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img
from keras.models import Model, Sequential, load_model
from keras.layers import *
from keras import backend as K
from keras import losses
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, CSVLogger, TensorBoard, EarlyStopping
from keras.metrics import categorical_accuracy
from utils import shuffle_together_simple, random_crop
from random import randint
import imgaug as ia
from keras.utils import to_categorical
from keras.optimizers import RMSprop
from keras.optimizers import Adam
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from imgaug import augmenters as iaa
from imgaug import parameters as iap



class TrainingClass:

    ### Funcion _INIT_() ###
    """
        Inicializa los parametros obtenidos del archivo de configuracion parameters.txt
    """
    def __init__(self, name, model_path, data_path, save_folder, no_epochs, kernel_size, batch_size, filters, lrate = 1e-4, reg = 0.0001,  loss = 'categorical_crossentropy', duplicate
        self.name = name
        self.model_path = model_path
        self.data_path = data_path
        self.save_folder = save_folder
        self.kernel_size = kernel_size
        self.batch_size = batch_size
        self.filters = filters
        self.lrate = lrate
        self.reg = reg
        self.no_epochs = no_epochs
        self.loss = loss
        self.load_data()
        if(duplicate == True):
            self.train, self.train_label, self.train_bodypart = self.duplicate()
            print("Finalizada la duplicacion de imagenes, el numero final del set de entrenamiento es %d images."%self.train.shape[0])
        self.write_metadata()
        self.compile()
```

```python
    ### Funcion LOAD_DATA() ###
    """
        Carga de datos desde un archivo de tipo H5
    """
    def load_data(self):
        hf = h5py.File(self.data_path, 'r')
        self.train = hf['train_img']
        self.no_images, self.height, self.width, self.channels= self.train.shape
        self.train_label = hf['train_label']
        self.train_bodypart = hf['train_bodypart'][:]
        self.no_images, _, _, self.no_classes = self.train_label.shape
        self.val = hf['val_img'][:]
        self.val_label = hf['val_label'][:]
        self.val_label = self.val_label.reshape((-1,self.height*self.width,self.no_classes))
        print("Datos cargados exitosamente.")

    ### Funcion WRITE_METADATA() ###
    """
        Escribe los metadato en un fichero txt, con todos los datos de entrenamiento
    """
    def write_metadata(self):
        metafile_path = self.save_folder + "/metadata.txt"

        if (os.path.isfile(metafile_path)):
            confirm_metada = input("Aviso! fichero de metadatos existe, continuar? (y/n) ")
            if(confirm_metada == "y"):
                shutil.rmtree(metafile_path)
            else:
                sys.exit()

        metadata = open(metafile_path, "w")
        metadata.write("name: %s \n"%self.name)
        metadata.write("Data: %s \n"%self.data_path)
        metadata.write("kernel_size: %d \n" %self.kernel_size)
        metadata.write("batch_size:%d \n" %self.batch_size)
        metadata.write("filters %s \n" %(self.filters,))
        metadata.write("lrate: %f \n" %self.lrate)
        metadata.write("reg: %f \n" %self.reg)
        metadata.write("Loss function: %s \n" %self.loss)
        metadata.write("no_epochs: %d \n" %self.no_epochs)
        metadata.close()

    ### Funcion GENERATOR_WITH_AUGMENTATIONS() ###
    """
        Este Generador es usado para pasar los datos al algoritmo de entrenamiento.
        Dando un tamanio de lote, se dividen los datos de entrenamiento en lotes.
        Esta Funcion permite el entrenamiento cuando todos los datos no pueden ser procesados en la memoria RAM
    """
    def generator(self):

        while True:
            indices = np.asarray(range(0, self.no_images))
            np.random.shuffle(indices)
            for idx in range(0, len(indices), self.batch_size):
                batch_indices = indices[idx:idx+self.batch_size]
                batch_indices.sort()
                batch_indices = batch_indices.tolist()
                by = self.train_label[batch_indices]
                by = by.reshape(-1, self.width*self.height, self.no_classes)
                bx = self.train[batch_indices]

                yield(bx,by)
```

- ***duplicate()***: Realiza una duplicación de las imágenes pertenecientes al set de entrenamiento para reducir el desequilibrio y evitar posibles problemas de *overfitting*.

```
### Funcion DUPLICATE() ###
"""
    Duplica las imagenes de entrenamiento para reducir el desequilibrio
"""
def duplicate(self):
    img_per_category, counts = np.unique(self.train_bodypart, return_counts=True)
    img_per_category = dict(zip(img_per_category, counts))
    EXAMPLES_PER_CATEGORY = max(img_per_category.values())
    duplications_per_category = dict(img_per_category)
    for key in img_per_category:
        duplications_per_category[key] = int(EXAMPLES_PER_CATEGORY/img_per_category[key])

    duplicated_size = sum(duplications_per_category[k]*img_per_category[k] + img_per_category[k] \
              for k in duplications_per_category)

    train_duplicated = np.zeros((duplicated_size,self.height,self.width,self.train.shape[3]))
    labels_duplicated = np.zeros((duplicated_size,self.height, self.width,self.no_classes))
    bodypart_duplicated = np.empty((duplicated_size),dtype = 'S10')

    train_duplicated[:self.no_images,...] = self.train
    labels_duplicated[:self.no_images,...] = self.train_label
    bodypart_duplicated[:self.no_images,...] = self.train_bodypart


    counter = self.no_images
    counter_block = 0
    for i, (k, v) in enumerate(duplications_per_category.items()):
        indices = np.array(np.where(self.train_bodypart == k )[0])
        counter_block += len(indices)
        # Numero de aumentos por imagen
        N = int(v)
        for j in indices:
            for l in range(N):
                train_duplicated[counter,...] =self.train[j]
                labels_duplicated[counter,...] = self.train_label[j]
                bodypart_duplicated[counter] = k
                counter +=1

    train_duplicated, labels_duplicated, bodypart_duplicated = shuffle_together_simple(train_duplicated, labels_duplicated, bodypart_duplicated)
    self.no_images = train_duplicated.shape[0]
    return train_duplicated, labels_duplicated, bodypart_duplicated
```

- ***augmentator()***: Define las transformaciones que se aplican a las imágenes y a sus respectivas máscaras de segmentación para aumentarlas, se hace uso de las mismas transformaciones utilizadas en el apartado de aumento del dataset.

```
### Funcion AUGMENTATOR() ###
"""
    Define las transformaciones que se aplican a las imagenes
    y a sus respectivas mascaras de segmentacion para aumentarlas
"""
def augmentator(self, index):

    translate_max = 0.01
    rotate_max = 15
    shear_max = 2

    affine_trasform = iaa.Affine( translate_percent={"x": (-translate_max, translate_max),
                                                     "y": (-translate_max, translate_max)},
                                  rotate=(-rotate_max, rotate_max),
                                  shear=(-shear_max, shear_max),
                                  order=[1],
                                  cval=125,
                                  mode="reflect",
                                  name="Affine",
                                 )


    spatial_aug = iaa.Sequential([iaa.Fliplr(0.5), iaa.Flipud(0.5), affine_trasform])

    other_aug = iaa.SomeOf((1, None),
            [
                iaa.OneOf([
                    iaa.GaussianBlur((0, 0.4)),
                    iaa.ElasticTransformation(alpha=(0.5, 1.5), sigma=0.25),

                ]),

            ])


    augmentator = [spatial_aug,other_aug]
    spatial_det = augmentator[0].to_deterministic()
    other_det = augmentator[1]

    image_aug = spatial_det.augment_image(self.train[index])
    label_aug = spatial_det.augment_image(self.train_label[index])
    img_crop, label_crop = random_crop(image_aug,label_aug,0.1,0.4)
    image_aug = other_det.augment_image(img_crop )
    label_aug = to_categorical(np.argmax(label_crop,axis=-1), num_classes = self.no_classes)
    return image_aug, label_aug
```

- ***generator_with_augmentations()***: Este generador es usado para pasar los datos al algoritmo de entrenamiento. A diferencia del otro generador antes definido, en este se dividen los datos de

entrenamiento en lotes, aumentando cada imagen una vez.

```
### Funcion GENERATOR_WITH_AUGMENTATIONS() ###
"""
    Este Generador es usado para pasar los datos al algoritmo de entrenamiento.
    Dando un tamanio de lote, se dividen los datos de entrenamiento en lotes, aumentando cada imagen una vez
"""
def generator_with_augmentations(self):

    batch_images = np.zeros((self.batch_size, self.width, self.height, 1))
    batch_labels = np.zeros((self.batch_size, self.width*self.height, self.no_classes))
    while True:
        indices = np.asarray(range(0, self.no_images))
        np.random.shuffle(indices)
        for idx in range(0, len(indices), self.batch_size):
            batch_indices = indices[idx:idx+self.batch_size]
            batch_indices.sort()
            batch_indices = batch_indices.tolist()
            for i, idx2 in enumerate(batch_indices):
                augmented_image, augmented_label = self.augmentator(idx)
                augmented_label = augmented_label.reshape(self.width*self.height, self.no_classes)
                batch_images[i] = augmented_image
                batch_labels[i] = augmented_label

            yield (batch_images,batch_labels)
```

- ***compile()***: carga un modelo desde un archivo externo, configura sus parámetros y lo compila para que esté listo para el entrenamiento. Permite flexibilidad a la hora de cargar diferentes modelos de manera dinámica.
- ***fit()***: entrena un modelo de red neuronal utilizando uno de los generadores de datos según si se aplica un aumento o no. Durante el entrenamiento, se verifica si la pérdida en el conjunto de validación mejora, y si no mejora el entrenamiento se detiene para evitar el sobreajuste.

```
### Funcion COMPILE() ###
"""
    carga un modelo desde un archivo externo, configura sus parametros y lo compila para que este listo para el entrenamiento.
    Permite flexibilidad a la hora de cargar diferentes modelos
"""
def compile(self):
    spec = importlib.util.spec_from_file_location("module.name", self.model_path)
    print(self.model_path)
    self.model_module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(self.model_module)
    self.model = self.model_module.model(input_shape = (self.height, self.width, self.channels), classes = self.no_classes, kernel_size = self.kernel_size, filter_depth = self.fil
    self.model.compile(optimizer = RMSprop(lr = self.lrate, decay = 1e-6), loss = self.loss, metrics = ['accuracy'])
    print("Modelo Cargado y compilado con exito")

### Funcion FIT() ###
"""
    entrenar un modelo de red neuronal utilizando uno de los generadores de datos segun si se aplica aumentacion de datos o no.
    Durante el entrenamiento, se verifica si la perdida en el conjunto de validacion mejora, y si no mejora
    el entrenamiento se detiene para evitar el sobreajuste.
"""
def fit(self):
    csv_logger = CSVLogger(self.save_folder + "/" + self.name + ".csv")
    save_path = self.name + ".h5"
    save_path = self.save_folder + "/" + save_path
    earlystop = EarlyStopping(monitor="val_loss", min_delta = 0, patience = 20, verbose = 1, mode = 'min')
    checkpoint = ModelCheckpoint(save_path, monitor = "val_loss", verbose = 1, save_best_only = True, save_weights_only = False, mode = "auto", period = 1)

# Depende de si se usa un dataset con aumento o sin aumento, se descomenta una linea u otra.
    #self.model.fit_generator(self.generator_with_augmentations(), steps_per_epoch = self.no_images // self.batch_size, epochs = self.no_epochs, callbacks = [csv_logger, checkpoint
    self.model.fit_generator(self.generator(), steps_per_epoch = self.no_images // self.batch_size, epochs = self.no_epochs, callbacks = [csv_logger, checkpoint, earlystop], valid
```

POSTPROCESSIONG.PY

Contiene el script que define la clase *postprocessing*. Realiza tareas de procesado tras el entrenamiento del modelo. Consigue todos los valores e información para poder realizar un análisis de rendimiento del modelo, como son los falsos positivos, verdaderos positivos, curva de entrenamiento y curva ROC entre otros.

A continuación se define el objetivo de cada función que componen a la clase *postprocessing*.

- ***_init_()***: Inicia la clase, carga un modelo y los parámetros de configuración necesarios, obtenidos de un archivo de tipo *H5*.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import h5py
import glob
import pandas as pd
import tensorflow as tf
import cv2
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img
from keras.models import Model, Sequential
from keras.layers import  *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.metrics import categorical_accuracy
from keras import backend as K
from keras import losses
from keras.models import load_model as keras_load_model
from keras.utils import to_categorical
from keras.preprocessing.image import ImageDataGenerator
import sys
from keras.utils.generic_utils import get_custom_objects
from sklearn.metrics import roc_curve, auc
from keras.optimizers import Adam


import tensorflow.compat.v1.keras.backend as K
import tensorflow as tf
tf.compat.v1.disable_eager_execution()

sys.path.insert(0, '../')



class PostProcessing:
    beam =  0
    tissue = 1
    bone = 2
    ### Funcion _INIT_() ###
    """
          Inicializa la clase, carga un modelo y los parametros de configuracion necesarios.
          Se obtienen de un archivo tipo H5
    """
    def __init__(self, model_path, dataset_path, loss = 'categorical_crossentropy', device = "cpu"):
        self.model_path = model_path
        self.dataset_path = dataset_path
        self.read_h5_file()
        print(loss)
        self.load_model(device = device, loss = loss)
        print('Modelo cargado.')
        self.prediction_prob_rs, self.prediction_argmax = self.predict(device=device)
```

- ***read_h5_file():*** Lee datos de un archivo *H5*, incluyendo imágenes, etiquetas y nombres de archivos, y reorganiza las etiquetas.
- ***load_model():*** Carga un modelo de red neuronal, lo compila especificando el dispositivo (CPU o GPU) a utilizar para la ejecución.
- ***predict():*** Realiza predicciones utilizando el modelo cargado en un dispositivo especifico para un conjunto de imágenes del dataset aportado.
- ***evaluate_overall():*** Evalúa el rendimiento del modelo con el set de imágenes de prueba, calculando la pérdida y la precisión. También ofrece el número de parámetros entrenables en el modelo.

```
### Funcion READ_H5_FILE() ###
"""
   Lee datos de un archivo H5, en este caso el dataset creado mediante aumento de datos
"""
def read_h5_file(self):
    dataset = h5py.File(self.dataset_path, 'r')
    self.train_images = dataset['train_img']
    self.test_images =  dataset['test_img'][:]
    self.val_images =  dataset['val_img'][:]
    self.train_labels = dataset['train_label']
    self.train_body = dataset['train_bodypart'][:]
    self.test_labels = dataset['test_label'][:]
    self.val_labels = dataset['val_label'][:]
    self.test_body = dataset['test_bodypart'][:]
    self.val_body = dataset['val_bodypart'][:]
    self.test_filenames = dataset['test_file'][:]
    self.val_filenames = dataset['val_file'][:]
    self.no_images_training, self.height, self.width, self.classes = self.train_labels.shape
    self.train_labels = np.reshape(self.train_labels, (-1,self.height*self.width ,self.classes))
    self.test_labels = np.reshape(self.test_labels, (-1,self.height*self.width ,self.classes))
    self.val_labels = np.reshape(self.val_labels, (-1,self.height*self.width ,self.classes))
    self.test_images = np.concatenate((self.test_images, self.val_images))
    self.test_labels = np.concatenate((self.test_labels, self.val_labels))
    self.test_filenames = np.concatenate((self.test_filenames, self.val_filenames))

    dataset.close()

### Funcion LOAD_MODEL() ###
"""
   Carga un modelo de red neuronal, lo compila.
   Se especifica el dispositivo a usar (cpu o la gpu) para la ejecucion
"""
def load_model(self, device = "cpu", optimizer = Adam(lr=1e-4), loss = "categorical_crossentropy",\
               metrics = ['accuracy'] ):
    if(device == "cpu"):
        with tf.device("/cpu:0"):
```

```
                self.model = keras_load_model(self.model_path)
                self.model.compile(optimizer, loss, metrics)
    elif(device == "gpu"):
            self.model = keras_load_model(self.model_path)
            self.model.compile(optimizer, loss, metrics)
    else:
        print("Dispositivo no entendido")
        return None

### Funcion PREDICT() ###
"""
   Realiza predicciones utilizando el modelo cargado en un dispositivo especificado (CPU o GPU) para un conjunto de imagenes dado
"""
def predict(self, device = "gpu", images = None):
    if(images is None):
        images = self.test_images
    if( device == "cpu"):
        with tf.device("/cpu:0"):
            prediction_prob = self.model.predict(images, batch_size=1)
    elif(device == "gpu"):
        prediction_prob = self.model.predict(images, batch_size=1)
    else:
        print("Dispositivo no encontrado")
        return None
    prediction_prob_rs = prediction_prob.reshape((-1,self.classes))
    prediction_argmax = np.argmax(prediction_prob_rs, axis = -1)
    if((prediction_argmax == 1).any()):
     print('predicciones_argmax1')
     print(prediction_argmax)
    return prediction_prob_rs, prediction_argmax

### Funcion EVALUATE_OVERALL() ###
"""
    Evalua el rendimiento del modelo en el set de test, calculando la perdida y la precision.
    Tambien cuenta el numero de parametros entrenables en el modelo
"""
def evaluate_overall(self, device = "gpu"):
    images = self.test_images
    labels = self.test_labels
    if(device == "cpu"):
        with tf.device("/cpu:0"):
            loss_test, accuracy_test = self.model.evaluate(images,labels, batch_size = 1)

    elif(device == "gpu"):
        loss_test, accuracy_test = self.model.evaluate(images,labels, batch_size = 1)
    else:
        print("Dispositivo no entendido")
        return None

    print("Precision general : \n")
    print ('En el conjunto de prueba {}%'.format(round(accuracy_test,2)*100))

    # Contador del numero de parametros entrenables
    trainable_count =  int(np.sum([K.count_params(p) for p in set(self.model.trainable_weights)]))
    print('Parametros entrenables: {:,}'.format(trainable_count))
    return accuracy_test, trainable_count
```

- ***tpfp()***: Calcula verdaderos positivos y falsos positivos basados en las predicciones del modelo. Serán usados para calcular la sensibilidad y la especificidad.

```
### Funcion TPFP() ###
"""
    Calcula verdaderos positivos y falsos positivos conseguidos mediante las predicciones del modelo.
    Se utilizan para el calculo de la sensibilidad y la especificidad
"""
def tpfp(self, predictions = None, single_index = -1):

    if (not (single_index == -1)):
        labels = self.test_labels[single_index]
        if(predictions is not None):
            prediction_argmax = predictions.reshape(-1,200,200)
        else:
            prediction_argmax = self.prediction_argmax.reshape(-1,200,200)

        prediction_argmax = prediction_argmax[single_index]
        prediction_argmax = prediction_argmax.flatten()
    else:
        if( predictions is not None):
            prediction_argmax = predictions
        else:
            prediction_argmax = self.prediction_argmax
        labels = self.test_labels

    labels = np.argmax(labels, axis = -1)
    labels = labels.flatten()

    beam_gt = np.where(labels == self.beam)[0]
    beam_pred = np.where(prediction_argmax == self.beam)[0]
```

```
    tissue_gt = np.where(labels == self.tissue)[0]
    tissue_pred = np.where(prediction_argmax == self.tissue)[0]

    if (len(tissue_pred) == 0):
        return 0,0


    bone_gt = np.where(labels == self.bone)[0]
    bone_pred = np.where(prediction_argmax == self.bone)[0]

    # FALSOS POSITIVOS
    false_positives = 0
    beam_as_tissue = float(len(np.intersect1d(beam_gt, tissue_pred, assume_unique=True)))/float(len(tissue_pred))
    false_positives = beam_as_tissue
    bone_as_tissue = float(len(np.intersect1d(bone_gt, tissue_pred, assume_unique=True)))/float(len(tissue_pred))
    false_positives += bone_as_tissue

    # VERDADEROS POSITIVOS

    true_positives = 0
    true_positives = len(np.intersect1d(tissue_gt, tissue_pred, assume_unique=True))/len(tissue_gt)

    # FALSOS NEGATIVOS

    false_negatives = 0
    tissue_as_beam = float(len(np.intersect1d(tissue_gt, beam_pred, assume_unique=True)))/float(len(tissue_gt))
    false_negatives = tissue_as_beam
    tissue_as_bone = float(len(np.intersect1d(tissue_gt, bone_pred, assume_unique=True)))/float(len(tissue_gt))
    false_negatives += tissue_as_bone


    return true_positives, false_positives


### Funcion THRESHOLDING() ###
"""
   Aplica un umbral a las predicciones del modelo para clasificar los pixeles como tejido o no tejido
"""
def thresholding(self,threshold, device = "cpu"):
    prob_prediction_tissue = self.prediction_prob_rs[...,self.tissue]
    tissue_pred = np.where((prob_prediction_tissue > threshold))[0]

    prediction_improved = self.prediction_argmax
    prediction_improved[tissue_pred] = self.tissue

    tissue_notsure = np.where((prob_prediction_tissue <= threshold))[0]
    openbeam_bone = self.prediction_prob_rs[...,[0,2]]
    prediction_improved[tissue_notsure] = 2 * np.argmax(openbeam_bone[tissue_notsure], axis = -1)
    self.prediction_argmax = prediction_improved
    return prediction_improved
```

- ***pixel_dilation():*** Dilata los píxeles de psoriasis en las predicciones, permitiendo mejorar ciertas zonas en la salida de segmentación.

```
### Funcion PIXEL_DILATION() ###
"""
   Dilata los pixeles de psoriasis y piel sana en las predicciones, mejorando ciertas regiones en la salida de segmentacion
"""
def pixel_dilation(self, dilation_factor, predictions = None, both = False):

    if( predictions is None):
        _, predictions = self.predict()
    predictions = predictions.reshape((-1, self.height, self.width))
    predictions = predictions.astype(np.float32)
    predictions_dilated = np.ones_like(predictions)

    prediction_bone = np.zeros_like(predictions)
    bone_indices = np.where(predictions == self.bone)
    prediction_bone[bone_indices] = self.bone
    prediction_bone = prediction_bone.reshape((-1, self.height, self.width))
    prediction_bone = prediction_bone.astype(np.float32)
    prediction_bone_dilated = np.zeros_like(predictions)

    for i,prediction in enumerate(prediction_bone):
        #elimina grupos chcios de psoriasis
        kernel_opening = np.ones((10,10), np.uint8)
        bone_pred = np.where(prediction == self.bone)
        opening = cv2.morphologyEx(prediction, cv2.MORPH_OPEN, kernel_opening)

        #dilatacion de imagen
        kernel_dilate = np.ones((dilation_factor, dilation_factor), np.uint8 )
        dilated = cv2.dilate(opening, kernel_dilate)
        prediction_bone_dilated[i,...] = dilated

    predictions_dilated[np.where(prediction_bone_dilated == self.bone)] = self.bone
    predictions_dilated[np.where(predictions == self.beam)] = self.beam

    return predictions_dilated
```

- ***plot():*** Genera gráficos de las imágenes originales, etiquetas de referencia, predicciones del modelo, mapas de probabilidad, predicciones con umbral y predicciones dilatadas para imágenes de prueba.

```
### Funcion PLOT() ###
"""
   Genera graficos de las imagenes originales, etiquetas de referencia, predicciones del modelo,
   predicciones con umbral y predicciones dilatadas para imagenes de prueba
"""
def plot(self,threshold, dilation_factor):
    probability_map, prediction = self.predict()
    prediction_threshold = self.thresholding(0.9)
    prediction_dilation = self.pixel_dilation(dilation_factor, prediction_threshold)
    ntestimages = len(self.test_images)
    left  = 0.1  # lado izquierdo de las subgraficas de la figura
    right = 0.4    # lado izquierdo de las subgraficas de la figura
    bottom = 0.1   # parte inferior de las subgraficas de la figura
    top = 0.9      # parte superior de las subgraficas de la figura
    wspace = 0.08   # ancho del margen reservado para espacio en blanco entre las subgraficas
    hspace = 0.1   # altura del margen reservado para espacio en blanco entre las subgraficas

    labels_plot = self.test_labels.reshape(-1, self.height, self.width, 3) * 255
    prediction = prediction.reshape(-1, self.height, self.width)
    prediction_threshold = prediction_threshold.reshape(-1, self.height, self.width)
    prediction_dilation = prediction_dilation.reshape(-1, self.height, self.width)

    for i, image in enumerate(self.test_images):
        print(self.test_filenames[i])
        fig=plt.figure(figsize=(50, 50), dpi= 80, edgecolor='k',frameon=False)
        plt.subplots_adjust(left=left, bottom=bottom, right=right, top=top, wspace=wspace, hspace=hspace)
        index_show = i
        print(i)
        plt.subplot(ntestimages,5,1)
        plt.title('Image')
        plt.imshow(image[...,0],cmap='gray')
        plt.axis('off')

        plt.subplot(ntestimages,5,2)
        plt.title('Ground truth')
        plt.imshow(labels_plot[i])
        plt.axis('off')

        plt.subplot(ntestimages,5,3)
        plt.title('Prediction')
        plt.imshow(prediction[i])
        plt.axis('off')

        plt.subplot(ntestimages,5,4)
        plt.title('Threshold')
        plt.imshow(prediction_threshold[i])
        plt.axis('off')

        plt.subplot(ntestimages,5,5)
        plt.title('Dilated')
        plt.imshow(prediction_dilation[i])
        plt.axis('off')
        plt.show()
```

- ***learning_curve():*** Para poder evaluar el progreso en el entrenamiento se generan las curvas de aprendizaje para el error de entrenamiento y validación, así como la pérdida de entrenamiento y validación.
- ***ROC_curve():*** Calcula y traza la curva de región de convergencia, esta representa la relación entre el número de verdaderos positivos y el número de falsos positivos, como una función del nivel de corte para la segmentación de clase de tejido.

```
### Funcion LEARNING_CURVE() ###
"""
   Genera curvas de aprendizaje para el error de entrenamiento y validacion, se incluye la perdida de entrenamiento y validacion,
   Sirve para Evaluar el progreso del entrenamiento del modelo
"""
def learning_curve(self, path_to_csv):

    csv_file = pd.read_csv(path_to_csv)
    self.csv = csv_file
    epochs = self.csv['epoch']
    train_loss = self.csv['loss']
    val_loss = self.csv['val_loss']
    train_acc = self.csv['accuracy']
    val_acc = self.csv['val_accuracy']

    train_err = 1 - train_acc
    val_err = 1 - val_acc
    fig, ax = plt.subplots(2,1, figsize=(15,15))
    ax[0].plot(epochs, train_err, color = 'blue', label = 'training error')
    ax[0].plot(epochs, val_err, color = 'orange', label = 'validation error')
    ax[0].plot(epochs, np.linspace(0.02,0.02,len(epochs)), color = 'green', label = 'desired error')
    ax[0].set_xlabel('number of epochs')
    ax[0].set_ylabel('error')
    ax[0].set_title('Error')
    ax[0].legend()

    ax[1].plot(epochs, train_loss, label = "training loss")
    ax[1].plot(epochs, val_loss, label = "validation loss")
    ax[1].legend()
    ax[1].set_xlabel("Number of epochs")
    ax[1].set_ylabel("Loss")
```

```
    ax[1].set_title("Loss")

    return fig, ax



### Funcion ROC_CURVE() ###
"""
   Calcula y traza una curva Caracteristica Region de Convergencia (ROC) para la segmentacion de la clase de tejido.
"""
def ROC_curve(self):

    fpr, tpr, thresholds = roc_curve(self.test_labels[..., 1].reshape(-1), self.prediction_prob_rs[...,1].reshape(-1))
    roc_auc = auc(fpr, tpr)
    fig, ax = plt.subplots()
    ax.plot(fpr, tpr,
            label='Tissue ROC curve (area = {0:0.2f})'
                  ''.format(roc_auc),
            color='indianred', linestyle=':', linewidth=4)
    ax.plot([0, 1], [0, 1], 'k--')
    ax.set_xlim([0.0, 1.0])
    ax.set_ylim([0.0, 1.05])
    ax.set_xlabel('False Positive Rate')
    ax.set_ylabel('True Positive Rate')
    ax.set_title('Some extension of Receiver operating characteristic to multi-class')
    ax.legend(loc="lower right")
    return fig, ax
```