Universitat Oberta de Catalunya

uoc.edu

Estudio de técnicas de *machine learning* para el diagnóstico del melanoma y otras lesiones cutáneas a partir de imágenes

Agustín Miguel Barba Sánchez
Máster Universitario en Ingeniería Informática
Inteligencia Artificial

Longlong Yu
Carles Ventura Royo

Diciembre 2021

"A Ana, Lúa e Iris, por darme o tempo para esta viaxe"
"A meus país, por darme os zapatos para o camiño"

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Estudio de técnicas de machine learning para el diagnóstico del melanoma y otras lesiones cutáneas a partir de imágenes*
Nombre del autor:	*Agustín Miguel Barba Sánchez*
Nombre del consultor/a:	*Longlong Yu*
Nombre del PRA:	*Carles Ventura Royo*
Fecha de entrega (mm/aaaa):	12/2021
Titulación:	*Máster Universitario en Ingeniería Informática*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Deep Learning, Imagen médica, GANs*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo.*

El cáncer de piel es el tipo de cáncer más común y aunque el melanoma representa únicamente el 1% de este cáncer, es uno de los más mortales, especialmente si se detecta en estados avanzados. Un diagnóstico precoz permitiría aumentar las opciones de su tratamiento y la supervivencia de los pacientes.

Este diagnóstico se realiza mediante la inspección visual de las lesiones, con la medición de parámetros que son potencialmente detectables por un sistema de visión artificial, como el tamaño, color y forma. Esto abre la puerta a poder contar con sistemas automáticos para el diagnóstico de lesiones de piel.

El objetivo de este trabajo es la implementación de un clasificador automático de lesiones de piel. Para esto se ha utilizado el dataset HAM 10000, un conjunto de imágenes dermatoscópicas compilado con el propósito entrenar este tipo de sistemas.

Para ello se ha partido de un prototipo inicial basado en el modelo EfficientNet y se han aplicado distintas técnicas para mejorar la respuesta del sistema. Además, con el fin de mejorar el entrenamiento del modelo, se ha aumentado el conjunto de datos original mediante el uso de redes generativas adversarias.

El resultado de este trabajo es que partiendo de un resultado inicial de 0.34 de f1-score y 0.63 de accuracy, se ha logrado mejorar hasta un 0.75 de f1-score y 0.86 de accuracy.

Este resultado es similar al obtenido por expertos humanos, por lo que podría ser utilizado como ayuda al diagnóstico y a la toma de decisiones.

Abstract (in English, 250 words or less):

Skin cancer is the most common type of cancer and although melanoma accounts for only 1% of skin cancer, it is one of the deadliest, especially if detected in advanced stages. An early diagnosis would allow increasing the options of its treatment and patient survival.

This diagnosis is made by visual inspection of the lesions, measuring parameters that are potentially detectable by a computer vision system, such as size, colour and shape. This opens the door to automatic systems for the diagnosis of skin lesions.

The objective of this work is the implementation of an automatic skin lesion classifier. For this purpose, we used the HAM 10000 dataset, a set of dermatoscopic images compiled to train this type of systems.

To do this, we have started from an initial prototype based on the EfficientNet model and then different techniques have been applied to improve the system's response. In addition, to improve the training of the model, the original dataset has been augmented using generative adversarial networks.

The result of this work is that starting from an initial result of 0.34 for the f1-score and 0.63 for the accuracy, it has been improved to 0.75 for the f1-score and 0.86 for the accuracy.

This result is similar to that obtained by human experts, so it could be used as an aid to diagnosis and decision-making.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

La mayoría de los autores coinciden en situar el llamado boom del *machine learning* y de la IA [1] en el año 2012, con el nacimiento de AlexNet [2].

El uso nuevas técnicas de aprendizaje en las redes convolucionales junto con una serie de mejoras del hardware durante estos últimos años, han hecho que las aplicaciones de la IA se empezaran a utilizar en todos los ámbitos y hoy podemos decir que están en todas partes.

Uno de los campos en los cuales la IA y en particular el *machine learning* y el *deep learning* están teniendo mejores resultados, es en la Medicina.

En los últimos 30 años se han comenzado a utilizar herramientas para el diagnóstico asistido para ayudar a la toma de decisiones médicas. Estas herramientas se basan en técnicas de inteligencia artificial que son capaces de dar diagnósticos con porcentajes de éxito muy elevados, lo que permite pensar en establecer, no dentro de mucho tiempo, sistemas de detección temprana de enfermedades basadas en estos sistemas. Esto, en conjunto con otras innovaciones esperadas a corto y medio plazo como el 5G y el *IoT* hacen pensar en una revolución en la Medicina muy cercana.

No es descabellado pensar en dispositivos ponibles recogiendo datos de su portador y emitiendo alertas cuando se detecten patrones que la IA reconoce como patogénicos; o en aplicaciones de móvil que tengan lo que se denomina en medicina “ojo clínico” y que sean capaces de realizar diagnósticos rápidos y correctos a partir de imágenes tomadas desde el propio dispositivo por el usuario.

Dentro de la diagnosis médica automática, el campo de la dermatología siempre ha estado a la cabeza en cuanto a avances en el uso de la inteligencia artificial. Desde que en 1987 Cascinelli, Ferrario, Tonelli y Leo propusieron el uso de los ordenadores en el diagnóstico clínico del melanoma [3], han sido numerosos los avances en esta empresa, como, por ejemplo, los esfuerzos de Binder et al. que en 1994 lograron entrenar una red de neuronas artificiales para la clasificación de melanomas utilizando imágenes [4].

El procedimiento habitual de diagnosis del cáncer de piel y del resto de patologías relacionadas, se basa en la inspección visual del tipo de lesión, utilizando la medición de parámetros como el tamaño, color y forma. Este tipo de patrones son detectables por un sistema de visión artificial, de ahí la idoneidad de los ordenadores para la detección de enfermedades de la piel.

El cáncer de piel es el más común de todos los cánceres y aunque el melanoma representa apenas el 1% de los casos dentro de esta variedad [5], es uno de los más mortales [6], y el devenir de la enfermedad depende en gran medida de su diagnóstico temprano. Contar con sistemas de diagnóstico automático que pudieran detectar el cáncer en sus primeras fases, cuando son difíciles de

distinguir de otros tipos de lesiones de piel más benignas, sería de una importancia capital a la hora de salvar miles de vidas. La supervivencia a 5 años de un paciente con un melanoma asciende al 93%, si se detecta en sus primeras fases [5].

Dado el interés del ámbito descrito, en este trabajo se propone un estudio sobre diferentes técnicas de aprendizaje automático, en particular, de aprendizaje profundo, que se pueden aplicar para conseguir un sistema de diagnóstico automatizado de lesiones de la piel.

Otro elemento para tener muy en cuenta cuando se utilizan técnicas de *deep learning*, son las grandes necesidades computacionales que el entrenamiento de estos sistemas exige. Hoy en día, debido a la creciente demanda en el minado de criptomonedas [7], las GPUs necesarias para el entrenamiento de las redes neurales tienen unos precios muy elevados, por lo que contar con la infraestructura necesaria para abordar un estudio de este calibre tiene un coste muy alto.

Por lo tanto, para la realización de todo el trabajo se ha utilizado únicamente las versiones gratuitas de las plataformas Google Colab y Google Drive.

1.2 Objetivos del Trabajo

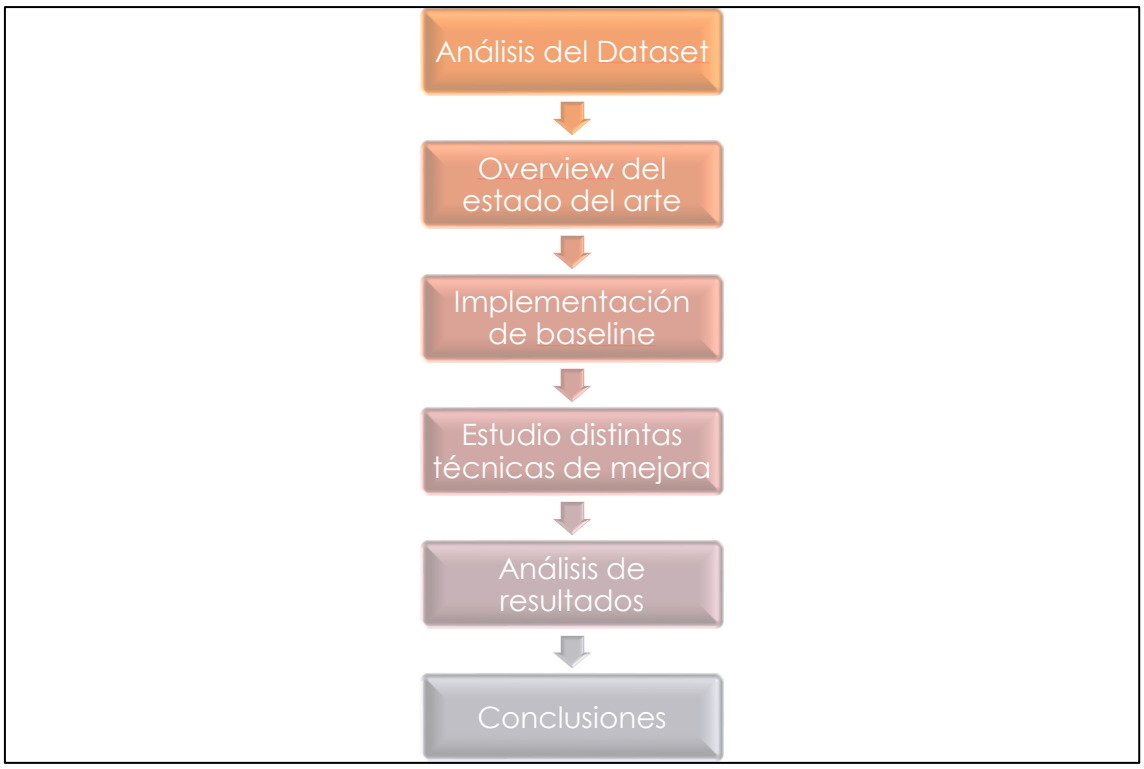
El objetivo de este trabajo de fin de máster es el estudio e implementación de diferentes técnicas de aprendizaje profundo para resolver un problema del diagnóstico de imagen médica.

A partir de la implementación de una solución inicial que ha servido de línea base, se ha estudiado como se podía mejorar sus resultados. Bien con cambios en su arquitectura como mediante la optimización de los conjuntos de datos empleados para el entrenamiento, utilizando entre otras, técnicas de balanceo de datos y modelos generativos.

1.3 Enfoque y método seguido

La línea de trabajo parte del conjunto de datos HAM 10000, publicado en 2018 [8] como parte de una iniciativa para desarrollar sistemas que pudieran aportar en el diagnóstico precoz del melanoma.

Sobre este *dataset*, se realiza un estudio del problema, se implementa una solución de línea base y se implementan y evalúan distintas técnicas para mejorar el comportamiento del modelo.

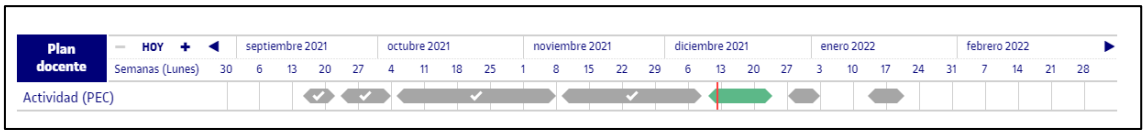


1. Diagrama de flujo del trabajo realizado
Fuente: Elaboración propia

Finalmente se detallan las conclusiones del trabajo, las reflexiones sobre los logros obtenidos y se comentan las líneas de trabajo futuro que se podrían seguir en relación con la temática que se ha descrito.

1.4 Planificación del Trabajo

La planificación que se ha estimado sigue el marco de las entregas que se han sugerido desde la asignatura para la realización del trabajo de fin de Máster:



2. Planificación de la asignatura
Fuente: www.uoc.edu Acceso 23/12/21

Hitos de la asignatura

PEC 0
- Elección de la temática del TFM
- Definición de los contenidos del trabajo

PEC 1
- Elaborar un plan de trabajo
- Escribir plan de trabajo
- Elaborar la planificación

PEC 2
- Estado del arte sobre técnicas de machine learning para clasificación de imágenes
- Análisis de datos
    - Histograma de categorías
    - Rango de valores de RGB

- Estudio de la métrica de evaluación
- Diseño de la solución
- Selección de hardware y software
- Estudio de la técnica a implementar

- Implementación de una prueba de concepto (primera solución)
- Análisis de la solución
- Escribir memoria de fase 1

PEC 3
- Mejora iterativa de la solución
    - Estudio de las mejoras para la solución propuesta
    - Aplicar las técnicas seleccionadas para mejorar la solución
    - Análisis de la mejora obtenida

- Escribir memoria de fase 2

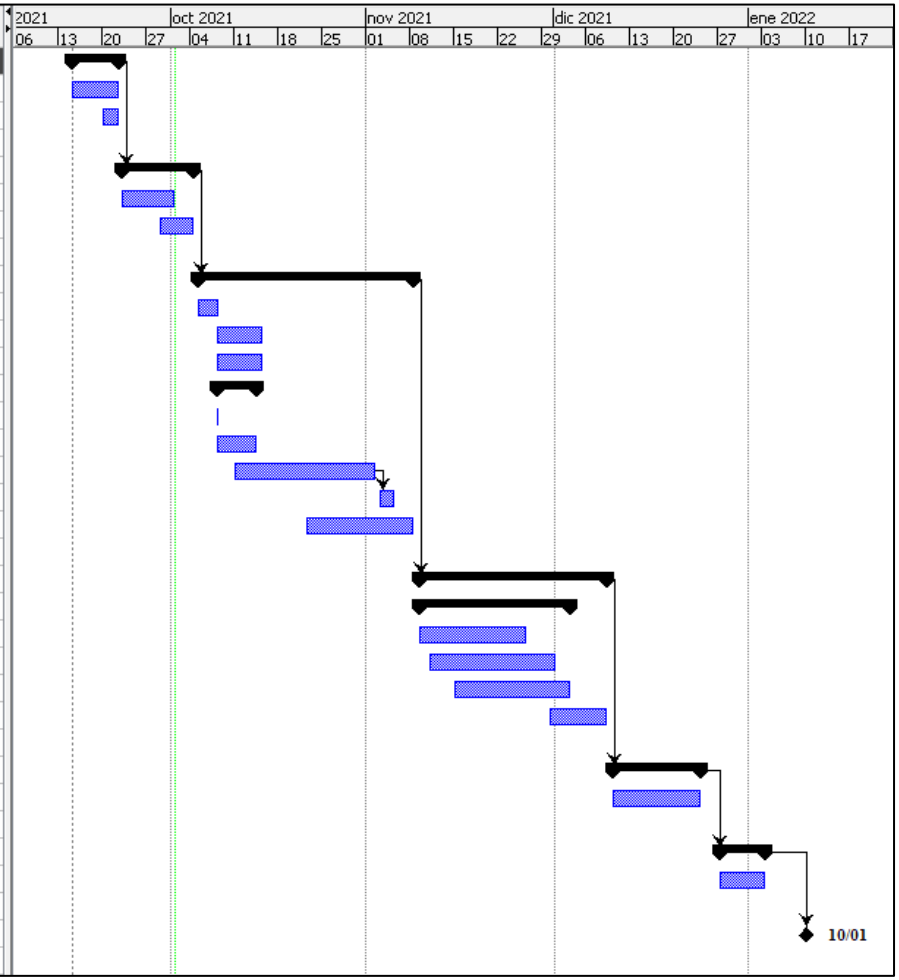
PEC 4
- Redacción de la memoria final

PEC 5a
- Elaborar presentación

PEC 5b
- Presentar TFM

Diagrama de Gantt de la planificación del trabajo

	Nombre	Inicio	Terminado
1	**Inicio (PEC 0)**	**15/09/21 8:00**	**22/09/21 17:00**
2	Elección temática TFM	15/09/21 8:00	22/09/21 17:00
3	Definición de los contenidos del trabajo	20/09/21 8:00	22/09/21 17:00
4	**Elaboración del plan de trabajo (PEC 1)**	**23/09/21 8:00**	**4/10/21 17:00**
5	Escribir plan de trabajo	23/09/21 8:00	1/10/21 17:00
6	Planificación	29/09/21 8:00	4/10/21 17:00
7	**Desarrollo del trabajo Fase 1 (PEC 2)**	**5/10/21 8:00**	**8/11/21 14:00**
8	Estado del arte	5/10/21 8:00	8/10/21 17:00
9	Analisis de datos	8/10/21 8:00	15/10/21 17:00
10	Estudio métrica evaluación	8/10/21 8:00	15/10/21 17:00
11	**Diseño de la solución**	**8/10/21 8:00**	**14/10/21 17:00**
12	Selección de hardware y software a utilizar	8/10/21 8:00	8/10/21 17:00
13	Estudio de la técnica a emplear	8/10/21 8:00	14/10/21 17:00
14	Implementar una prueba de concepto	11/10/21 8:00	2/11/21 17:00
15	Análisis de la solución	3/11/21 8:00	5/11/21 17:00
16	Escribir memoria de la Fase 1	22/10/21 15:00	8/11/21 14:00
17	**Desarrollo del trabajo Fase 2 (PEC 3)**	**9/11/21 14:00**	**9/12/21 14:00**
18	**Búsqueda iterativa de mejoras**	**9/11/21 14:00**	**3/12/21 17:00**
19	Estudio de mejoras a aplicar	9/11/21 14:00	26/11/21 17:00
20	Aplicar técnicas seleccionadas para mejorar el prototipo	11/11/21 8:00	1/12/21 10:00
21	Análisis de la mejora obtenida	15/11/21 8:00	3/12/21 17:00
22	Escribir memoria de la Fase 2	30/11/21 8:00	9/12/21 14:00
23	**Redacción de la memoria (PEC 4)**	**10/12/21 10:00**	**24/12/21 10:00**
24	Elaborar la memoria del TFM	10/12/21 10:00	24/12/21 10:00
25	**Preparar presentación (PEC 5)**	**27/12/21 10:00**	**3/01/22 17:00**
26	Elaborar presentación	27/12/21 10:00	3/01/22 17:00
27	Presentación	10/01/22 8:00	10/01/22 17:00



3.Planificación del TFM
Fuente: Elaboración propia

1.5 Productos obtenidos

En la primera fase del trabajo se ha implementado un sistema base (*baseline*) capaz de clasificar las imágenes del conjunto de prueba en una de las 7 categorías posibles.

En la segunda fase se han implementado sucesivas mejoras del sistema para mejorar su comportamiento.

Además, en la segunda fase se ha implementado un sistema independiente formado por dos redes neurales formando una GAN.

Con esta GAN se ha generado un nuevo conjunto de imágenes (33.000) con los que aumentar el *dataset* original y mejorar el entrenamiento de la red original.

1.6 Descripción de la memoria

En el tema 2 se describirá el problema abordado y se realizará un estado del arte sobre la temática. También se presentará el conjunto de datos seleccionado y se relatarán los medios dispuestos para realizar el trabajo.

En el tema 3 se explicará la solución base implementada para resolver el problema.

En el tema 4 se describirán las distintas mejoras que se han implementado para intentar mejorar el desempeño del sistema base descrito en el tema 3.

En el tema 5 se relatarán las conclusiones a las que el autor ha llegado al realizar este trabajo, y cuáles son los trabajos pendientes o líneas de investigación que se podrían seguir para continuar esta investigación.

En el tema 6 se presentará un glosario de los términos empleados en el trabajo

En el tema 7 se listará la bibliografía de trabajo

Para finalizar en el apartado de Anexos se presentará todo el código fuente utilizado en este trabajo.

2. El Problema

2.1 Estado del arte

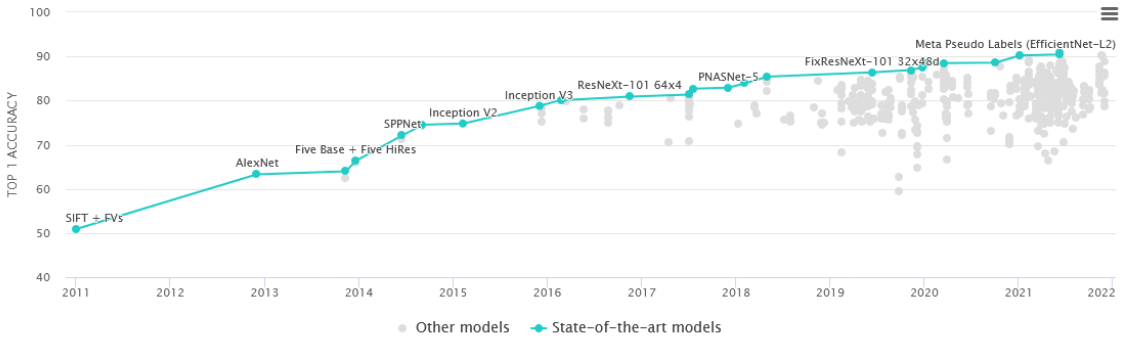
Las CNN y en particular las DCNNs (*Deep convolutional neural networks*) son el estado del arte en el diagnóstico automático de lesiones de piel mediante el uso de imágenes [9][10].

Cada año se realizan distintos desafíos donde se pone a prueba la eficacia de estas redes compitiendo contra humanos. Uno de los eventos más importantes es el ILSVRC (*ImageNet Large Scale Visual Recognition Challenge*) y ya en 2015 las máquinas lograron mejorar a los humanos en ratio de reconocimiento de imágenes [11].

Para este desafío en particular se utiliza el conjunto de entrenamiento ImageNet [12], que consta de 1,2 millones de imágenes clasificados en 1.000 categorías. El problema consiste en utilizar dicho conjunto y entrenarlo, para después probarlo con un conjunto de prueba de 150.000 imágenes.

En 2021, el actual número uno en el ranking de Exactitud (*accuracy*) en este dataset es CoAtNet-7 [13], del Google Research Brain Team con una puntuación de *accuracy* de 90.88%

En la imagen al pie se puede ver la evolución de las puntuaciones de diferentes sistemas durante los últimos 10 años.



4. Modelos estado del arte en ImageNet
Fuente: https://paperswithcode.com/sota/image-classification-on-imagenet, Acceso: 18/12/21

2.2 El conjunto de datos HAM 10000

Cuando hablamos de aplicar técnicas de *machine learning* y en particular de *deep learning* a ámbitos relacionados con la biotecnología, la primera limitación que surge es la falta de datos, debido a la dificultad de su adquisición o a problemas de confidencialidad.

En particular cuando hablamos de imagen médica, además tenemos otras dificultades añadidas, como la protección especial de este tipo de datos o la calidad y heterogeneidad de las imágenes utilizadas.

En este contexto surge el conjunto de datos HAM 10000 (*Human Against Machine*), publicado en 2018 [14].

El HAM 10000 se publica en un esfuerzo por dotar a los investigadores de un conjunto de datos que se puedan utilizar en la investigación y que fomentar así un avance en el diagnóstico automático de lesiones malignas de la piel, con atención especial al melanoma.

El conjunto de datos HAM 10000, consiste en 10015 imágenes etiquetadas con un diagnóstico de una de las siguientes 7 categorías:

- AKIEK: *Actinic Keratoses (Solar Keratoses) and Intraepithelial Carcinoma (Bowen's disease*). Son variantes comúnmente no invasivas del carcinoma de células escamosas de la piel que pueden ser tratadas sin cirugía. Muchos autores no los consideran carcinomas si no precursores de estos.

- BCC: *Basal Cell Carcinoma*. Es una variedad común del cáncer epitelial de piel que no tiene tendencia a metastatizar, pero si a crecer sin control.

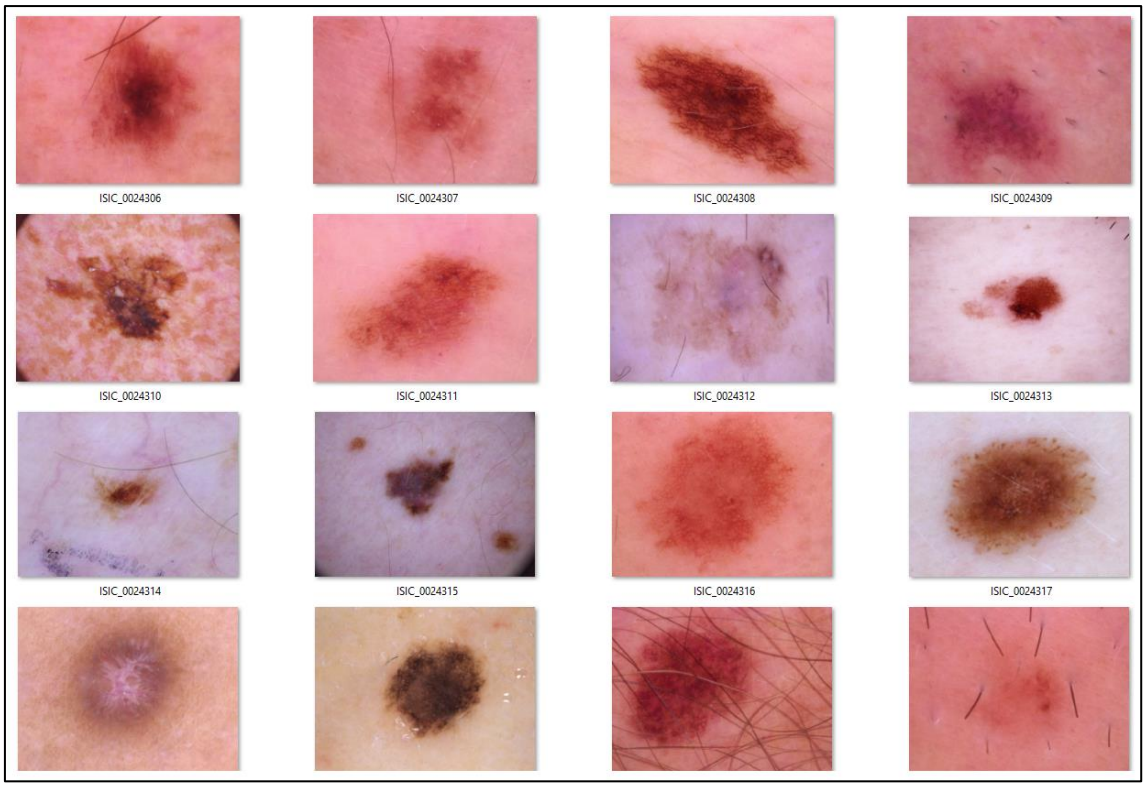
- BKL: *Bening Keratosis and Lichen-planus like keratoses. La queratosis benigna en una clase en la que se incluyen diferentes queratosis seborreicas benignas.*

- DF: *Dermatofibroma. Es una lesión benigna asociada a un trauma ligero.*

- NV: *Melanocitic Nevi. Los nevus melanocíticos son neoplasmas benignos muy comunes. (Lunares comunes)*

- MEL: *Melanoma.* El melanoma es un neoplasma maligno que a diferencia de los nevus, se presenta asimetrico en cuanto a su color y estructura.

- VASC: *Vascular Skin Lesions. En esta clase aparecen los hemangiomas, que son lesiones vasculares benignas.*



5. *Ejemplo de imágenes del HAM 10000*
Fuente: Elaboración propia

El HAM 10000 ha sido utilizado en el Challenge 2018 del ISIC (International Skin Imaging Collaboration) en un esfuerzo promovido por la *International Society for Digital Imaging of the Skins* [15]. Este desafío consistía en 3 tareas:

1. Segmentación de lesiones
2. Detección de atributos de las lesiones
3. Clasificación de enfermedades.

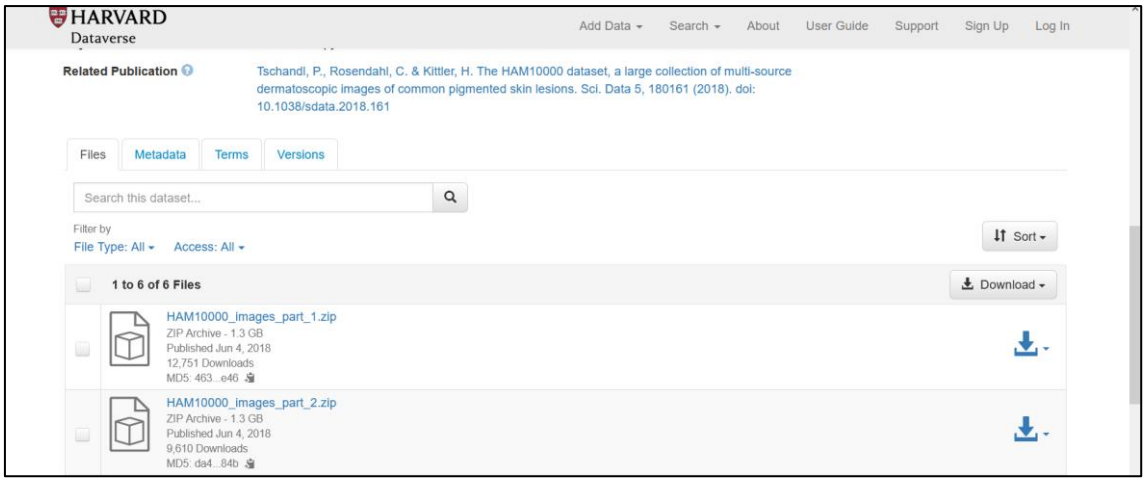
Las principales diferencias de este desafío con otros como puede ser el anteriormente comentado ILSVRC es por un lado el tipo de imagen utilizada y por otro, el tamaño de los conjuntos de datos.

En el Challenge 2018 del ISIC, las imágenes que se utilizan son dermatoscópicas.

La dermatoscopia es una técnica de detección de imágenes en la que se elimina el reflejo de la luz en la superficie de la piel, lo que permite que los expertos detecten detalles importantes en el diagnóstico que no se verían en una fotografía normal [16].

Además de la diferencia en el tipo de imágenes, el conjunto de datos que se emplea es pequeño comparado con ImageNet en un factor de más de 100 a 1.

El *dataset* HAM10000 está disponible en la web Hardvard Dataverse Repository en forma de 2 ficheros comprimidos tipo zip con un tamaño de 1.3 GB cada uno, además de otros ficheros complementarios con metadatos e imágenes de apoyo que no se han utilizado en este trabajo.



6. Página de descarga del HAM 10000
Fuente: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/DBW86T, Acceso: 20/12/21

Debido a las limitaciones de medios expuestas al inicio de esta memoria, y a utilizar plataformas como Google Colab y Google drive, el hecho de contar con un conjunto de datos relativamente reducido, que se pueda utilizar sin demasiados problemas de forma online, lo hacía el candidato ideal para el trabajo.

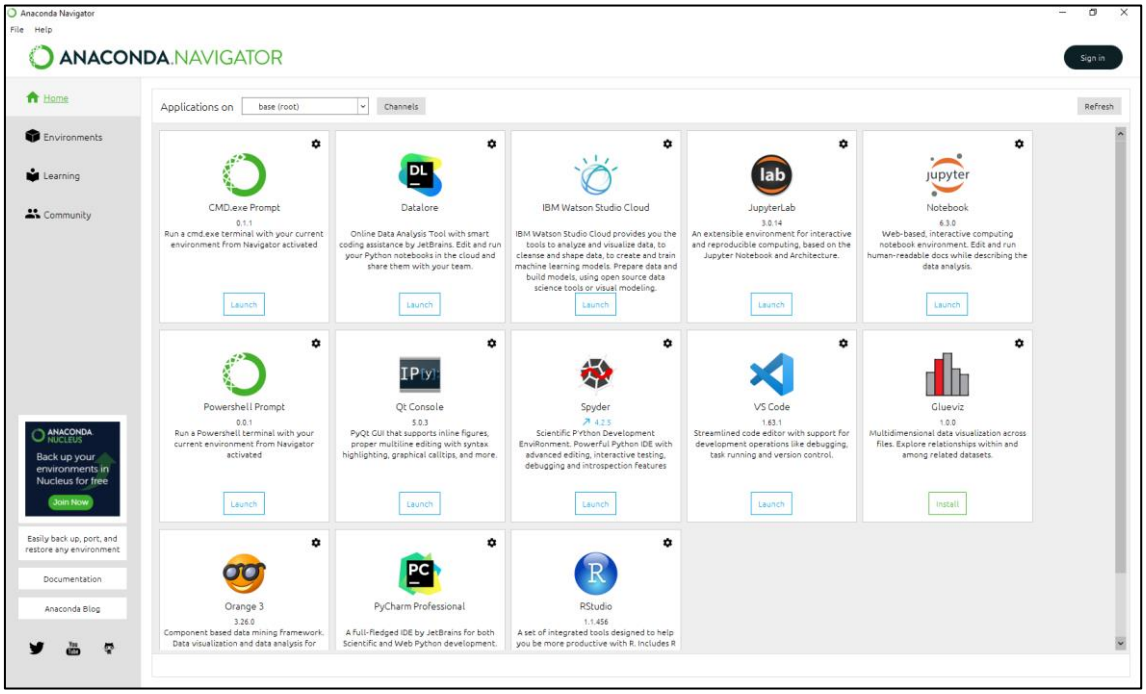
2.3 Medios utilizados

Google Colaboratory o Colab [17] es una plataforma de Google que permite la ejecución de código Python en una máquina remota utilizando un navegador. En particular, lo que se utiliza en Google Colab son documentos tipo Jupyter Notebook [18]. Esto permite generar código ejecutable apoyado por texto y recursos gráficos y multimedia, lo que lo hace una herramienta muy útil para la investigación y la docencia.

Sin embargo, la característica de Colab que la hace más interesante para este trabajo es que ofrece acceso a recursos hardware como GPUs, utilizados en este proyecto o las novedosas unidades de procesamiento tensorial o TPUs, desarrolladas por Google específicamente para utilizarlas en el aprendizaje automático con redes de neuronas artificiales [19].

7. Página de bienvenida de Google Colab.
Fuente: https://colab.research.google.com Acceso: 19/12/21

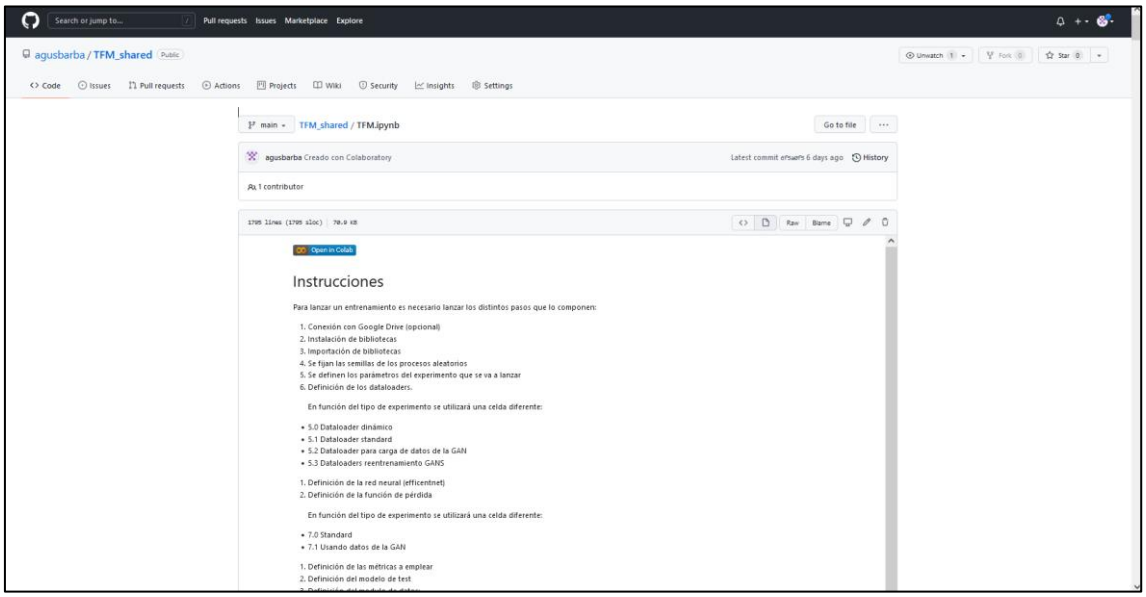
Además de Colab, se ha utilizado una instalación local de Anaconda 3 [20], para realizar operaciones sobre las imágenes de los conjuntos de datos. Anaconda es una distribución de Python y R que contiene numerosas bibliotecas y aplicaciones diseñados para ciencia de datos y aprendizaje automático. En este trabajo se han utilizado los IDEs de Python, Spyder 4.2.5 y Jupyter Notebook 6.3.0.



8. Página de bienvenida de Anaconda
Fuente: Instalación local

Como repositorio de la información utilizada, tanto de las imágenes de los conjuntos de entrenamiento y pruebas, como el código fuente se ha utilizado

Google Drive. Como repositorio de las distintas versiones del código fuente también se ha utilizado la plataforma Github.



9. Pantalla principal del proyecto en Github
Fuente: https://github.com/agusbarba/TFM_shared/blob/main/TFM.ipynb, Acceso 23/12/21

2.4 Otras consideraciones

Existen numerosas técnicas que se pueden utilizar para mejorar el desempeño de una red neural en un *dataset* como el HAM 10000.

Este TFM se ha centrado en el uso de redes generativas adversarias (GAN) para mejorar el conjunto de entrenamiento utilizando *data augmentation*. Por lo tanto, para comprobar su efectividad, se ha descartado el uso de otras técnicas de aumento de datos que podrían ser complementarias.

3. La línea base

3.1 EfficientNet

Como red neural base se ha optado por una EfficientNet, versión b0.

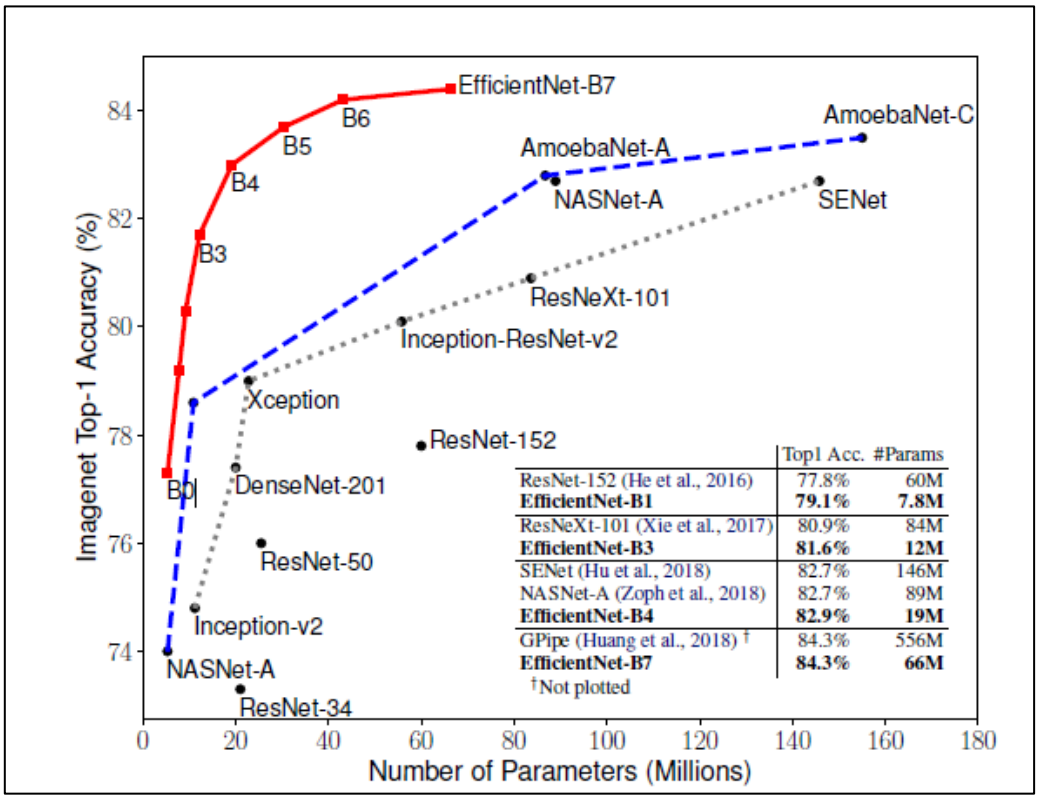
EfficientNet es un tipo de redes neurales convolucionales que nace en 2019 con el objetivo de mejorar las capacidades de escalado de las redes.[21]

La idea es poder desarrollar sistemas a bajo coste y que después si el sistema es prometedor se pueda escalar y ese escalado produzca una mejora lineal o casi lineal en su comportamiento.

En este caso, el funcionamiento de EfficientNet se basa en el escalado de todas las dimensiones, profundidad, anchura y resolución mediante un coeficiente.

Ya que este trabajo parte con recursos muy limitados, se propone partir con la versión b0, con el posible trabajo futuro de escalar el modelo base para mejorar los resultados.

Como se puede apreciar en el siguiente gráfico, EfficientNet en su versión b0 ya obtiene resultados muy aceptables en ImageNet, con un Accuracy del 77%, pero al escalarlo hasta b7, este valor sube hasta 84%, marcando el estado del arte en la fecha de su publicación.

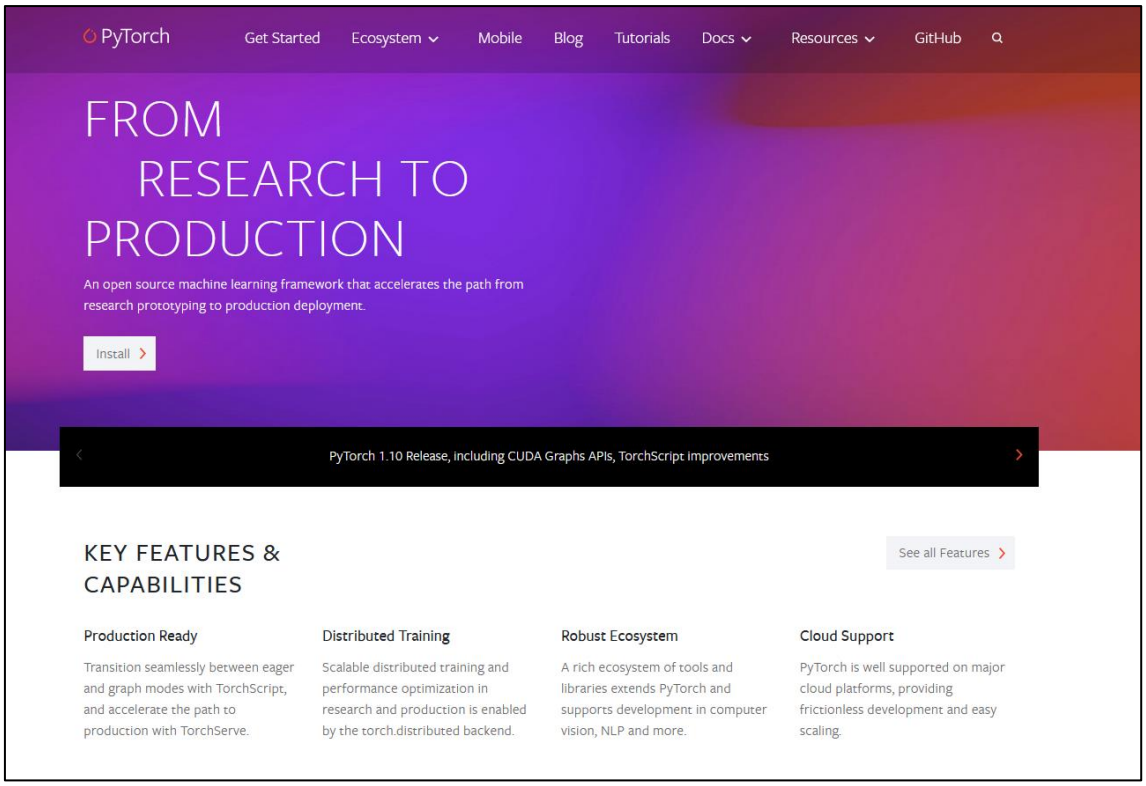


10. Tamaño en número de parámetros de algunas redes populares
Fuente: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. Mingxing Tan, Quoc V. Le, Acceso: 17/12/21

3.2 Frameworks

Como biblioteca principal para implementar el aprendizaje automático de la red, se ha utilizado el *framework* de código libre basado en Python, PyTorch [22].

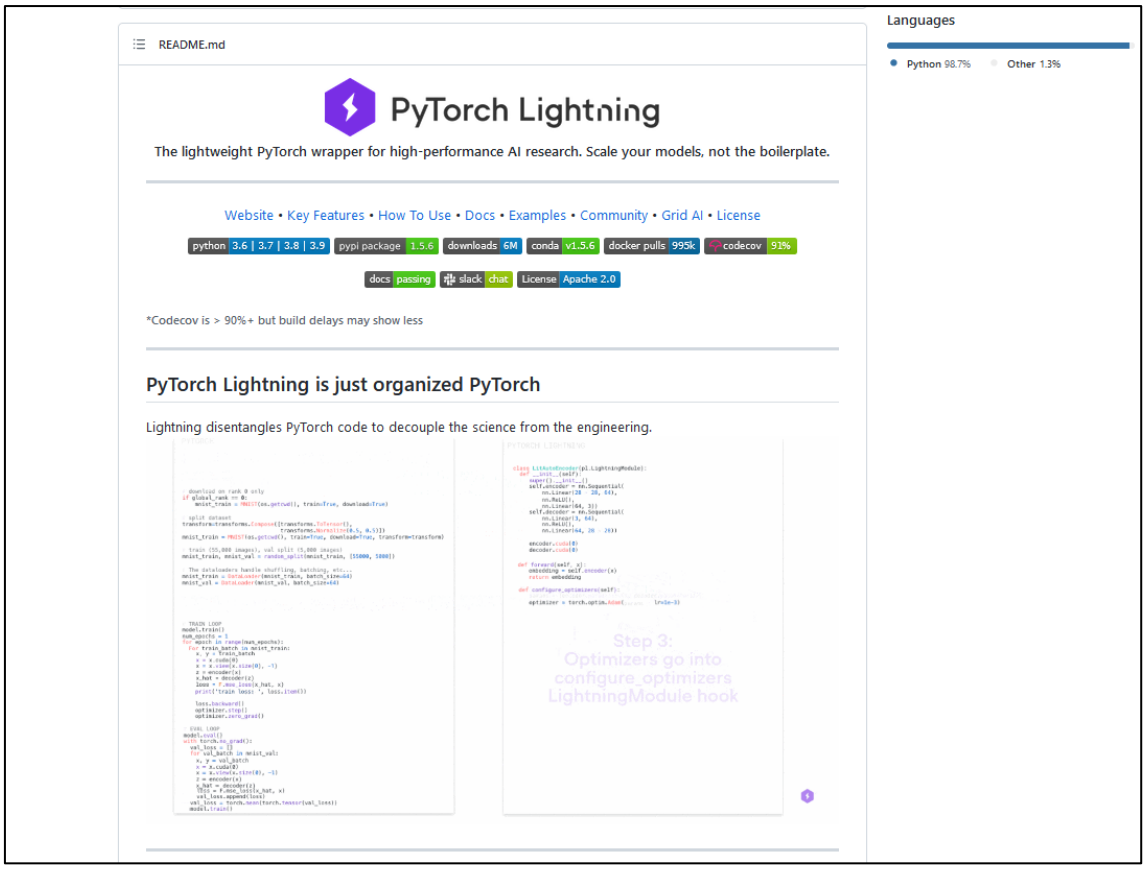
Pytorch es una biblioteca diseñada para crear proyectos de Machine Learning desarrollada por Facebook AI Research que entre otros proyectos se ha utilizado en el Autopilot de Tesla y en proyecto Pyro de Uber [23].



11. Web oficial del proyecto PyTorch
Fuente: https://pytorch.org/ Acceso: 16/12/21

Como complemento a PyTorch, se ha utilizado PyTorch Lightning [24] que es una biblioteca de código libre que ofrece un interfaz de alto nivel para PyTorch; una manera de organizar los programas escritos en PyTorch para conseguir implementaciones modulares, flexibilidad y módulos autocontenidos.

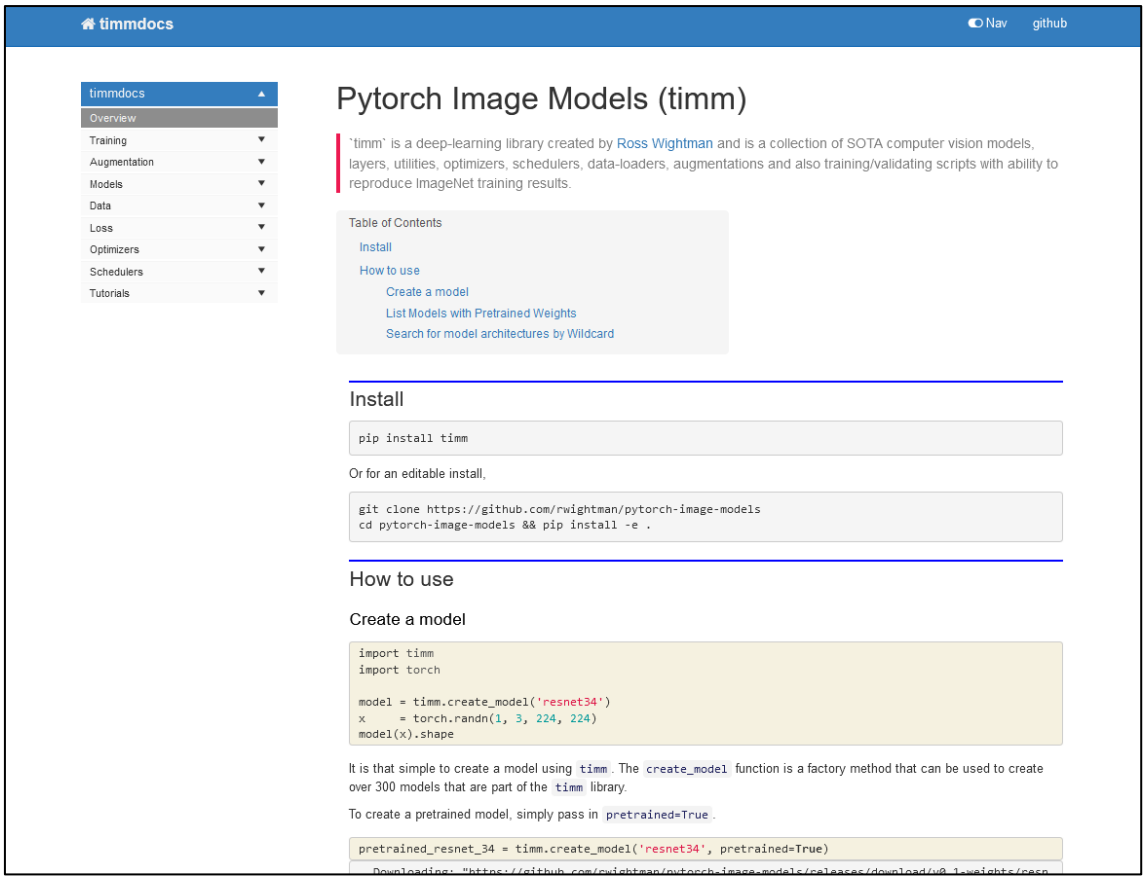
Además, ofrece funciones de alto nivel para las operaciones más habituales a la hora de entrenar una red neural con soporte GPU y TPU nativos.



12. Portada del proyecto PyTorch Lightning en Github.
Fuente: https://github.com/PyTorchLightning/pytorch-lightning Acceso: 14/12/21

Para la implementación de la red neural EfficientNet utilizada en este trabajo se ha utilizado la biblioteca Pytorch Image Models, conocida popularmente como timm [25][26].

timm es una biblioteca de aprendizaje automático creada por el conocido investigador Ross Wightman [27].



13. Página principal de la biblioteca timm
Fuente: https://fastai.github.io/timmdocs, Acceso: 14/12/21

Para el cálculo de las distintas métricas y matriz de confusión se han utilizado las bibliotecas Scikit-learn y Torchmetrics.

Scikit-learn [28], también conocida como sklearn es una de las bibliotecas de funciones más populares para aprendizaje automático.

Torchmetrics [29] es una biblioteca de métricas para de machine Learning que funciona sobre modelos creados con PyTorch.

Además de los *frameworks* descritos, se han utilizado otras bibliotecas para obtener distintas funcionalidades que se usan en este trabajo. Las principales son:

- Matplotlib
- PIL
- Numpy
- Panda
- Seaborn

3.3 Implementación

Estudio de las clases

Se comienza la implementación con un estudio del conjunto de datos que se va a utilizar para entrenar a la red.

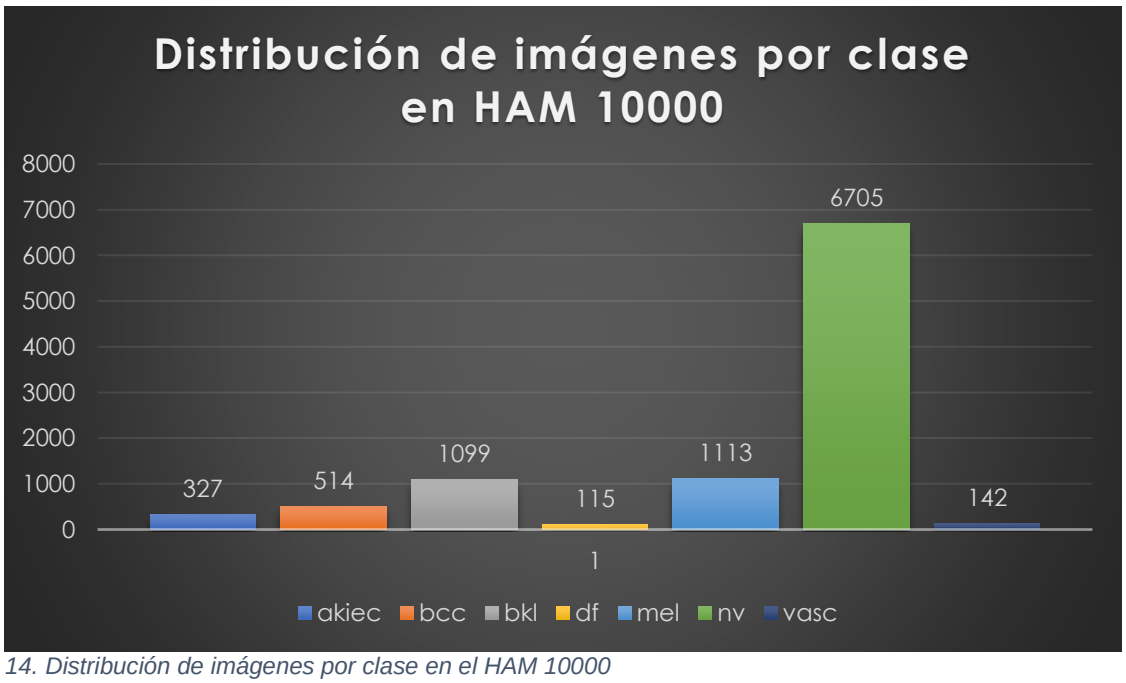
En este caso, el conjunto de datos HAM 10000 tiene 10015 imágenes distribuidas en 7 clases.

Tabla 1. Distribución en clases del HAM 10000

Clase	Cantidad	Proporción
akiec	327	3.3%
bcc	514	5.1%
bkl	1099	11.0%
df	115	1.1%
mel	1113	11.1%
nv	6705	66.9%
vasc	142	1.4%

Se puede apreciar que la distribución entre las clases está claramente desbalanceada:

La clase nv, correspondiente a los nevus melanocíticos, contiene casi el 67% de las imágenes, mientras que otras como df o vasc, tienen menos del 2% cada una.



14. Distribución de imágenes por clase en el HAM 10000
Fuente: Elaboración propia

Estudio del rango de valores de RGB

A la hora de utilizar imágenes de diferente procedencia para entrenar una red es importante que los parámetros de entrada tengan una distribución de datos similar; esto facilita el aprendizaje del sistema.

En nuestro caso, al utilizar imágenes como entrada, nos interesa que los píxeles de las imágenes estén normalizados.

Esta normalización se hará en 2 pasos, primero se calculará la media y la desviación típica de todo el conjunto de imágenes y después se utilizará esta información con la función *torchvision.transforms.Normalize()*, que restará a los valores la media de su canal y los dividirá por la desviación típica.

Se ha utilizado el siguiente script para realizar el estudio de los valores del conjunto de imágenes que debemos usar.

```
# -*- coding: utf-8 -*-
"""
Created on Sat Oct  9 19:35:21 2021

@author: abarsan
"""
import torch
from torch.utils.data import DataLoader
from torchvision import transforms, datasets

def obtain_mean_and_std(dataloader):

    channels_sum, channels_squared_sum, num_batches = 0, 0, 0

    print("Starting the calculation")

    for data, _ in dataloader:
        channels_sum += torch.mean(data, dim=[0,2,3])
        channels_squared_sum += torch.mean(data**2, dim=[0,2,3])
        num_batches += 1

    mean = channels_sum / num_batches
    std = (channels_squared_sum / num_batches - mean ** 2) ** 0.5
    return mean, std

DATASET_NAME = "224"
MAIN_DATA_DIR = 'C:/Users/Usuario/Google Drive/Master/TFM/'
TRAIN_DIR = MAIN_DATA_DIR + DATASET_NAME

transform = transforms.Compose([
    transforms.ToTensor(),
])

imageDataset = datasets.ImageFolder(TRAIN_DIR, transform = transform)
dataLoader = DataLoader(imageDataset,  batch_size=len(imageDataset), shuffle=False, num_workers=1)
obtain_mean_and_std(dataLoader)
```

Este script nos devuelve los siguientes valores:

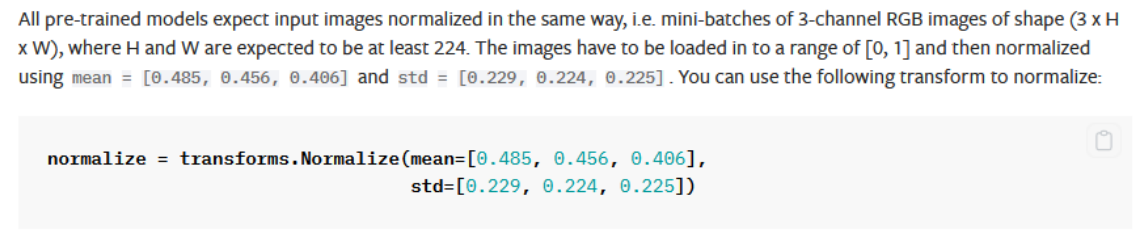
Medias en formato RGB por canal: 0.7649, 0.5422, 0.5683

Desviaciones típicas en formato RGB por canal: 0.1376, 0.1590, 0.1754

Estos son los valores que se aplicarán al cargar las imágenes en los *dataloaders*:

```
transform = transforms.Compose([
  transforms.ToTensor(),
        transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
      ])
```

Como era de esperar, estos valores difieren considerablemente de los que recomienda PyTorch para normalizar imágenes de ImageNet, ya que la naturaleza de unas y otras es muy distinta.



All pre-trained models expect input images normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), where H and W are expected to be at least 224. The images have to be loaded in to a range of [0, 1] and then normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`. You can use the following transform to normalize:

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

15. Normalización para ImageNet en PyTorch
Fuente:https://pytorch.org/vision/stable/models.html. Acceso: 24/12/21

Reducción de las imágenes

Debido a las limitaciones en el hardware, especialmente de la memoria tipo GPU disponible que tiene la plataforma utilizada en la realización de este trabajo, se optó por reducir el tamaño de las imágenes. El tamaño original de las imágenes del *dataset* HAM 10000 es de 600 x 450.

Se le ha aplicado una reducción de tamaño utilizando interpolación bilineal y después se ha cortado la sección central de la imagen (*crop*) para obtener una imagen cuadrada de 224 x 224 píxeles.

Se ha decidido aplicar la operación de corte en lugar de cambiar la relación de aspecto debido a que en imagen médica y en particular, en el campo de las lesiones de piel, estas dimensiones pueden ser importantes para el diagnóstico y no se desea introducir un cambio en la simetría de una imagen.

Se ha utilizado el valor de 224x224 píxeles por ser el tamaño para el cual fue diseñada la versión b0 de EfficientNet y porque ofrece un buen compromiso entre un tamaño de imagen que permita obtener buenos resultados y cuyo peso no sea excesivo para la configuración de hardware de la que se dispone.

Se han publicado estudios en los cuales el límite de tamaño de imagen para obtener resultados aceptables se encuentra en 128x128 píxeles, para DenseNet-121, ResNet-18 y ResNet-50 [30], por lo que 224x224 nos ha parecido un valor adecuado.

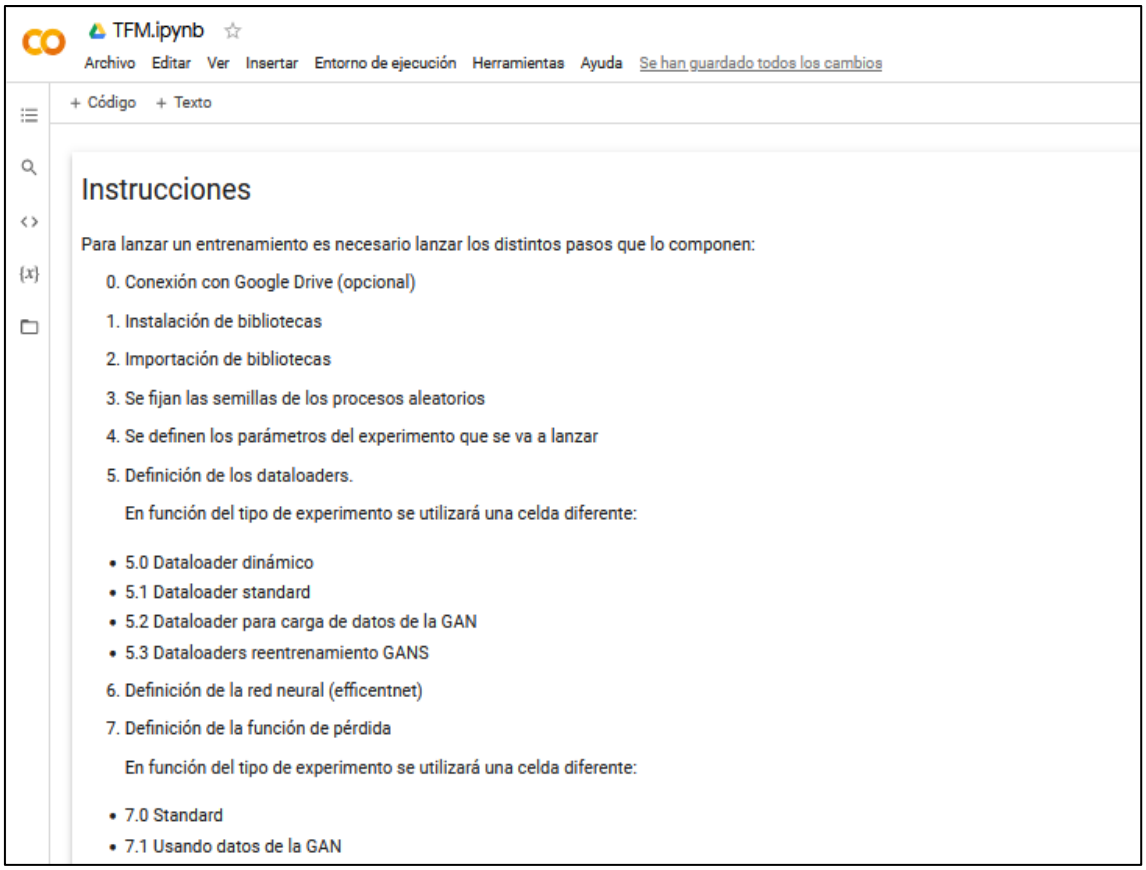
16. Ejemplo de imagen original e imagen tras la transformación
Fuente: Elaboración propia

El primer prototipo

Una vez estudiado el *dataset*, pasamos a implementar el modelo. Para ello utilizamos un único documento de Jupyter Notebook que se puede revisar en los Anexos.

Este documento contiene varios pasos, cada uno de ellos con una función específica. Algunos pasos tienen varias versiones; por ejemplo, para el paso 5, hay 4 versiones distintas, correspondientes a distintos tipos de entrenamientos utilizados durante este trabajo.

El primero punto del documento es un glosario con instrucciones sobre el contenido de cada uno de estos elementos:

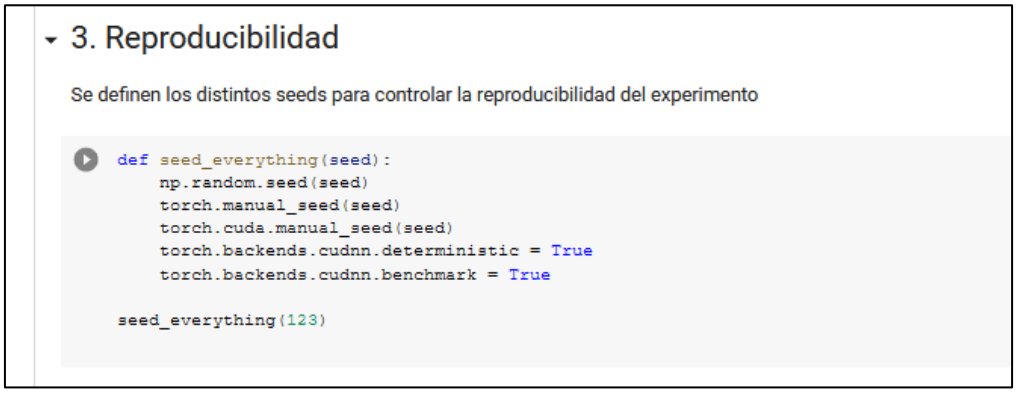


17. Instrucciones en Colab de como utilizar el documento Jupyter Notebook
Fuente: Elaboración propia

El primero paso sería gestionar la conexión con Google Drive, donde tenemos todos los datos y donde vamos a guardar los resultados de los distintos entrenamientos.

Los pasos segundo y tercero serían para instalar e importar las distintas bibliotecas utilizadas en el programa.

El paso tercero es fijar en la medida de los posible los distintos *seeds* que se utilizan en las distintas partes del programa, para controlar la reproducibilidad del experimento.

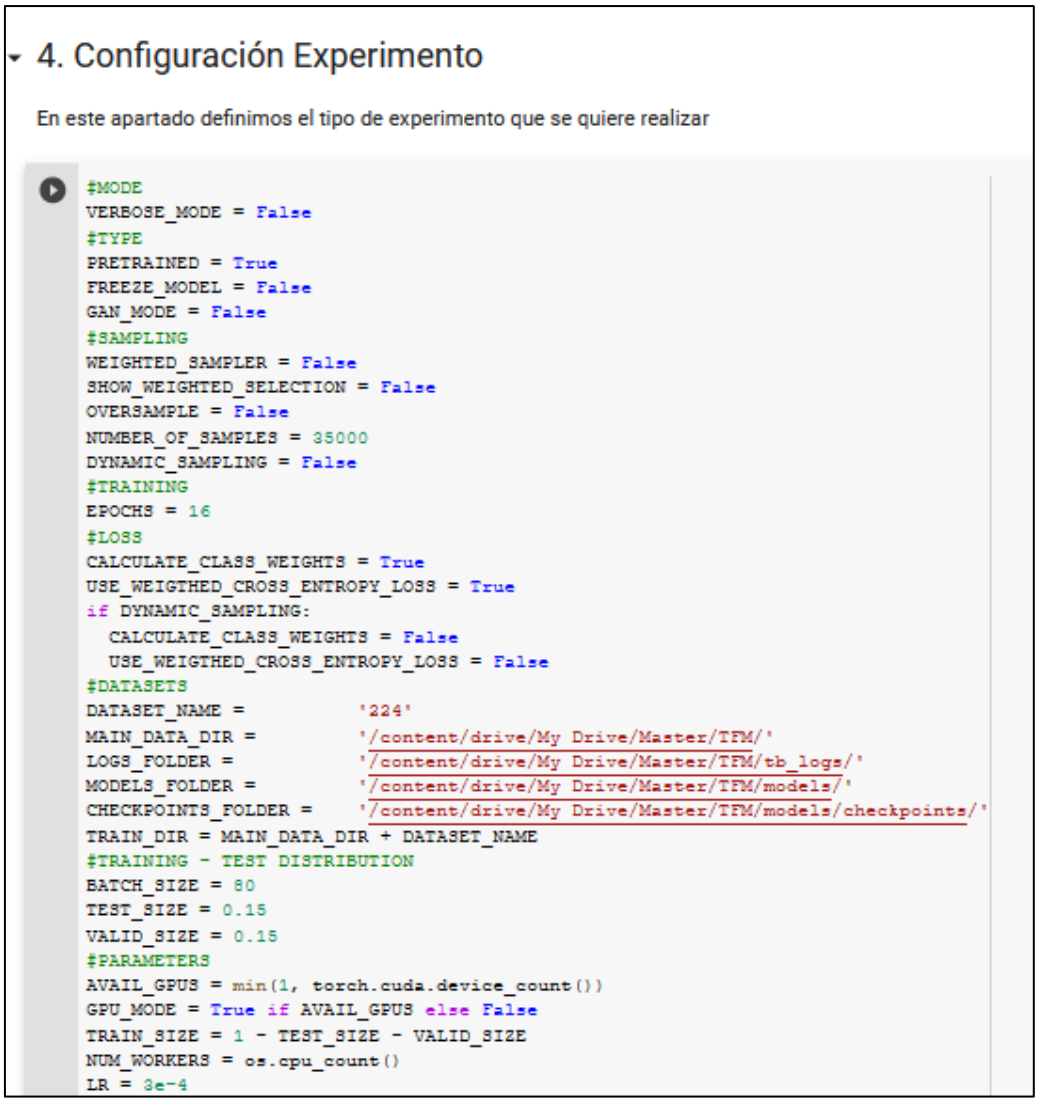


18. Detalle del paso 3
Fuente: Elaboración propia

Los valores que se han utilizado en el entrenamiento, como las proporciones de los distintos conjuntos de datos, o el tamaño de los lotes, son parámetros que se pueden modificar fácilmente en el sistema para cambiar esta proporción, pero es la que finalmente se ha utilizado.

Hay otros parámetros que controlan el tipo de *logging* que se realizará al lanzar la aplicación, la ubicación de los distintos grupos de imágenes utilizados, el número de *epochs* utilizados en el aprendizaje, hiperparámetros como el *learning rate* o modos de entrenamiento más avanzado utilizados en etapas posteriores del trabajo.

Todos ellos se gestionan en el paso 4 del documento.



19. Detalle del paso 4
Fuente: Elaboración propia

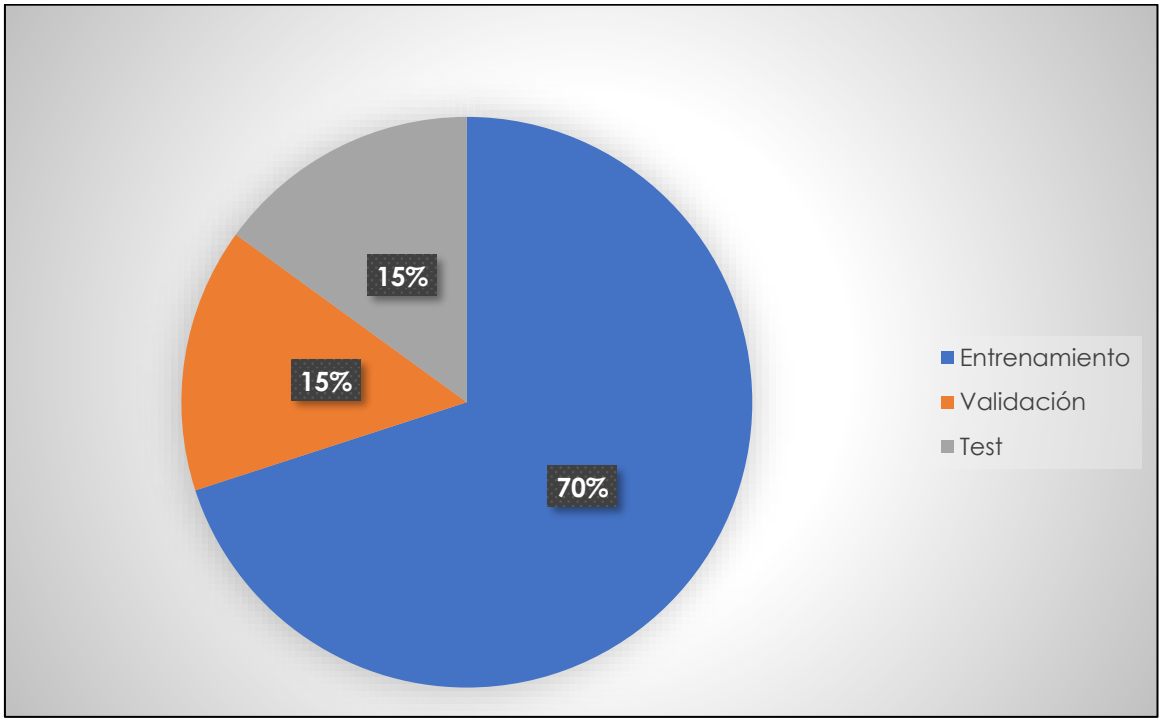
El siguiente paso es dividir el total de imágenes en tres partes:

- Conjunto de entrenamiento: Este es el grupo de datos que utilizaremos para realizar el entrenamiento de nuestro modelo. El objetivo es que la red aprenda a ver y reconocer este conjunto.

- Conjunto de validación: El conjunto de validación se utiliza para realizar evaluaciones frecuentes del modelo y ajustar sus parámetros adecuadamente para optimizar el aprendizaje.

- Conjunto de test: El conjunto de test es el grupo de imágenes que servirán como evaluación final de un modelo, después de haber realizado el entrenamiento.

Se ha utilizado una proporción de 70,15,15:



20. Distribución en conjuntos de entrenamiento, validación y test
Fuente: Elaboración propia

El *framework* de PyTorch proporciona funcionalidad muy avanzada para crear los componentes, llamados *dataloaders* que se utilizarán para cargar lotes de imágenes en el sistema de entrenamiento de la red [31]

```
train_loader = torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, drop_last= T
rue, sampler=train_sampler, pin_memory=GPU_MODE)
print("Train Loader created successfully")

valid_loader = torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, drop_last= T
rue, sampler=valid_sampler, shuffle=False, num_workers=num_workers, pin_memory=GPU_MODE)
print("Validation Loader created successfully")

test_loader = torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, drop_last= Tr
ue, sampler=test_sampler, shuffle=False, num_workers=num_workers, pin_memory=GPU_MODE)
print("Test Loader created successfully")
```

Estos *dataloaders* se nutrirán de una partición aleatoria del *dataset* completo, en la cual distribuiremos las imágenes en proporciones de 70-15-15.

```
#CREATE SPLITS
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
split1 = int(np.floor(valid_size * length))
split2 =int(np.floor((test_size + valid_size) * length))
train_idx, test_idx, valid_idx = indexes[split2:], indexes[:split1], indexes[split1:split2]

train_sampler = SubsetRandomSampler(train_idx)
test_sampler = SubsetRandomSampler(test_idx)
valid_sampler = SubsetRandomSampler(valid_idx)
```

Esta funcionalidad se codifica en el punto 5 del documento.

21. Detalle del punto 5
Fuente: Elaboración propia

En los siguientes puntos del documento se definen el modelo de red utilizando el *framework* timm, se definen las métricas que se van a obtener para determinar la calidad del entrenamiento y del modelo, y se define la rutina del entrenamiento.

22. Detalle del punto 6. Definición del modelo

El *framework* PyTorch Lightning recomienda hacer estos pasos de una forma determinada y propone definir por lado, una clase de entrenamiento tipo Lightning Module, un modelo de datos, donde se gestionan los *dataloaders* tipo LightningDataModule y finalmente entrenar el modelo utilizando una función llamada Trainer asociada. De este modo, utilizando este *framework* se abstrae la implementación de las distintas partes del entrenamiento y se puede enfocar el trabajo hacia el estudio de los resultados y la mejora del sistema.

9. Definición del Modelo de Test

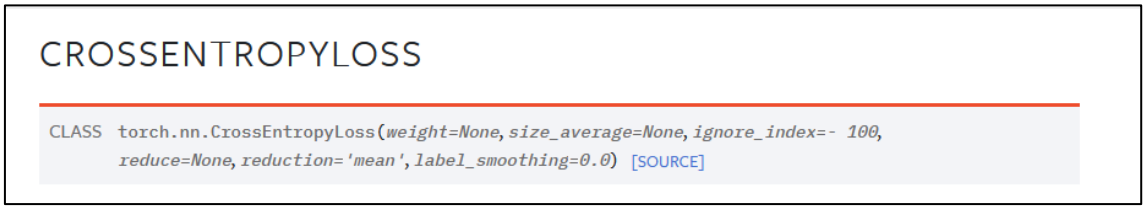
```
class TFMModel(pl.LightningModule):

    def __init__(self, lr = LR, input_size=None):
        super().__init__()
        self.save_hyperparameters()
        self.model = createModel()
        self.train_acc = metric
        self.valid_acc = metric

    def forward(self, x):
        return self.model(x)

    def training_step(self, batch, batch_idx):
        #visualize_filters(self.model)
        inputs, labels = batch
        y_pred = self(inputs)
        loss = trainLossFunction(y_pred, labels)
        self.log('train_loss', loss)
        ap = self.train_acc(y_pred, labels)
        self.log('train_acc', ap)
        return loss

    def evaluate(self, batch, stage=None):
        inputs, labels = batch
        y_pred = self(inputs)
        if stage == "val":
          loss = validationLossFunction(y_pred, labels)
        else:
          loss = testLossFunction(y_pred, labels)
        if VERBOSE_MODE:
          print("**************",stage, "********************")
          print("LABELS ", labels)
          print("Y pred ", y_pred)
          print("Loss:", loss)
        self.train_acc(y_pred, labels)
        if (stage):
          self.log(f"{stage}_loss", loss, prog_bar = True)
          self.log(f"{stage}_acc", self.train_acc, prog_bar = True)
        return {'loss': loss, 'preds': y_pred, 'target': labels}

    def validation_step(self, batch, batch_idx):
        outputs = self.evaluate(batch, "val")
```

23. Detalle de la definición del modelo de test
Fuente: Elaboración propia

Función de pérdida

Para el entrenamiento en aprendizaje profundo se utiliza un descenso de gradiente estocástico por lo que tenemos que definir una función de pérdida cuando definimos el modelo. El objetivo del entrenamiento será ajustar los pesos del sistema para minimizar esta función.

En este caso, hemos utilizado la función de pérdida de entropía cruzada, en su implementación de PyTorch:



24. Ayuda de PyTorch para la función CrossEntropyLoss
Fuente: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html Acceso: 23/12/2021

La idea de usar esta función es comparar la salida la red con la salida deseada y calcular una puntuación que simboliza lo lejos que estamos del valor deseado.

La función de pérdida por entropía cruzada está recomendada para problemas de clasificación con varias clases. Esta implementación en particular permite además pasarle a la función un tensor con el peso de cada clase en la distribución, lo que lo es especialmente útil en conjuntos de datos desbalanceados como el que nos ocupa.

Función de optimización

Como función de optimización se ha usado la estimación de momento adaptativo (ADAM), que es uno de los algoritmos más utilizados por sus buenos resultados.

La función de optimización guía al algoritmo en su descenso de gradiente, idealmente en búsqueda de un punto mínimo global de la función. Según se realice este descenso, de una forma más abrupta o menos, es más o menos probable obtener mínimos globales y soluciones óptimas a nuestra función.

Con ADAM se combinan las ideas de la estrategia del aprendizaje adaptativo implementada en Adagrad de que modula las actualizaciones del sistema en función de la frecuencia de las características a adaptar; y por otro lado de la estrategia Momentum, donde se añade el concepto del impulso, que acelera la velocidad de convergencia cuando la oscilación de la función objetivo se ralentiza. [32]

El optimizador ADAM se define en el método *configure_optimizers* de la clase modelo de PyTorch Lightning.

```
def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.hparams['lr'])
        return optimizer
```

Entrenamiento

Finalmente definimos un objeto Trainer de PyTorch Lightning y se le pasan los módulos creados anteriormente llamando a la función de entrenamiento.

El *trainer* permite muchas configuraciones distintas; desde añadir un *logger* que recopile la información del entrenamiento, seleccionar si el entrenamiento se realiza en GPU o CPU, el número máximo de iteraciones o incluso si se retoman entrenamientos anteriores.

12. Entrenamiento

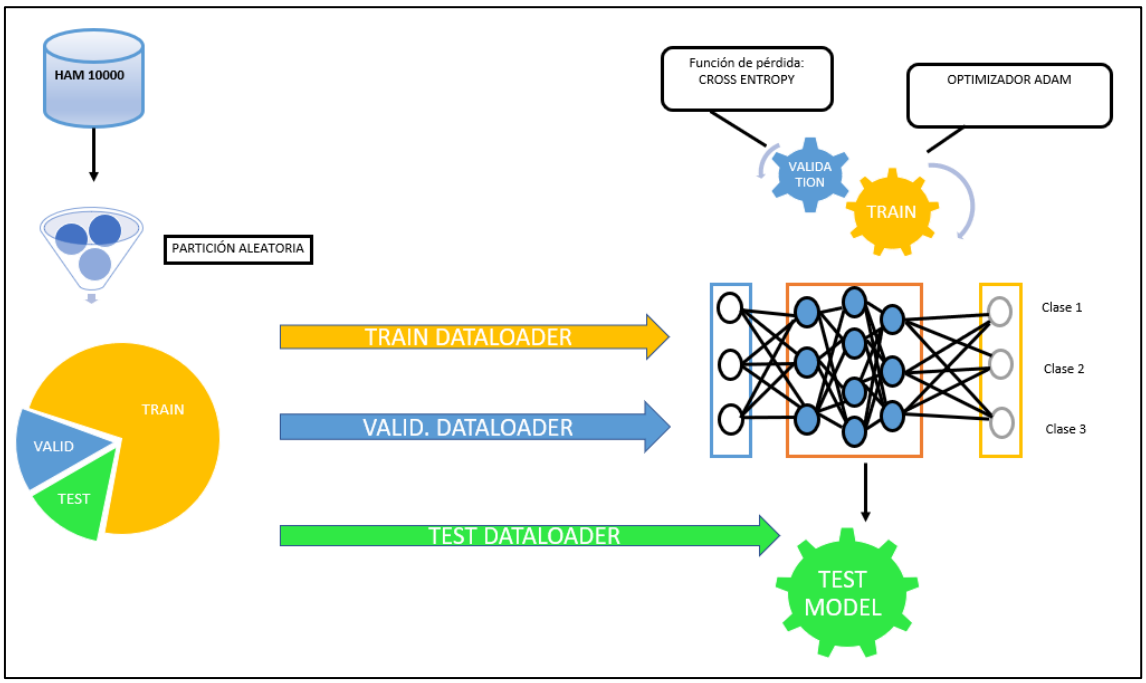
```
from time import strftime

TIME = strftime("%Y-%m-%d %H:%M:%S")
name = NAME_OF_DATASET + "-" + TIME
EPOCHS = 10
print("Training dataset ", name)
print("EPOCHS: " , EPOCHS)
print("LR: ", LR)
tb_logger = TensorBoardLogger(LOGS_FOLDER, name = name)
csv_logger = CSVLogger(LOGS_FOLDER, name = name)
gc.collect()
torch.cuda.empty_cache()
RESUME = False

#CHECKPOINT = '/content/drive/MyDrive/Master/TFM/models/checkpoints/224_80_0.0003(70:15:15)_TR-GAN-.ckpt'

if GPU_MODE:
  if RESUME:
      trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      precision = 16,
                      logger=[tb_logger, csv_logger],
                      gpus=-1,
                      max_epochs=EPOCHS,
                      log_every_n_steps=10,
                      deterministic=True,
                      reload_dataloaders_every_n_epochs=2,
                      default_root_dir=CHECKPOINTS_FOLDER,
                      resume_from_checkpoint = CHECKPOINT)
  else:
    trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      precision = 16,
                      logger=[tb_logger, csv_logger],
                      gpus=-1,
                      max_epochs=EPOCHS,
                      log_every_n_steps=50,
                      reload_dataloaders_every_n_epochs=2,
                      deterministic=True,
                      default_root_dir=CHECKPOINTS_FOLDER)
```

25. Detalle del paso de entrenamiento del documento de Colab
Fuente: Elaboración propia

Resumen

El *framework* completo de entrenamiento sería por lo tanto un documento tipo Jupyter Notebook corriendo en Google Colab, que obtiene los datos y guarda los resultados en una cuenta de Google Drive.

El esquema de funcionamiento de este sistema sería el siguiente:



26. Esquema del modelo
Fuente: Elaboración propia

3.4 Métricas

Como ya hemos visto el conjunto de datos HAM 10000, tiene un gran desbalance entre el número de imágenes que presenta en cada clase.

La medida de Exactitud (Accuracy) nos da la medida de los casos identificados correctamente. Se suele utilizar cuando las medidas de todas las clases son de la misma importancia. Sin embargo, esta métrica no ofrece mucha información sobre los casos clasificados incorrectamente, y en el caso de diagnóstico médico esto es algo a tener en cuenta, ya que un falso negativo en una clase crítica como el melanoma puede ser crítico.

Por lo tanto, aunque la exactitud se calculará en todos los casos, no lo tendremos en cuenta como nuestro estimador principal.

En su lugar vamos a utilizar la métrica denominada f1-score, que nos permite revisar en un solo indicador la precisión y la exhaustividad.

La precisión en la ratio que tiene en cuenta los verdaderos positivos y los falsos positivos en la forma de:

$$\frac{VP}{VP + FP}$$

La exhaustividad o *recall* se calcula utilizando los verdaderos positivos y los falsos negativos utilizando la fórmula:

$$\frac{VP}{VP + FN}$$

La precisión nos va a dar la medida de la calidad del modelo clasificador mientras que la exhaustividad nos da información sobre la cantidad de elementos que va a ser capaz de identificar.

El valor f1 o f1-score, se calcula como la media armónica de ambos.

$$2 * \frac{(precisión * exhaustividad)}{precisión + exhaustividad}$$

El resultado que se utilizará como métrica principal va a ser el promedio de la puntuación de este f1-score en cada categoría. También denominado f1-score macro.

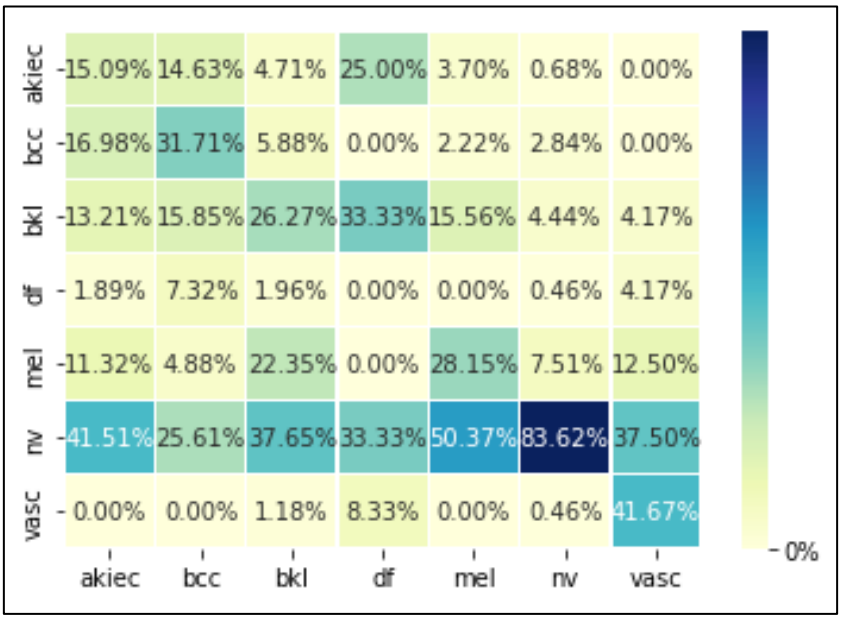
Los resultados los completaremos con una matriz de confusión para detectar cómo se comporta cada una de las clases independientemente.

Una matriz de confusión es una herramienta que se utiliza en Machine Learning para revisar la calidad del entrenamiento de los modelos ya que permite

visualizar muy fácilmente si el sistema está asignando predicciones de una clase a otra, lo que nos da una información granular sobre cada clase.

Cada columna de la matriz representa las predicciones de la red y las filas representa las instancias de la clase real.

Por ejemplo, en el caso del HAM 10000 tendríamos una matriz 7x7 como esta:



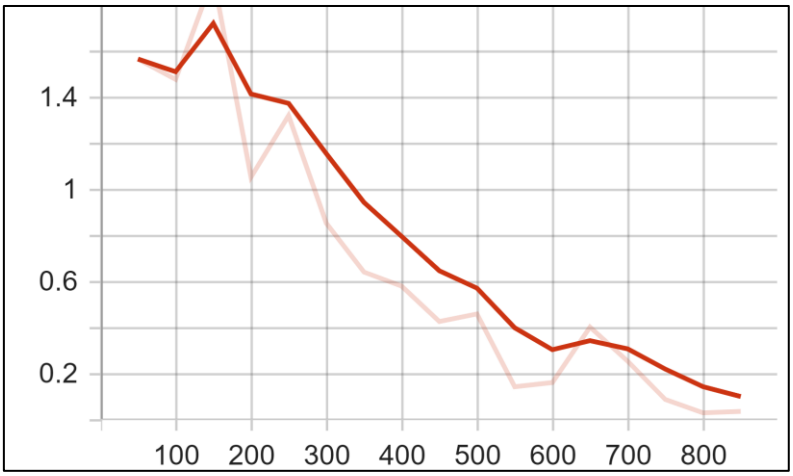
27. Ejemplo matriz de confusión
Fuente Elaboración propia

3.5 Resultados

La línea base

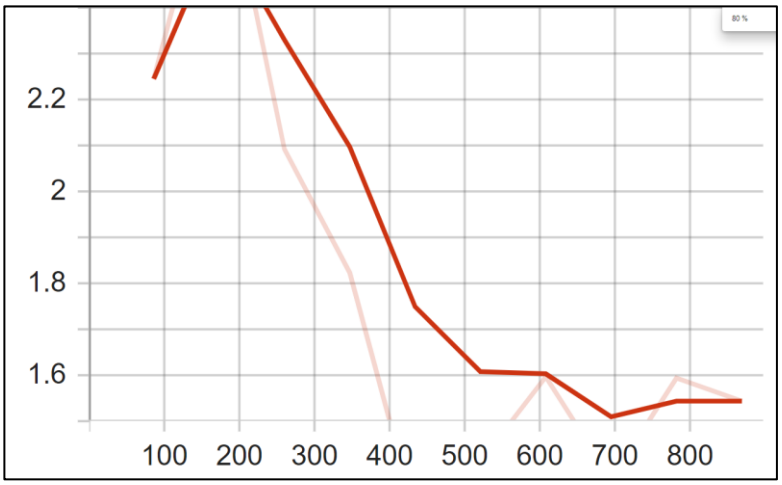
El primer modelo utiliza una red sin entrenamiento previo, con un entrenamiento a 10 *epochs*, un tamaño de lotes de 80 y una ratio de aprendizaje de 3e-4.

En la etapa de aprendizaje monitorizamos las funciones de pérdida del entrenamiento y de la validación.



28. Funciones de pérdida durante el entrenamiento de la línea base
Fuente: Elaboración propia

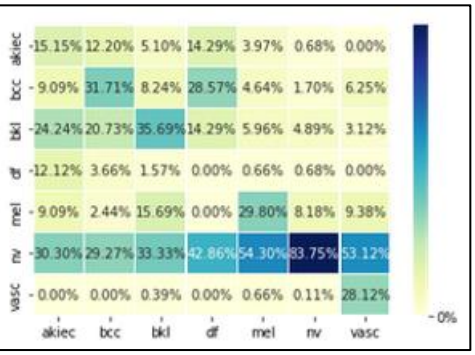
La función de pérdida indica una bajada continua tanto en el control del conjunto de entrenamiento como en el de validación, de lo cual podemos deducir que la red está aprendiendo.



29. Función de pérdida durante la validación de la linea base
Fuente: Elaboración propia

Después del entrenamiento, se prueba el modelo entrenado con conjunto de test.

	precision	recall	f1-score	support
akiec	0.15	0.12	0.14	41
bcc	0.32	0.34	0.33	76
bkl	0.36	0.54	0.43	170
df	0.00	0.00	0.00	18
mel	0.30	0.27	0.28	165
nv	0.84	0.77	0.80	958
vasc	0.28	0.75	0.41	12
accuracy			0.63	1440
macro avg	0.32	0.40	0.34	1440
weighted avg	0.66	0.63	0.64	1440



30. Resultado baseline
Fuente: Elaboración propia

El resultado con este modelo es de un f1-score macro de **0.34.**

Vemos en este ejemplo que la accuracy es de 0.63, pero este valor está aportado totalmente por los aciertos en la clase mayoritaria, por lo que no nos parecen relevantes.

```
Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben
ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

Con la siguiente configuración de variables:

#MODE
VERBOSE_MODE = False

#TYPE
PRETRAINED = False
FREEZE_MODEL = False
GAN_MODE = False

#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False

#TRAINING
EPOCHS = 10
#LOSS
CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

3.6 Conclusiones acerca de la línea base

Los resultados no son demasiado buenos después del primer entrenamiento. El resultado de 0.34 se antoja bajo. Se puede apreciar, sin embargo, que, en la clase con más datos, la puntuación es de 0.80 y que la nota global se ve perjudicado gravemente por el pobre desempeño de la red en las clases con menos muestras.

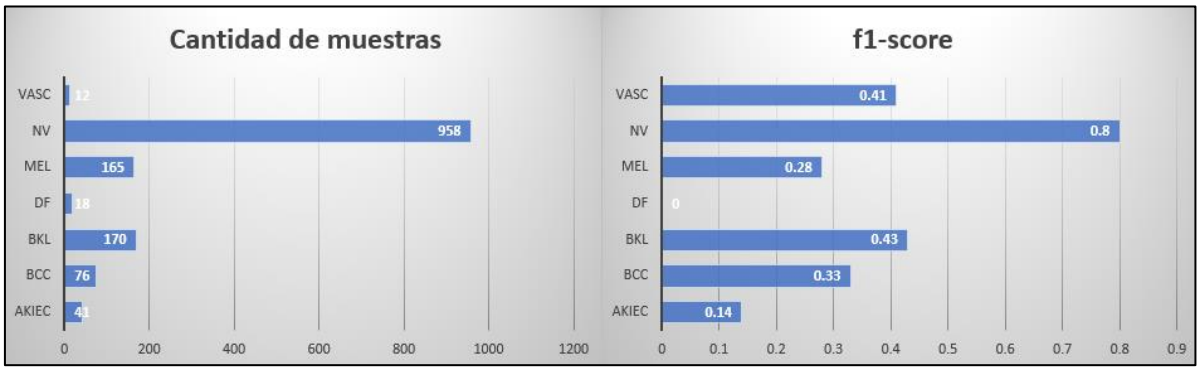
Se observa también que, en las últimas etapas del entrenamiento, la función de pérdida del conjunto de validación comienza a subir, lo cual nos indica que puede estar empezando un proceso de *overfitting*.

La conclusión principal es que el desbalance tan acusado de las distintas clases del conjunto, no permite un entrenamiento en el que el modelo sea capaz de generalizar su conocimiento.

Para algunas clases, las que tienen menos elementos, la red no es capaz de adaptar sus pesos para identificarlas, sin embargo, las clases mayoritarias y en particular, la clase NV, que tiene el 66% de las imágenes, consigue unos valores muy buenos.

Se puede apreciar también que la red confunde mucho unas clases con otras, especialmente con la clase mayoritaria.

En el siguiente gráfico se puede apreciar una cierta correlación entre el número de imágenes y los resultados obtenidos.

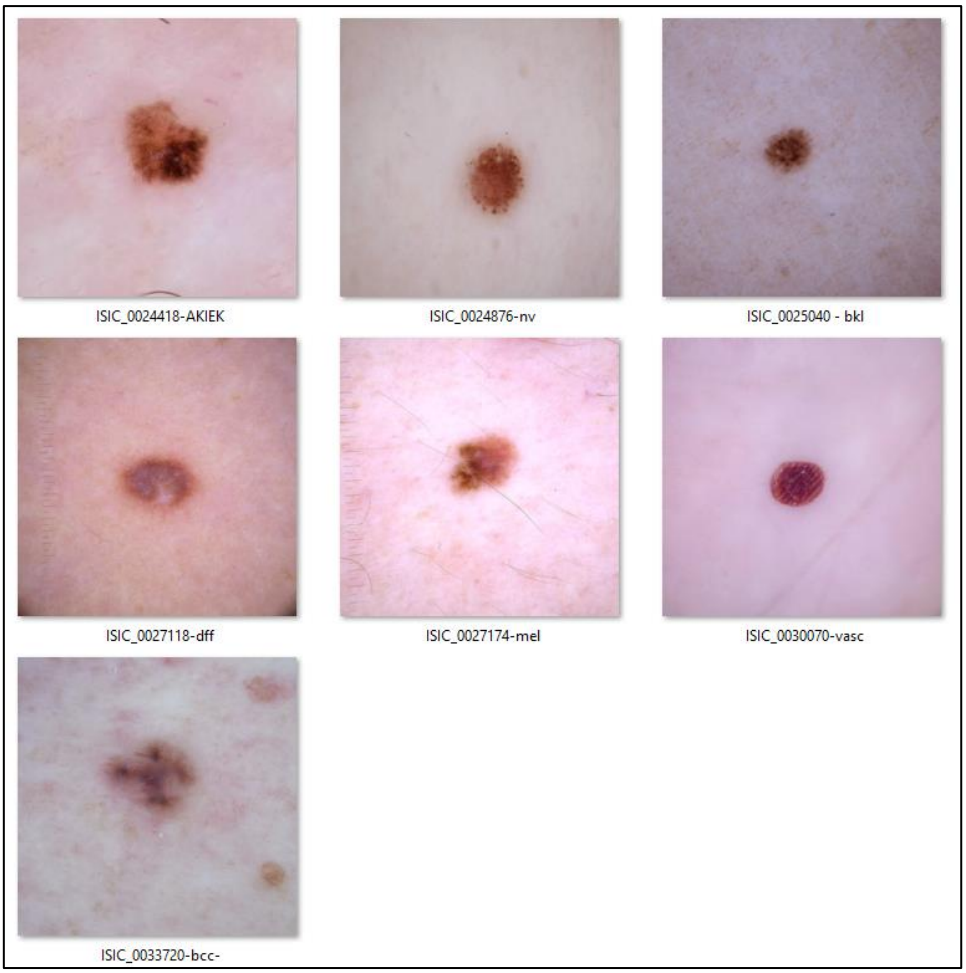


31. Resultados vs Cantidad de muestras por clase
Fuente: Elaboración propia

El resultado de la clase nv, con un gran número de muestras es bastante notable y comparable al de un experto humano.

En una comparativa realizad por Haenssle et al.[33] 58 dermatólogos compitieron contra una red tipo Inception v4 [34] y el resultado es que los especialistas obtuvieron un 86.6% en sensibilidad (tasa de verdaderos positivos) y un 71.3% de especificidad (tasa de verdaderos negativos) en la detección [35].

En la imagen 31 se puede observar a un representante de cada una de las 7 clases. Como se puede apreciar, existe un gran parecido entre ellas, de ahí la dificultad de los expertos y de las máquinas para determinar con precisión un diagnóstico acertado.



32. Detalle de una imagen de cada clase
Fuente: Elaboración propia

Se finaliza con esta reflexión la primera fase del trabajo y se comienza la segunda, en donde se buscará mejorar el desempeño de la red, y en particular de las clases que tienen pocas imágenes y que están lastrando el resultado global del sistema.

4. Técnicas de mejora

4.1 Función de pérdida ponderada

Para intentar remediar el desbalanceo en el conjunto de entrenamiento, en la función de pérdida se implementará una mejora que tendrá en cuenta los pesos de las distintas clases a la hora de ajustar los valores de la red [36].

En el módulo de cálculo de la función de pérdida, se chequea el valor de una variable global, definida en el módulo de configuración del experimento, y si está activa, se procede primero a calcular el peso de cada clase en los distintos *datasets* y posteriormente se define la función de pérdida utilizando la entropía cruzada pasando los pesos. Los pesos que se le pasan al algoritmo se normalizan primero, utilizando el valor medio:

```
if USE_WEIGTHED_CROSS_ENTROPY_LOSS:

    classes = [trainImageDataset.targets[i] for i in train_sampler.indices]
    class_tensor = torch.FloatTensor(classes)

    validation_classes = [validImageDataset.targets[i] for i in valid_sampler.indices]
    validation_class_tensor = torch.FloatTensor(validation_classes)

    test_classes = [testImageDataset.targets[i] for i in test_sampler.indices]
    test_class_tensor = torch.FloatTensor(test_classes)
```

```
class_weights = sklearn.utils.class_weight.compute_class_weight('balanced',classes =np.unique
(class_tensor),y = class_tensor.numpy())
class_weights=torch.tensor(class_weights,dtype=torch.float)
class_weights = normalizeWeights(class_weights)

validation_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',classes =
 np.unique(validation_class_tensor),y = validation_class_tensor.numpy())
validation_class_weights=torch.tensor(validation_class_weights,dtype=torch.float)
validation_class_weights = normalizeWeights(validation_class_weights)

test_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',classes =np.uni
que(test_class_tensor),y = test_class_tensor.numpy())
test_class_weights=torch.tensor(test_class_weights,dtype=torch.float)
test_class_weights = normalizeWeights(test_class_weights)
```

Antes de enviar los pesos a la función se normalizan utilizando el valor medio:

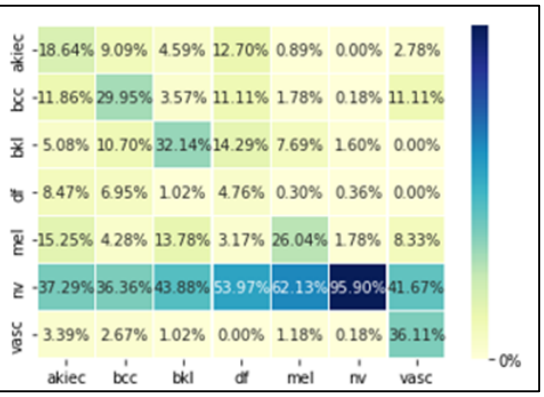
```
def normalizeWeights(inputTensor):
  mean = torch.mean(inputTensor)
  normalization_factor = 1/mean
  output = inputTensor * normalization_factor
  return output
```

Se utiliza una red sin entrenamiento previo, con un entrenamiento a 10 *epochs*, un tamaño de lotes de 80 y una ratio de aprendizaje de 3e-4.

Se añade como técnica de mejora la función de pérdida ponderada.

El resultado en un f1-score de 0.36, que mejora ligeramente el obtenido solamente con el *baseline*.

	precision	recall	f1-score	support
akiec	0.19	0.22	0.20	49
bcc	0.30	0.64	0.41	88
bkl	0.32	0.48	0.39	130
df	0.05	0.12	0.07	26
mel	0.26	0.60	0.36	147
nv	0.96	0.55	0.70	973
vasc	0.36	0.48	0.41	27
accuracy			0.54	1440
macro avg	0.35	0.44	0.36	1440
weighted avg	0.74	0.54	0.59	1440



33. Resultado utilizando Loss ponderada
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

Con la siguiente configuración de variables:

```
#MODE
VERBOSE_MODE = False

#TYPE
PRETRAINED = False
FREEZE_MODEL = False
GAN_MODE = False

#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False

#TRAINING
EPOCHS = 10
#LOSS
CALCULATE_CLASS_WEIGHTS = True
USE_WEIGTHED_CROSS_ENTROPY_LOSS = True
```

4.2 Sobremuestreo y submuestreo

En el siguiente experimento se ha realizado sobremuestreo y submuestreo del conjunto de entrenamiento. Para ello se ha aumentado el número de muestras de las clases menos representadas, (sobremuestreo u *oversampling*) y se ha disminuido el número de muestras de la clase más representada (sub muestreo o *undersampling*) de forma que durante el entrenamiento el peso de todas las clases será muy parecido.

Por lo tanto, se escogen de forma aleatoria elementos de todas las clases, pero se permite emplear el mismo elemento varias veces, de forma que, para una clase con pocos elementos, éstos se utilizarán varias veces en el entrenamiento.

El número de imágenes para cada clase en un entrenamiento para el modelo básico sería:

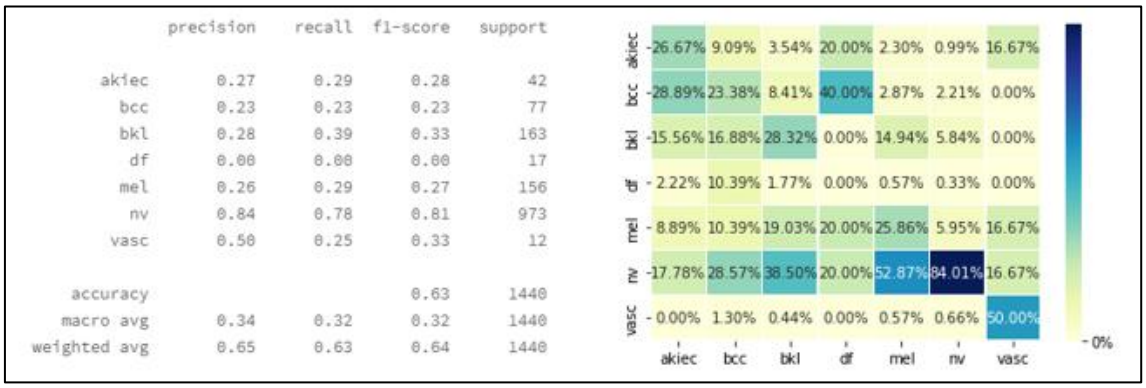
0: 232, 1: 418, 2: 771, 3: 75, 4: 768, 5: 4715, 6: 108

Como se puede apreciar, la clase 5 está muy sobrerrepresentada y la 3 muy poco representada.

Utilizando esta técnica se pretende conseguir un número parecido de elementos en todas las clases. La nueva distribución después de aplicar la corrección es:

0: 976, 1: 1054, 2: 995, 3: 958, 4: 1009, 5: 978, 6: 990

El resultado obtenido tras 10 *epochs* es un f1-score de 0.32, lo que empeora el modelo base.



	precision	recall	f1-score	support
akiec	0.27	0.29	0.28	42
bcc	0.23	0.23	0.23	77
bkl	0.28	0.39	0.33	163
df	0.00	0.00	0.00	17
mel	0.26	0.29	0.27	156
nv	0.84	0.78	0.81	973
vasc	0.50	0.25	0.33	12
accuracy			0.63	1440
macro avg	0.34	0.32	0.32	1440
weighted avg	0.65	0.63	0.64	1440

34. Resultado con oversampling y undesampling
Fuente: Elaboración propia

A continuación, se explica el código asociado a esta funcionalidad.

Primero se marcan las opciones en el apartado de configuración:

```
#SAMPLING
WEIGHTED_SAMPLER = True
SHOW_WEIGHTED_SELECTION = True
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000

CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

Esto activa el siguiente código, que utiliza un *sampler* específico que utiliza el componente **WeightedRandomSampler** de PyTorch y que realiza la función de ponderar el número de elementos que se toman por clase para el entrenamiento.

```
def createWeightedSampler():

  length = len(imageDataset)
  indexes = list(range(length))
  classes = [imageDataset.targets[i] for i in train_idx]
  class_sample_count = Counter(classes)
  print(class_sample_count)
  class_count = np.array([class_sample_count[0],class_sample_count[1], class_sample_count[2]
,class_sample_count[3],class_sample_count[4],class_sample_count[5],class_sample_count[6]])
  weight = 1. / class_count
  samples_weight = np.array([weight[t] for t in classes])
  samples_weight=torch.from_numpy(samples_weight)
  print("samples_weight", samples_weight)
  samples_weight_length = len(samples_weight)

[…]
 weighted_train_sampler = torch.utils.data.sampler.WeightedRandomSampler(samples_weight, sam
ples_weight_length, replacement=True)
  return weighted_train_sampler

testDataset = torch.utils.data.Subset(imageDataset, train_idx)
train_loader = torch.utils.data.DataLoader(testDataset, batch_size=batch_size, drop_last= Tr
ue, sampler=createWeightedSampler(), pin_memory=GPU_MODE)
```

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar la configuración ya comentada y los pasos:

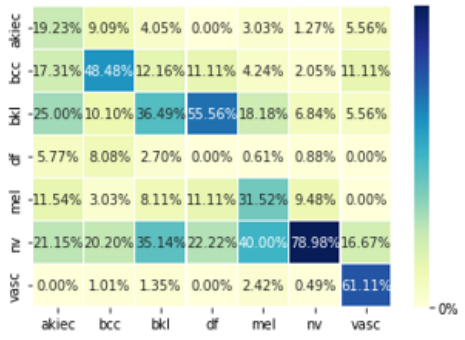
0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

4.3 Sobremuestreo

Ya que el uso del submuestreo y el sobremuestreo a la vez no ha funcionado, probamos únicamente con el sobremuestreo y aumentaremos la entrada hasta 35000 elementos, para usar aproximadamente para todas las clases, el mismo número de elementos que los que tiene la clase mayoritaria.

El resultado es un f1-score de 0.38, ligeramente mejor que el modelo inicial.

	precision	recall	f1-score	support
akiec	0.19	0.24	0.22	41
bcc	0.48	0.49	0.49	97
bkl	0.36	0.17	0.23	156
df	0.00	0.00	0.00	23
mel	0.32	0.32	0.32	165
nv	0.79	0.86	0.82	936
vasc	0.61	0.50	0.55	22
accuracy			0.66	1440
macro avg	0.39	0.37	0.38	1440
weighted avg	0.64	0.66	0.65	1440



35. Resultado con oversampling y undesampling
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

Con la siguiente configuración de variables:

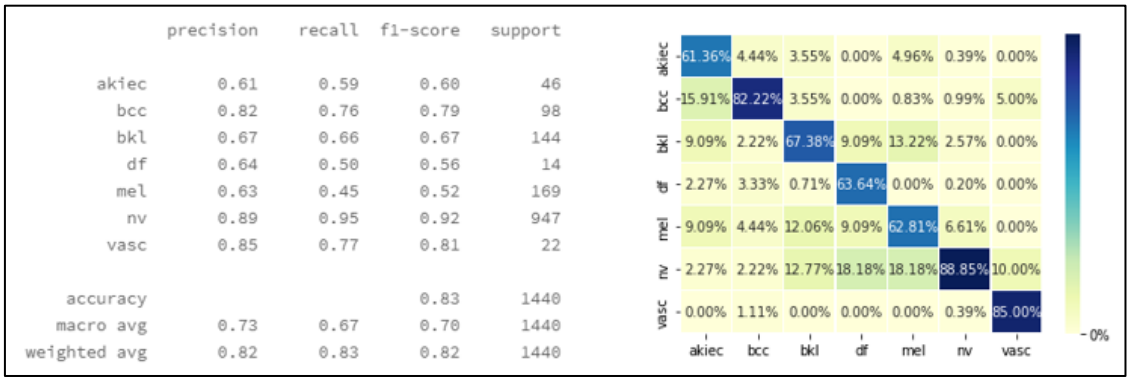
```
#SAMPLING
WEIGHTED_SAMPLER = True
SHOW_WEIGHTED_SELECTION = True
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000

CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

4.4 Aprendizaje transferido

En el intento de subsanar la carencia de un conjunto de entrenamiento extenso y balanceado, añadimos la técnica de aprendizaje transferido *o transferred learning* al paquete de mejoras del entrenamiento de la red.

Esta técnica consiste en utilizar los pesos de la misma red entrenada en otro conjunto de entrenamiento.

En particular se van a utilizar los pesos de la red previamente entrenada con ImageNet [11].



	precision	recall	f1-score	support
akiec	0.61	0.59	0.60	46
bcc	0.82	0.76	0.79	98
bkl	0.67	0.66	0.67	144
df	0.64	0.50	0.56	14
mel	0.63	0.45	0.52	169
nv	0.89	0.95	0.92	947
vasc	0.85	0.77	0.81	22
accuracy			0.83	1440
macro avg	0.73	0.67	0.70	1440
weighted avg	0.82	0.83	0.82	1440

36. Resultado con aprendizaje transferido
Fuente: Elaboración propia

El resultado presenta una gran mejora y se llega a la barrera de **0.7** puntos de f1-score. Ya se puede apreciar claramente en la diagonal de la matriz de confusión como la red está acertando mayoritariamente en todas las clases.

Para activar el aprendizaje transferido en el código, se utiliza una opción de la biblioteca timm:

```
model = timm.create_model('tf_efficientnet_b0', pretrained = PRETRAINED, num_classes=7)
```

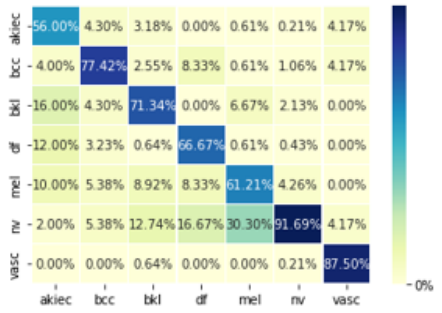
```
Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

Con la siguiente configuración de variables:

#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = False
#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False
CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

4.5. Aprendizaje transferido y oversampling

Al modelo en el que utilizamos *transferred learning*, le aplicamos ahora las técnicas que ya habíamos usado para intentar corregir el desbalance del conjunto de datos. En este experimento en particular, hemos utilizado otra vez *oversampling* a 35.000 imágenes.

El resultado en un f1-score de 0.71.

	precision	recall	f1-score	support
akiec	0.56	0.68	0.62	41
bcc	0.77	0.79	0.78	91
bkl	0.71	0.72	0.72	155
df	0.67	0.35	0.46	23
mel	0.61	0.61	0.61	166
nv	0.92	0.92	0.92	940
vasc	0.88	0.88	0.88	24
accuracy			0.84	1440
macro avg	0.73	0.71	0.71	1440
weighted avg	0.84	0.84	0.83	1440



37. Aprendizaje transferido y oversampling
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

Con la siguiente configuración de variables:

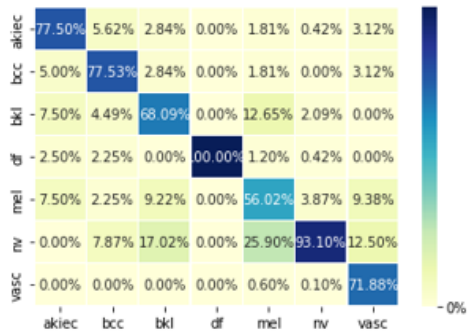
```
#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = False
#SAMPLING
WEIGHTED_SAMPLER = True
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = True
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False
CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

4.6 Aprendizaje transferido y función de pérdida ponderada

Se añade la función de pérdida ponderada ya descrita en el punto 4.1 al aprendizaje transferido.

El resultado es el siguiente:

	precision	recall	f1-score	support
akiec	0.78	0.65	0.70	48
bcc	0.78	0.87	0.82	79
bkl	0.68	0.67	0.67	144
df	1.00	0.62	0.77	24
mel	0.56	0.62	0.59	151
nv	0.93	0.92	0.93	969
vasc	0.72	0.92	0.81	25
accuracy			0.85	1440
macro avg	0.78	0.75	0.76	1440
weighted avg	0.85	0.85	0.85	1440



38. Aprendizaje transferido y función de pérdida ponderada
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

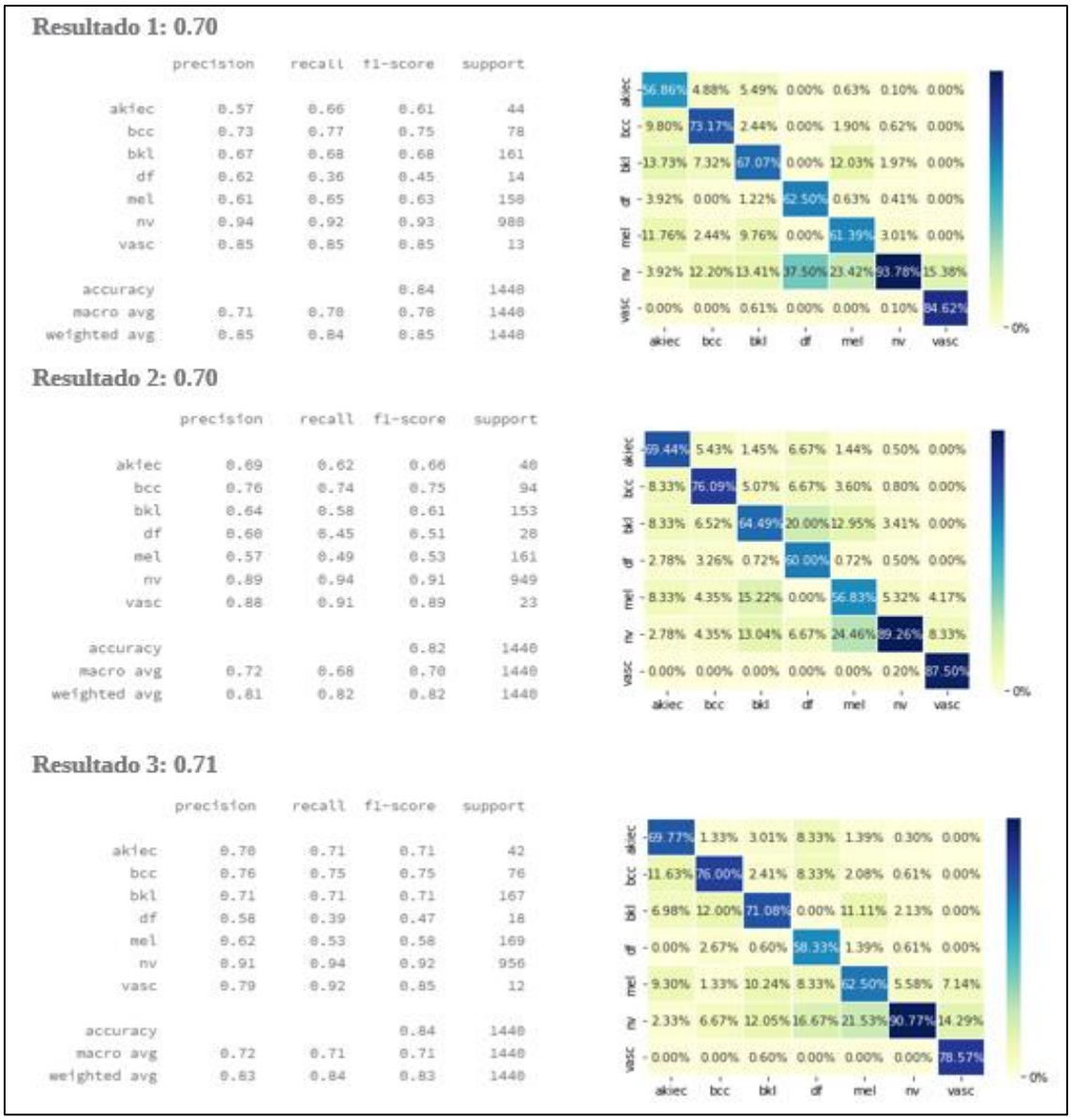
0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

Con la siguiente configuración de variables:

```
#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = False
#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False

CALCULATE_CLASS_WEIGHTS = True
USE_WEIGTHED_CROSS_ENTROPY_LOSS = True
```

En este caso se obtiene en el primer experimento un valor bastante alto, para contrastar su consistencia y mitigar un posible error, se ha repetido en experimento en otras 5 ocasiones con los siguientes resultados.

Aunque estamos fijando la semilla para fijar en la medida de lo posible la reproducibilidad de los experimentos, en cada sesión de Google Colab, el hardware utilizado es diferente, por lo que se introduce una variabilidad que explica estos ligeros cambios en los resultados.



Resultado 1: 0.70

	precision	recall	f1-score	support
akiec	0.57	0.66	0.61	44
bcc	0.73	0.77	0.75	78
bkl	0.67	0.68	0.68	161
df	0.62	0.36	0.45	14
mel	0.61	0.65	0.63	158
nv	0.94	0.92	0.93	988
vasc	0.85	0.85	0.85	13
accuracy			0.84	1448
macro avg	0.71	0.70	0.70	1448
weighted avg	0.85	0.84	0.85	1448

Resultado 2: 0.70

	precision	recall	f1-score	support
akiec	0.69	0.62	0.66	48
bcc	0.76	0.74	0.75	94
bkl	0.64	0.58	0.61	153
df	0.60	0.45	0.51	28
mel	0.57	0.49	0.53	161
nv	0.89	0.94	0.91	949
vasc	0.88	0.91	0.89	23
accuracy			0.82	1448
macro avg	0.72	0.68	0.70	1448
weighted avg	0.81	0.82	0.82	1448

Resultado 3: 0.71

	precision	recall	f1-score	support
akiec	0.70	0.71	0.71	42
bcc	0.76	0.75	0.75	76
bkl	0.71	0.71	0.71	167
df	0.58	0.39	0.47	18
mel	0.62	0.53	0.58	169
nv	0.91	0.94	0.92	956
vasc	0.79	0.92	0.85	12
accuracy			0.84	1440
macro avg	0.72	0.71	0.71	1440
weighted avg	0.83	0.84	0.83	1440

39. Repetición aprendizaje transferido y función de pérdida ponderada I
Fuente: Elaboración propia

40. Repetición aprendizaje transferido y función de pérdida ponderada II
Fuente: Elaboración propia

A la vista de los 5 entrenamientos repetidos, el resultado de 0.76 es muy extremo y lo más razonable es considerarlo un *outlier* y no tenerlo en cuenta para el resultado.

Tomaremos como resultado final de este experimento la media de los 5 experimentos repetidos:

$$\frac{0.70 + 0.70 + 0.71 + 0.71 + 0.71}{5} = 0.71$$

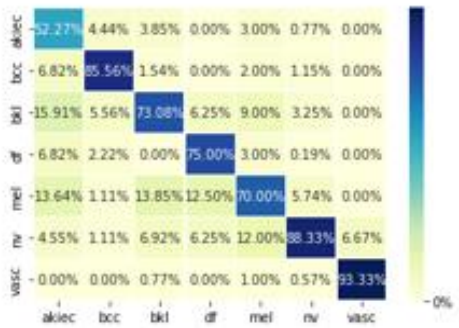
Resultado final: 0.71

4.7 Aprendizaje transferido, oversampling y pérdida ponderada

En este experimento además del *transferred learning* y el uso de una función de pérdida ponderada, hemos utilizado el *oversampling* a 35000 imágenes.

El resultado es de un f1-score de 0.70

	precision	recall	f1-score	support
akiec	0.52	0.53	0.53	43
bcc	0.86	0.80	0.83	96
bkl	0.73	0.63	0.68	151
df	0.75	0.55	0.63	22
mel	0.70	0.45	0.54	157
nv	0.88	0.97	0.93	949
vasc	0.93	0.64	0.76	22
accuracy			0.84	1440
macro avg	0.77	0.65	0.70	1440
weighted avg	0.83	0.84	0.83	1440



41. Repeticion aprendizaje transferido, oversampling y loss ponderada
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10, 11 y 12.

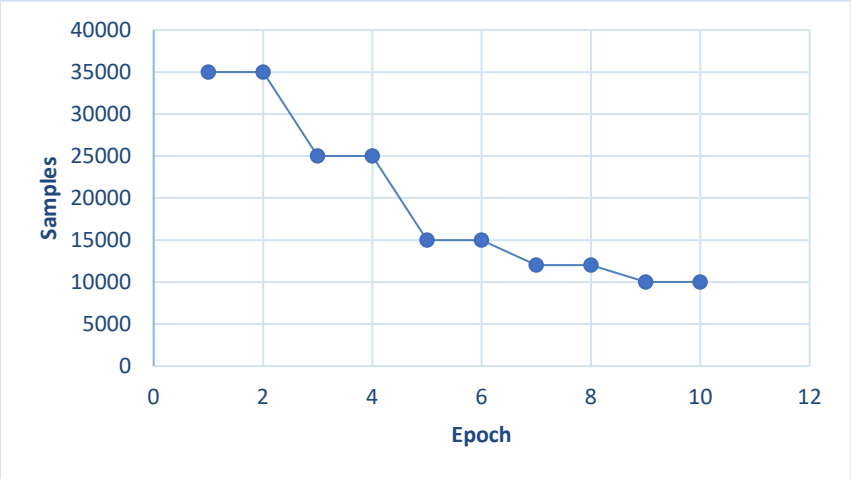
Con la siguiente configuración de variables:

```
#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = False
#SAMPLING
WEIGHTED_SAMPLER = True
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = True
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False

CALCULATE_CLASS_WEIGHTS = True
USE_WEIGTHED_CROSS_ENTROPY_LOSS = True
```

4.8 Sobremuestreo dinámico

El sobremuestreo o *oversampling* dinámico va a consistir en utilizar *oversampling* durante el entrenamiento, pero ir disminuyendo gradualmente su aplicación hasta no utilizarlo en las últimas iteraciones.

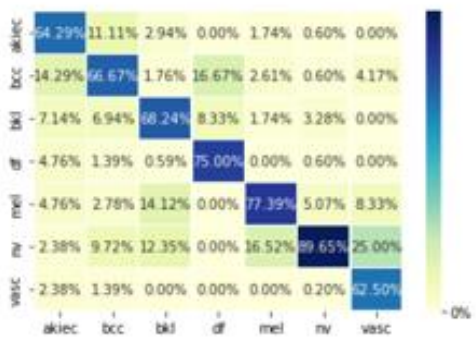
Por lo tanto, empezamos sobre muestreando todas las clases hasta un número elevado de muestras, en este caso 35000 y lo iremos reduciendo progresivamente hasta usar el *dataset* original. El sobremuestreo seguiría el siguiente esquema:



42.Detalle de la distribución del sobremuestreo dinámico
Fuente: Elaboración propia

El resultado de utilizar esta técnica **no ha traído mejoras sustanciales** al modelo, como se puede apreciar en el siguiente resultado donde obtenemos un f1-score de **0.69.**

	precision	recall	f1-score	support
akiec	0.64	0.56	0.60	48
bcc	0.67	0.70	0.68	69
bkl	0.68	0.72	0.70	160
df	0.75	0.47	0.58	19
mel	0.77	0.52	0.62	170
nv	0.90	0.94	0.92	955
vasc	0.62	0.79	0.70	19
accuracy			0.84	1440
macro avg	0.72	0.67	0.69	1440
weighted avg	0.83	0.84	0.83	1440



43. Resultado con aprendizaje transferido y sobremuestreo dinámico
Fuente: Elaboración propia

Para implementar este sobremuestreo dinámico, se modifica el *dataloader* de entrenamiento para que varíe el número de muestras que utiliza cada 2 *epochs*. El código que realiza la acción está en el punto 10.1 del documento TMF.ipynb de Google Colab:

10.1 Módulo de datos dinámico

```
class Ham10000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):

      samples_weight = getSampleWeights()

      print("self.trainer.current_epoch", self.trainer.current_epoch)
      if self.trainer.current_epoch < 2:
       samples_weight_length = 35000  #Oversample to this number
      elif  2 <= self.trainer.current_epoch <4:
        samples_weight_length = 25000
      elif 4 <= self.trainer.current_epoch < 6:
       samples_weight_length = 15000
      elif 6 <= self.trainer.current_epoch < 8:
        samples_weight_length = 12000
      else:
        samples_weight_length = len(samples_weight)

      print("samples_weight_length:" ,  samples_weight_length)

      weighted_train_sampler = torch.utils.data.sampler.WeightedRandomSampler(samples_weight, samples_weight_length, replacement=True)

      train_loader = torch.utils.data.DataLoader(trainDataset, batch_size = batch_size, drop_last = True, sampler = weighted_train_sampler, pin_memory = GPU_MODE)
      return  train_loader
    def val_dataloader(self):
      return torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, shuffle=False, drop_last= True, sampler=valid_sampler, num_workers=num_workers, pin_memory=GPU_MODE)

    def test_dataloader(self):
      return torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, shuffle=False, drop_last= True, sampler=test_sampler, num_workers=num_workers, pin_memory=GPU_MODE)


ham10000_dm = Ham10000DataModule(batch_size=BATCH_SIZE)
```

44. Código del módulo 10.1
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.1, 6, 7, 8, 9, 10.1, 11 y 12.

Con la siguiente configuración de variables:

```
#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = False
#SAMPLING
WEIGHTED_SAMPLER = True
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = True
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = True

CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

4.9 Aprendizaje transferido y congelación de capas

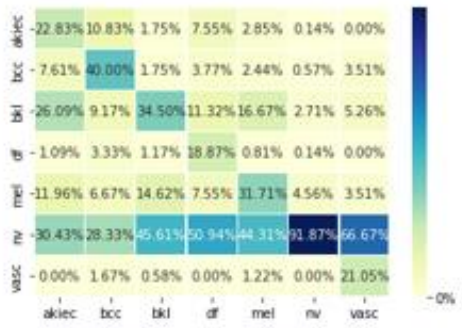
En este nuevo experimento se ha intentado reducir el número de capas de la red con capacidad de aprender, para ver si el resultado mejoraba el de los experimentos anteriores.

En general, no se ha tenido éxito y todos los experimentos han arrojado valores peores que entrenando toda la red. Se han probado las siguientes configuraciones:

- Congelar toda la red menos la última capa
- Congelar toda la red menos las dos últimas capas
- Congelar toda la red menos las tres últimas capas

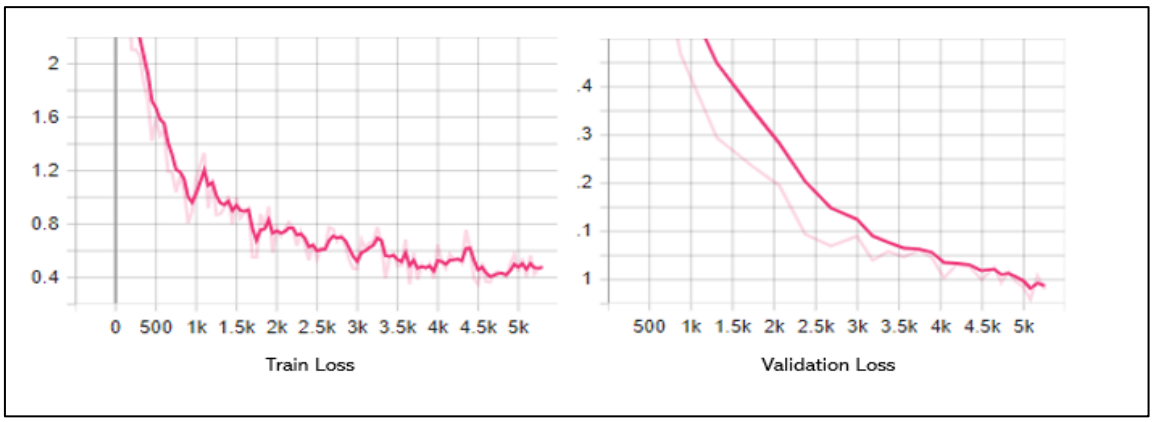
Se presentan ahora los resultados de un entrenamiento a 10 *epochs* con toda la red congelada menos las **dos últimas capas**, como se ha comentado, el rendimiento del modelo empeora notablemente hasta un resultado de f1-score de 0.42.

	precision	recall	f1-score	support
akiec	0.23	0.43	0.30	49
bcc	0.40	0.67	0.50	72
bkl	0.35	0.36	0.35	163
df	0.19	0.50	0.27	20
mel	0.32	0.49	0.38	160
nv	0.92	0.67	0.78	958
vasc	0.21	0.67	0.32	18
accuracy			0.61	1440
macro avg	0.37	0.54	0.42	1440
weighted avg	0.72	0.61	0.64	1440



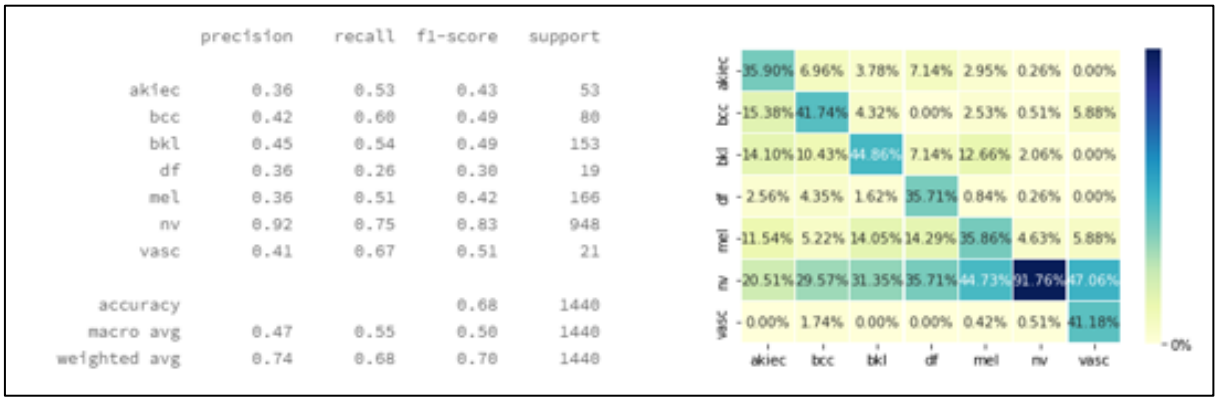
45. Ejemplo resultado
Fuente: Elaboración propia

Para ver si aumentando el número de *epochs*, el modelo podría mejorar, se ha realizado un entrenamiento sin límite de *epochs*, pero el rendimiento de los tests de validación se estanca alrededor de un f1-score de 0.50 a partir de la la *epoch* 15 y ya no mejora, por lo que el entrenamiento se ha cancelado en la *epoch* 25.



46. Función de pérdida durante entrenamiento y validación
Fuente: Elaboración propia

Resultado del test de validación en la epoch 24:



	precision	recall	f1-score	support
akiec	0.36	0.53	0.43	53
bcc	0.42	0.60	0.49	80
bkl	0.45	0.54	0.49	153
df	0.36	0.26	0.30	19
mel	0.36	0.51	0.42	166
nv	0.92	0.75	0.83	948
vasc	0.41	0.67	0.51	21
accuracy			0.68	1440
macro avg	0.47	0.55	0.50	1440
weighted avg	0.74	0.68	0.70	1440

47. Resultado del epoch 24
Fuente: Elaboración propia

La opción de congelación de capas se controla con la variable FREEZE_MODEL.

Si está activada al definir el modelo se congelan las últimas capas con el siguiente código:

6. Creación de la red

```
def createModel():
  model = timm.create_model('tf_efficientnet_b0', pretrained = PRETRAINED, num_classes=7)

  if FREEZE_MODEL:
    count = 0;
    print("Freezing model")
    for param in model.parameters():
      count = count + 1
      if count < 212: #212 to freeze the last two #211 last 3
        param.requires_grad = False

    for param in model.parameters():
      print(param)

  return model


test = createModel()
```

48. Código del módulo 6
Fuente: Elaboración propia

4.10 Conclusiones sobre el uso de las técnicas de mejora

Debido a la reducida cantidad de datos para el entrenamiento de la red, y de que presenta una ratio de desbalance muy acusado, se han empleado diferentes técnicas durante el aprendizaje para intentar mitigar estos problemas.

Sin embargo, solamente se ha podido conseguir un resultado de f1-score de 0.71

En la siguiente sección se plantea el uso de redes generativas para aumentar las muestras e intentar mejorar el entrenamiento de la red.

4.11. Aumento de datos utilizando GANS

Las redes neurales generativas adversarias (o antagónicas), o GANs (Generative Adversarial Networks) son un tipo de sistema de dos redes neuronales complementarias que tienen como objetivo generar imágenes nuevas a partir de un conjunto de imágenes dado [37].

El funcionamiento de las GANs se basa en dos redes compitiendo entre ellas, un Generador y un Discriminador. El objetivo del Discriminador es detectar si una imagen que se le presenta es real o generada artificialmente por el Generador, mientras que el objetivo de este último es el de engañar al Discriminador y que considere que una imagen generada artificialmente es real.

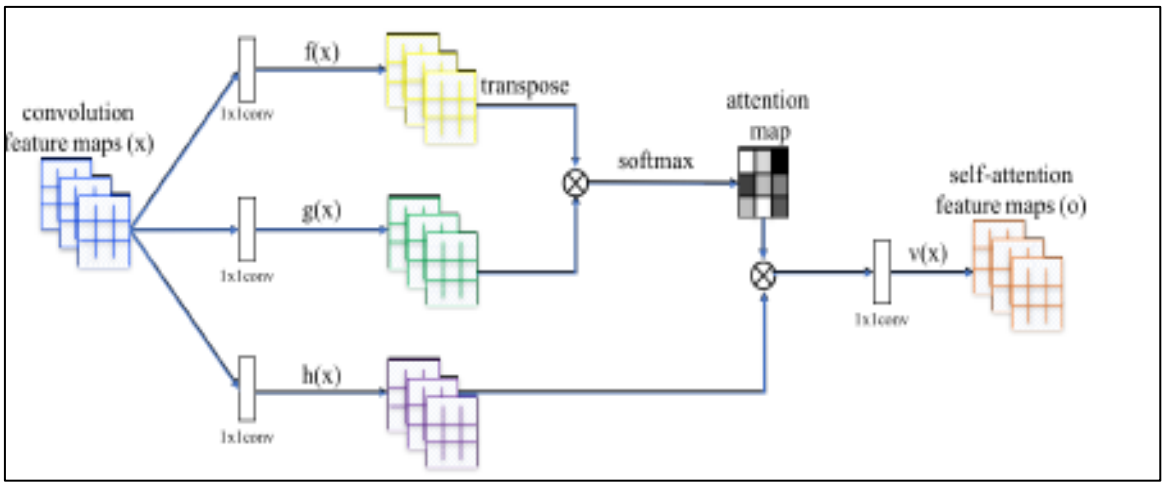
El poder de estas redes surge cuando se consiguen entrenar ambas redes armónicamente, de forma que ambas sean cada vez mejores en su trabajo, y las imágenes generadas se parecen cada vez más a las reales.

Debido a que el conjunto de imágenes HAM 10000 sufre de un grave desbalanceo en el número de imágenes por cada clase, una parte de este trabajo ha consistido en la creación de nuevas imágenes para completar el conjunto de entrenamiento, con el objetivo de mejorar el aprendizaje de la red.

Existen numerosos modelos de GAN, y múltiples artículos sobre su aprendizaje. Para este trabajo nos hemos centrado principalmente en las SAGAN, un tipo de redes GAN presentado en 2019 por Han Zhang, Ian Goodfellow, Dimitris Metaxas, Augustus Odena [38].

La novedad de este tipo de redes es que intentan generar imágenes utilizando información de toda la imagen que intenta aprender a generar, y no solo de detalles locales.

Para esto utiliza el concepto de la autoatención (Self-Attention); en contraste con las capas convolucionales que trabajan de modo local, las capas de autoatención son capaces de utilizar información de bloques distantes



49. Autoatención en CNNs.
Fuente: Self-Attention Generative Adversarial Networks. arXiv:1805.08318, Acceso: 12/12/21

Estas redes, además, son de tipo condicional, de modo que tendrán la capacidad de generar imágenes diferentes para cada clase.

Para la creación y entrenamiento de la GAN, se ha utilizado como base la librería Pytorch StudioGAN.

50. Portada del proyecto Studio GAN
Fuente: https://github.com/POSTECH-CVLab/PyTorch-StudioGAN, Acceso: 12/12/21

Para su utilización con el conjunto de entrenamiento HAM 10000 ha habido que añadir algunas configuraciones. Esto se ha hecho en un fork del proyecto original que se puede consultar en: https://github.com/agusbarba/PyTorch-StudioGAN

Además, para su uso en el entorno de Google Colab ha habido resolver una serie de dependencias y adaptar los comandos de invocación.

El código completo de dichas adaptaciones y de los comandos de entrenamiento utilizados se lista a continuación y también puede consultarse en: https://github.com/agusbarba/TFM_shared/blob/main/TestStudioGAN.ipynb

El documento creado para el uso de la GAN tiene 7 apartados:

- **Drive**: Para conectar Colab con una cuenta de Google Drive donde están las imágenes originales para entrenar la GAN y donde se guardan los resultados

- **Dependencias**: Donde se instalan todas las bibliotecas no presentes por defecto en la instancia de Python proporcionada en Colab.

- **Setup**: Para importar las bibliotecas y conectar con la base de datos wandb [39] donde se guardarán los resultados

- **Instrucciones**: Donde se listan los comandos para utilizar el framework

- **Entrenamiento**: Para lanzar un entrenamiento desde su inicio

- **Generar imágenes**: Para generar un lote de 10.000 imágenes a partir de un modelo ya entrenado

- Continuar entrenamiento: Para continuar un entrenamiento a partir del modelo guardado en un punto de guardado o *checkpoint*

El código correspondiente se detalla a continuación:

Drive

```
In [ ]: from google.colab import drive
        drive.mount('/content/drive')
```

Dependencias

```
In [ ]:
!pip3 install tqdm ninja h5py kornia matplotlib pandas sklearn scipy seaborn wandb
!pip3 install PyYaml click requests pyspng imageio-ffmpeg prdc
!git clone https://github.com/agusbarba/PyTorch-StudioGAN.git
!pip3 install torch==1.10.0+cu111 torchvision==0.11.1+cu111 torchaudio==0.10.0+cu111
             -f https://download.pytorch.org/whl/cu111/torch_stable.html
!pip install --upgrade scikit-learn
!pip install -U PyYAML
```

Setup

```
In [ ]: import torch
        torch.cuda.empty_cache()
        CUDA_VISIBLE_DEVICES = 0
        !wandb login '6db71e761011c9987686e2f307a2825e505ca398'
```

Instrucciones

```
In [ ]: !python3 PyTorch-StudioGAN/src/main.py
```

Entrenamiento

```
In [ ]: !python3 PyTorch-StudioGAN/src/main.py -t -e
            -cfg "/content/PyTorch-StudioGAN/src/configs/HAM10000/My SAGAN-Size 128.yaml"
            -data "/content/drive/My Drive/Master/TFM/128" -save "/content/drive/My Drive/Master/TFM/results"
            --num_workers 2
```

Generar imágenes

```
In [ ]: !python3 PyTorch-StudioGAN/src/main.py -s -best -resume_ct
        -cfg "/content/PyTorch-StudioGAN/src/configs/HAM10000/My SAGAN-Size 128.yaml"
        -ckpt "/content/drive/MyDrive/Master/TFM/results/checkpoints/Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10"
        -data "/content/drive/MyDrive/Master/TFM/128" -save "/content/drive/MyDrive/Master/TFM/results/"
        --num_workers 2
```

Continuar Entrenamiento

```
In [ ]:
!python3 PyTorch-StudioGAN/src/main.py -t -best -e
-cfg "/content/PyTorch-StudioGAN/src/configs/HAM10000/My SAGAN-Size 128.yaml"
-ckpt "/content/drive/MyDrive/Master/TFM/results/checkpoints/Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10"
-data "/content/drive/MyDrive/Master/TFM/128"
-save "/content/drive/MyDrive/Master/TFM/results"
--num_workers 2
```

51. Detalle del documento para gestionar la GAN en Colab
Fuente: Elaboración propia

Para el entrenamiento de la red se ha utilizado el *dataset* original HAM 10000, redimensionado a 128x128. En la imagen inferior se puede ver el fichero de configuración utilizado en el *framework*.

master PyTorch-StudioGAN / src / configs / HAM10000 / My SAGAN-Size 128.yaml Go to file

agusbarba Update My SAGAN-Size 128.yaml Latest commit 34d1ba6 20 days ago History

1 contributor

22 lines (22 sloc) 359 Bytes Raw Blame

```
DATA:
  name: "Ham10000"
  img_size: 128
  num_classes: 7
MODEL:
  g_cond_mtd: "cBN"
  d_cond_mtd: "PD"
  apply_g_sn: True
  apply_d_sn: True
  apply_attn: True
  attn_g_loc: [4]
  attn_d_loc: [1]
LOSS:
  adv_loss: "hinge"
OPTIMIZATION:
  batch_size: 20
  g_lr: 0.0001
  d_lr: 0.0004
  beta1: 0.0
  beta2: 0.999
  d_updates_per_step: 1
  total_steps: 1000000
```

52. Fichero de configuración de la SAGAN
Fuente: Elaboración propia

Se ha entrenado la red durante 22.000 *epochs* y el modelo con mejor puntuación ha sido el generado en 16.000 *epochs* por lo que es el que se ha utilizado para generar las imágenes con las que dopar el *dataset* original.

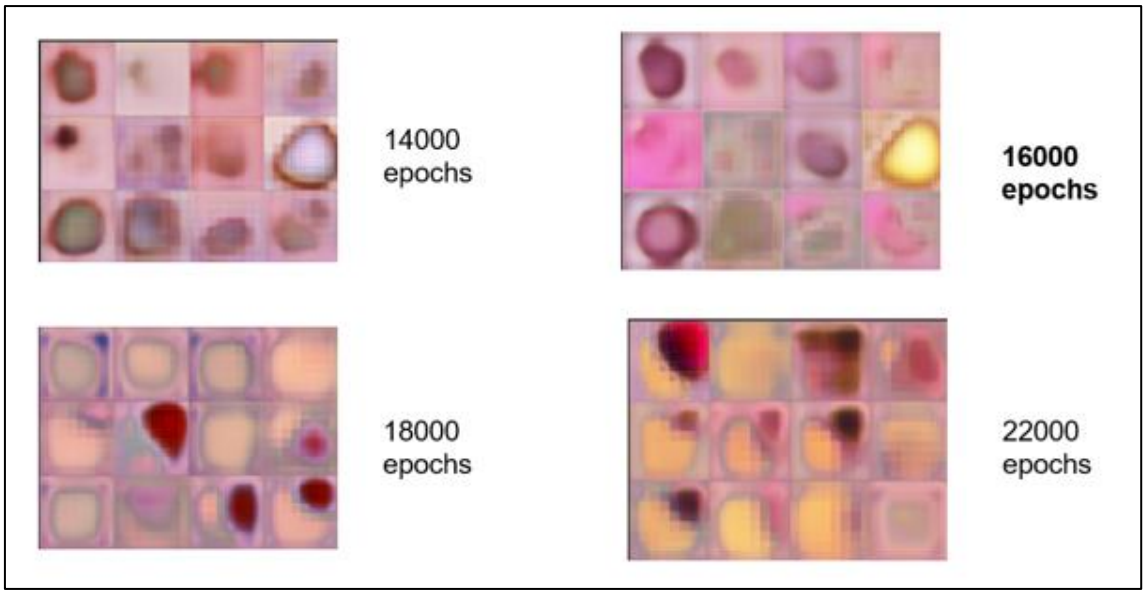
El entrenamiento de este modelo ha resultado un gran reto utilizando el entorno de Google Colab, ya que se necesitan un alto número de recursos, principalmente de GPU y la plataforma penaliza en uso excesivo de éstos restringiendo las horas de uso.

En nuestro caso particular, un entrenamiento de 2000 *epochs* de la GAN podía llevar entre 2 y 3 horas, y la limitación de uso de GPU de Colab es de aproximadamente 4 horas al día.

Para solventar esta limitación, se han utilizado tres cuentas diferentes en paralelo y se han reanudado los entrenamientos utilizando los valores de las redes que se guardan en cada iteración. De esta manera se ha conseguido entrenar la SAGAN hasta 22.000 *epochs* en varias ocasiones.

Sin embargo, para todos los entrenamientos, se obtenían mejores modelos en 16.000 *epochs* que entrenando más la red.

A continuación, veremos un detalle de las imágenes generadas por la GAN en distintos puntos de su entrenamiento:



53. Muestras de la salida de la GAN en diferentes puntos del entrenamiento
Fuente: Elaboración propia

El *framework* mide la distancia de Fréchet (FID) cada 2.000 iteraciones y guarda el mejor modelo. En este caso el de 16.000 (415,1378 en 18.000 vs 361,1874 en 16.000)

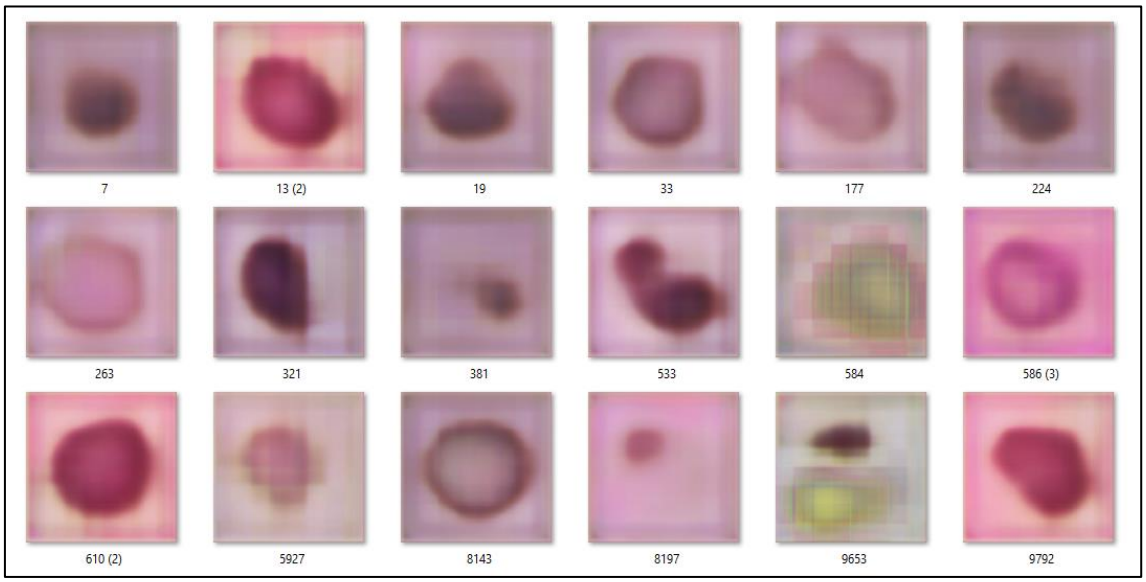
La distancia de Fréchet (FID del inglés Fréchet Inception Distance) compara la media y la covarianza de las imágenes generadas y las imágenes reales. Los valores más bajos indican mejores modelos.[40]

A continuación, se puede ver un detalle del log del entrenamiento donde se muestran los valores calculados:

```
[INFO] 2021-11-27 12:56:40 > Step:  17800 Progress: 1.8% Elapsed: 2:03:04 Gen_loss: 0.5964 Dis_loss: 0.5999 Cls_loss: N/A Topk:  N/A ada_p: N/A
[INFO] 2021-11-27 13:01:54 > Step:  17900 Progress: 1.8% Elapsed: 2:08:18 Gen_loss: 0.7943 Dis_loss: 1.002 Cls_loss: N/A Topk:  N/A ada_p: N/A
[INFO] 2021-11-27 13:07:11 > Step:  18000 Progress: 1.8% Elapsed: 2:13:35 Gen_loss: 0.5959 Dis_loss: 1.11 Cls_loss: N/A Topk:  N/A ada_p: N/A
[INFO] 2021-11-27 13:07:11 > Visualize (num_rows x 8) fake image canvans.
[INFO] 2021-11-27 13:07:11 > Save image canvas to /content/drive/MyDrive/Master/TFM/results/figures/Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10/generated_canvas_18000.png
[INFO] 2021-11-27 13:07:12 > Start Evaluation (18000 Step): Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10
[INFO] 2021-11-27 13:07:12 > Calculate Inception score of generated images (8020 images).
100% 401/401 [03:52<00:00,  1.72it/s]
  0% 0/401 [00:00<?, ?it/s][INFO] 2021-11-27 13:11:05 > Calculate FID score of generated images (8020 images).
100% 401/401 [03:52<00:00,  1.72it/s]
[INFO] 2021-11-27 13:15:10 > Calculate improved precision-recall and density-coverage of generated images (8020 images).
100% 401/401 [05:44<00:00,  1.16it/s]
Num real: 8020 Num fake: 8020
[INFO] 2021-11-27 13:21:26 > Inception score (Step: 18000, 8020 generated images): 1.6422837972640991
[INFO] 2021-11-27 13:21:26 > FID score (Step: 18000, Using train moments): 415.13775266711417
[INFO] 2021-11-27 13:21:26 > Improved Precision (Step: 18000, Using train images): 0.0
[INFO] 2021-11-27 13:21:26 > Improved Recall (Step: 18000, Using train images): 0.0
[INFO] 2021-11-27 13:21:26 > Density (Step: 18000, Using train images): 0.0
[INFO] 2021-11-27 13:21:26 > Coverage (Step: 18000, Using train images): 0.0
[INFO] 2021-11-27 13:21:26 > Best FID score (Step: 16000, Using train moments): 361.187365492305
[INFO] 2021-11-27 13:21:31 > Save model to /content/drive/MyDrive/Master/TFM/results/checkpoints/Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10
```

54. Log entrenamiento SAGAN
Fuente: Elaboración propia

Se han generado 33.000 imágenes falsas, de las 7 clases con un tamaño de 128 x 128. Para su uso en el clasificador, se han escalado hasta 224x224.



55. Imágenes generadas por la GAN
Fuente: Elaboración propia

Para la preparación de los distintos *datasets*, se ha utilizado el IDE de Python, Spyder 4.2.5 trabajando sobre una instalación de Anaconda. Los scripts utilizados han sido los siguientes:

Para reducir el tamaño de las imágenes, utilizando escalado bilinear y *croping* de los laterales se ha utilizado el siguiente script:

```
"""
Created on Sat Oct  9 12:44:15 2021
@author: abarsan
"""
from PIL import Image
from PIL import ImageOps
import os
basedir = "w:\\Ham10000\Images"
inside = os.listdir(basedir)
resolutions = [(60,45),(100, 75),(140, 115),(200, 150),(220, 165)]

resolutionsCenterCrop = [64,96,128,192,224]

for i in inside:
    fullPath = basedir + "\\" + i
    if os.path.isfile(fullPath):

        originalImage = Image.open(fullPath)
        border = (75, 0, 75, 0)
        originalImage = ImageOps.crop(originalImage, border)
        name = os.path.basename(fullPath)
        name, extension = os.path.splitext(name)
        for x in resolutionsCenterCrop:
            width = x
            outputDir = basedir + "\\" + str(width)
            im = originalImage.resize((width, width), Image.BILINEAR)
            im.save(outputDir + "\\" + name + ".png", 'PNG')
            print(name)
```

Para aumentar el tamaño desde las 128x128 que devuelve la GAN hasta el 224x224 que utiliza el clasificador se ha utilizado:

```
"""
Created on Sat Oct  9 12:44:15 2021
@author: abarsan
"""
from PIL import Image
import os
#baseDir =   "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/16000/fake3/"
#outputDir = "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/16000/fake3/bcc/"

baseDir =   "C:/Users/Usuario/Desktop/33000/"
outputDir = "C:/Users/Usuario/Desktop/33000_resized/"

inside = os.listdir(baseDir)
for classes in range(7):
    classFolder = baseDir + str(classes)
    outputClassFolder = outputDir + str(classes)
    print("Starting Folder", classFolder)
    print("Output Folder", outputClassFolder)

    inside = os.listdir(classFolder)
    for i in inside:
        print(i)
        fullPath = classFolder + "/" + i
        if os.path.isfile(fullPath):

            originalImage = Image.open(fullPath)
            width = 224
            height = 224
            im = originalImage.resize((width, height), Image.BILINEAR)
            im.save(outputClassFolder + "/" + i, 'PNG')
```

Una vez conseguidas las imágenes, el objetivo sería aumentar el conjunto de entrenamiento para conseguir mejorar el aprendizaje de la red.

Se diseñó un algoritmo de aprendizaje dinámico en el cual se partiría de un conjunto de entrenamiento muy aumentado con imágenes artificiales, y progresivamente se iría disminuyendo hasta acabar entrenando la red con el *dataset* de training original.

Se han realizado tres entrenamientos distintos.

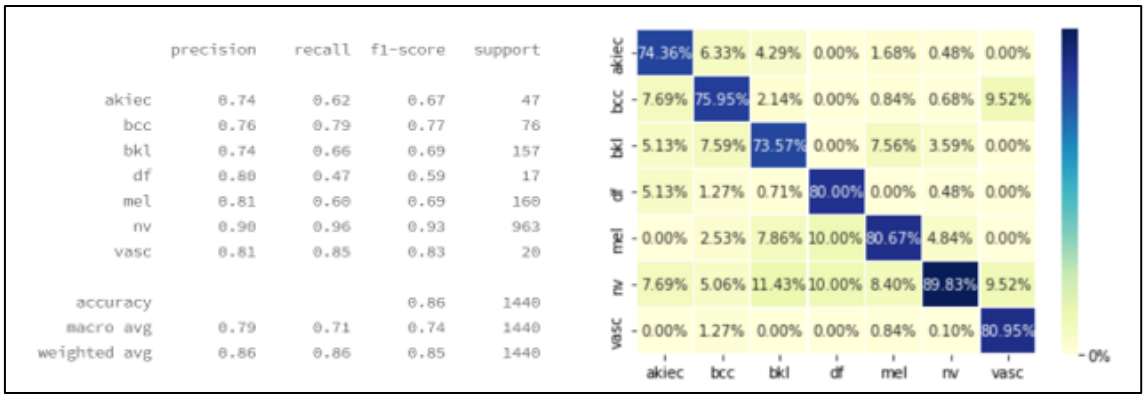
Primer experimento

En el primer caso se partía de un *dataset* en el que se aumentaba el tamaño de las clases hasta 2000 elementos y se iría bajando el número de imágenes artificiales por clase, siguiendo el siguiente esquema del dopaje.

- Epochs 1 y 2: Aumento hasta 2000
- Epochs 3 y 4: Aumento hasta 1000
- Epochs 5 y 6: Aumento hasta 750
- Epochs 7 y 8: Aumento hasta 500
- Epochs 9 y 10: Aumento hasta 250
- Epochs 11 y 12: Dataset original

Las clases con un número de elementos mayor que el aumento objetivo, permanecen inalteradas. Por ejemplo, la clase 6 con más de 4.000 elementos, nunca será aumentada.

Los resultados de este primer experimento son los siguientes:



	precision	recall	f1-score	support
akiec	0.74	0.62	0.67	47
bcc	0.76	0.79	0.77	76
bkl	0.74	0.66	0.69	157
df	0.80	0.47	0.59	17
mel	0.81	0.60	0.69	160
nv	0.90	0.96	0.93	963
vasc	0.81	0.85	0.83	20
accuracy			0.86	1440
macro avg	0.79	0.71	0.74	1440
weighted avg	0.86	0.86	0.85	1440

56. Resultado experimento 1 con GAN
Fuente: Elaboración propia

Con un f1-score de 0. 74 y una accuracy de 0.86

Para añadir los *datasets* aumentados de forma dinámica se diseña un nuevo algoritmo de carga de los datos. En lugar de realizar particiones dinámicas aleatorias del conjunto de datos original cada vez que se realiza un entrenamiento, se van a generar conjuntos de entrenamiento estáticos que se irán empleando conforme avancen las iteraciones.

Para generar los distintos *datasets* con distintos % de dopaje se ha utilizado el siguiente script:

```
"""
Created on Fri Nov 19 20:05:56 2021
@author: abarsan
"""
import os, shutil, random
origin = "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/33000_resized/"
destination = "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/dopped_33000/"
classes = ["akiec", "bcc", "bkl", "df", "mel", "vasc"]
lenghOfRealSets = {"akiec":229, "bcc":360, "bkl":769, "df": 81, "mel": 779, "nv": 4693, "vasc":
99}
finalSize = 3000
for classPath in classes:
    fromDir = origin + classPath
    toDir = destination + str(finalSize) + "/" + classPath
    files = os.listdir(fromDir)
    length = len(files)
    print("Available files ", length)
    print("Real Images for class" ,  classPath, " is ", lenghOfRealSets[classPath])
    testLength = finalSize - lenghOfRealSets[classPath]
    print("Adding ", testLength, " images")
    print("Creating  ", testLength, " files ")
    if testLength > 0:
        filenames = random.sample(files, testLength)
        for fname in filenames:
            srcpath = os.path.join(fromDir, fname)
            shutil.copy(srcpath, toDir)
```

Y la gestión de la carga de los distintos *datasets* se gestiona en un módulo nuevo, el 10.3:

10.3 Módulo de datos dinámico GAN

```
class Haml0000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):

      transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
      ])

      print("self.trainer.current_epoch", self.trainer.current_epoch)
      if self.trainer.current_epoch < 2:
       train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/4700'
       print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/4700")
      elif  2 <= self.trainer.current_epoch < 4:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/4000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/4000")
      elif 4 <= self.trainer.current_epoch < 6:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/3000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/3000")
      elif 6 <= self.trainer.current_epoch < 8:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/2000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/2000")
      elif 8 <= self.trainer.current_epoch < 10:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/1000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/1000")
      elif 10 <= self.trainer.current_epoch < 12:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/500'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/500")
      else:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/doppad_33000/train'
        print("Using original training dir:")

      trainImageDataset = datasets.ImageFolder(train_Dir, transform = transform)
      length = len(trainImageDataset)
      indexes = list(range(length))
      np.random.seed(123)
      np.random.shuffle(indexes)
      train_idx = indexes

      train_sampler = SubsetRandomSampler(train_idx)
      train_loader = torch.utils.data.DataLoader(trainImageDataset, batch_size=batch_size, sampler=train_sampler, pin_memory=GPU_MODE)
      return  train_loader


    def val_dataloader(self):
      return valid_loader
    def test_dataloader(self):
      return test_loader


haml0000_dm = Haml0000DataModule(batch_size=BATCH_SIZE)
```

57. Código del módulo 10.3
Fuente: Elaboración propia

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.2, 6, 7.1, 8, 9, 10.3, 11 y 12.
En el paso 10.3 es necesario reajustar los paths con los que se vayan a utilizar

Con la siguiente configuración de variables:

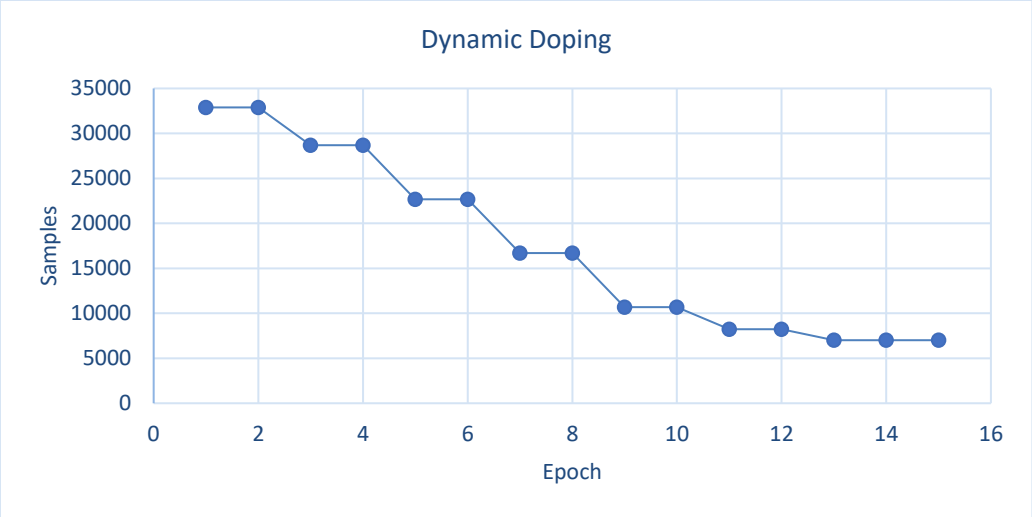
```
#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = True
#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False
CALCULATE_CLASS_WEIGHTS = False
USE_WEIGTHED_CROSS_ENTROPY_LOSS = False
```

Segundo experimento

En el segundo experimento, el planteamiento es el mismo, pero cambiaría la agresividad del uso de imágenes artificiales generadas por la GAN. En este caso se usaría el esquema:

- Epochs 1 y 2: Aumento hasta 4700
- Epochs 3 y 4: Aumento hasta 4000
- Epochs 5 y 6: Aumento hasta 3000
- Epochs 7 y 8: Aumento hasta 2000
- Epochs 9 y 10: Aumento hasta 1000
- Epochs 11 y 12: Aumento hasta 500
- Epochs 12-15: Dataset original

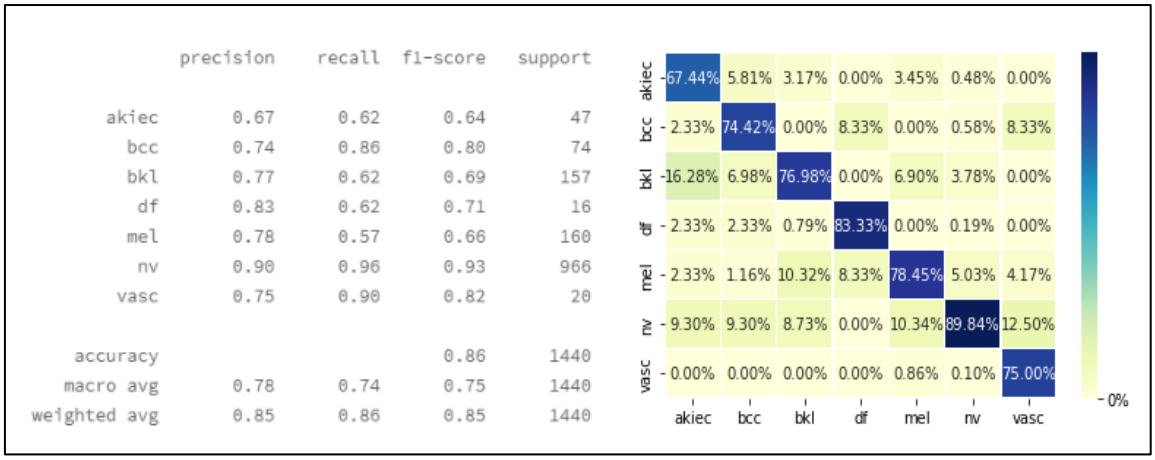
El número total de imágenes usadas sigue el siguiente esquema:



58. Esquema de aumento de datos
Fuente: Elaboración propia

La implementación de este experimento es análoga a la anterior y solo cambian los *datasets* utilizados.

El resultado es similar al anterior, aunque un poco mejor, con **un f1-score de 0.75 y una accuracy de 0.86**



	precision	recall	f1-score	support
akiec	0.67	0.62	0.64	47
bcc	0.74	0.86	0.80	74
bkl	0.77	0.62	0.69	157
df	0.83	0.62	0.71	16
mel	0.78	0.57	0.66	160
nv	0.90	0.96	0.93	966
vasc	0.75	0.90	0.82	20
accuracy			0.86	1440
macro avg	0.78	0.74	0.75	1440
weighted avg	0.85	0.86	0.85	1440

59. Resultado segundo experimento con GAN
Fuente: Elaboración propia

Este es el mejor resultado que se ha obtenido a lo largo de todos los experimentos.

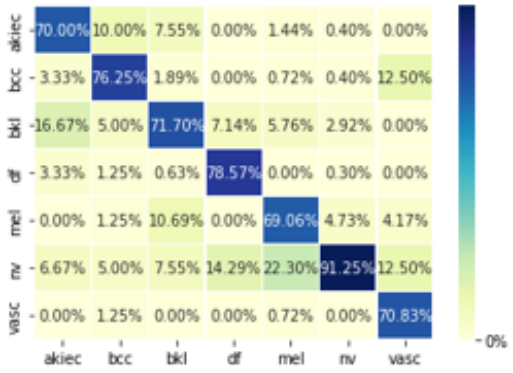
Tercer experimento

Al realizar el segundo experimento se apreciaba que el desempeño del modelo mejoraba las clases menos pobladas de imágenes, lo que nos daba pie a creer que la estrategia del dopaje de la red estaba funcionando. Sin embargo, no era tan bueno como otros experimentos en clases que si tenían imágenes de entrenamiento.

Para corregir esto, se ha tomado el modelo entrenado con las GANs y se ha vuelto a entrena otra 10 *epochs* más con el conjunto de datos original, pero utilizando el algoritmo de la función de pérdida ponderada para intentar corregir el desbalance del *dataset* sin aumentar.

El resultado no ha sido el buscado, ya que, aunque la clase mayoritaria nv mejora, el resto empeoran ligeramente y el resultado global es peor al devolver un f1-score de 0.73.

	precision	recall	f1-score	support
akiec	0.70	0.45	0.55	47
bcc	0.76	0.84	0.80	73
bkl	0.72	0.71	0.71	161
df	0.79	0.65	0.71	17
mel	0.69	0.59	0.64	162
nv	0.91	0.94	0.93	961
vasc	0.71	0.89	0.79	19
accuracy			0.85	1440
macro avg	0.75	0.72	0.73	1440
weighted avg	0.85	0.85	0.85	1440



60. Resultado de reentrenamiento con función de loss ponderada
Fuente: Elaboración propia

El algoritmo de cálculo de pesos dinámico para su envío a la función de pérdida se ha implementado en la sección 7.1

7.1 Función de Loss GAN

```
def normaliseWeights(inputTensor):
  mean = torch.mean(inputTensor)
  normalisation_factor = 1/mean
  output = inputTensor * normalisation_factor
  return output


if USE_WEIGTHED_CROSS_ENTROPY_LOSS:

    classes = [trainImageDataset.targets[i] for i in train_sampler.indices]
    class_tensor = torch.FloatTensor(classes)

    validation_classes = [validImageDataset.targets[i] for i in valid_sampler.indices]
    validation_class_tensor = torch.FloatTensor(validation_classes)

    test_classes = [testImageDataset.targets[i] for i in test_sampler.indices]
    test_class_tensor = torch.FloatTensor(test_classes)

    if WEIGHTED_SAMPLER:
       class_weights = torch.tensor([0.148, 0.144,  0.142, 0.141, 0.140, 0.141 , 0.145])
       print("Oversampled train class weights", class_weights)
    else:
      class_weights = sklearn.utils.class_weight.compute_class_weight('balanced',classes =np.unique(class_tensor),y = class_tensor.numpy())
      class_weights=torch.tensor(class_weights,dtype=torch.float)
      class_weights = normaliseWeights(class_weights)
      print("train class_weights", class_weights)


    validation_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',classes = np.unique(validation_class_tensor),y = validation_class_tensor.numpy())
    validation_class_weights=torch.tensor(validation_class_weights,dtype=torch.float)
    validation_class_weights = normaliseWeights(validation_class_weights)
    print("validation_class_weights", validation_class_weights)
    test_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',classes =np.unique(test_class_tensor),y = test_class_tensor.numpy())
    test_class_weights=torch.tensor(test_class_weights,dtype=torch.float)
    test_class_weights = normaliseWeights(test_class_weights)
    print("test_class_weights", test_class_weights)


    #train_class_weights 35000
    #class_weights = torch.tensor([ 6.949, 6.798,  7.060, 7.180,  7.075, 6.996 ,  6.954])

    ##Weighted Train Set 35000 :  {0: 5031, 1: 5143, 2: 4952, 3: 4869, 4: 4941, 5: 4997, 6: 5027}
    #Weights for oversampled: {0: 6.948916716358576, 1: 6.797588955862337, 2: 7.059773828756058, 3: 7.180119120969398,   4: 7.075490791337786, 5: 6.996197718631179, 6: 6.954445991645116}

    if GPU_MODE:
      trainLossFunction = nn.CrossEntropyLoss(class_weights.cuda())
      validationLossFunction = nn.CrossEntropyLoss(validation_class_weights.cuda())
      testLossFunction = nn.CrossEntropyLoss(test_class_weights.cuda())
    else:
      trainLossFunction = nn.CrossEntropyLoss(class_weights)
      validationLossFunction = nn.CrossEntropyLoss(validation_class_weights)
      testLossFunction = nn.CrossEntropyLoss(test_class_weights)
    print("Using Weighted Loss Function")

else:
  print("nn.CrossEntropyLoss()")
  trainLossFunction = nn.CrossEntropyLoss()
  validationLossFunction = nn.CrossEntropyLoss()
  testLossFunction = nn.CrossEntropyLoss()
```

61. Detalle del apartado 7.1
Fuente: Elaboración propia

También se han añadido ligeras modificaciones al apartado 5, creando el nuevo apartado 5.3 y al apartado 10, creando el módulo de datos dinámico para reentrenamiento de GAN 10.4

Para reproducir los resultados de este ejercicio utilizando el documento de Google Colab se deben ejecutar los pasos:

0, 1, 2, 3, 4, 5.3, 6, 7.1, 8, 9, 10.4, 11 y 12.

Con la siguiente configuración de variables:

```
#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = True
#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False
CALCULATE_CLASS_WEIGHTS = True
USE_WEIGTHED_CROSS_ENTROPY_LOSS = True
```

4.12 Resumen de resultados

A continuación, se presenta el resumen de los resultados obtenidos con las distintas configuraciones probadas.

El mejor resultado de f1-score se ha conseguido con aprendizaje transferido y aumentando el conjunto de entrenamiento con un gran número de imágenes generadas con la SAGAN.

62. Resumen final de resultados.
Fuente: Elaboración propia

5. Conclusiones

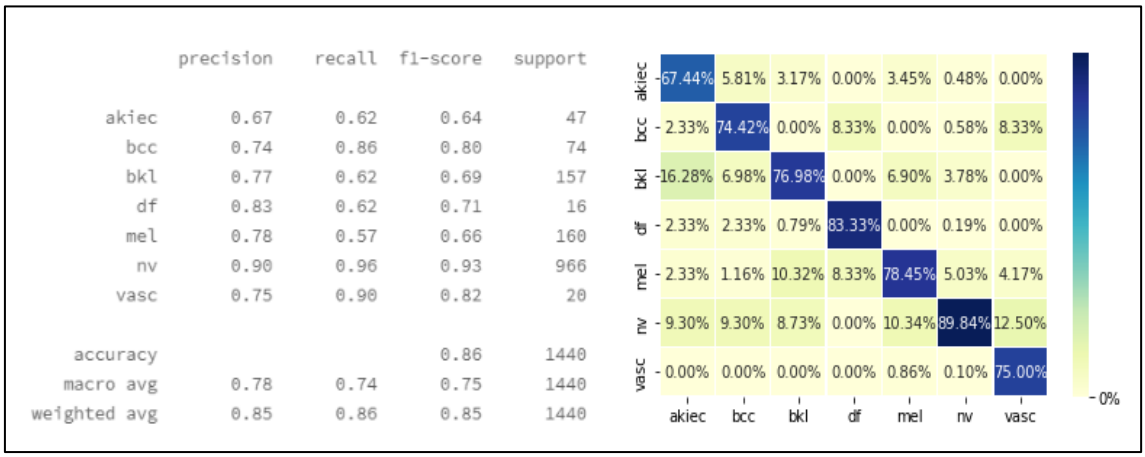
5.1 Conclusiones y reflexiones

La primera conclusión de ese trabajo es que hoy en día es posible afrontar el problema de entrenar una red neural con un coste de hardware y software muy reducido. Únicamente sería necesario tener una conexión a internet y un navegador en un ordenador razonablemente potente.

La plataforma Colab en su versión gratuita, no es óptima, pero es suficiente para realizar los entrenamientos y existe una excelente oferta de software open-source dedicado a deep-learning.

La segunda conclusión es que los resultados de nuestro modelo están cerca de los que podría obtener un experto.

Combinando la inspección visual y de imágenes dermatoscópicas, la tasa de detección de melanomas de un dermatólogo se establece entre un 75% y un 84% [41] citando a [42][43], rango en el que estarían los resultados de este trabajo:



	precision	recall	f1-score	support
akiec	0.67	0.62	0.64	47
bcc	0.74	0.86	0.80	74
bkl	0.77	0.62	0.69	157
df	0.83	0.62	0.71	16
mel	0.78	0.57	0.66	160
nv	0.90	0.96	0.93	966
vasc	0.75	0.90	0.82	20
accuracy			0.86	1440
macro avg	0.78	0.74	0.75	1440
weighted avg	0.85	0.86	0.85	1440

63. Resultado experimento 2 con GAN
Fuente: Elaboración propia

En resumen, se han logrado todos los objetivos iniciales con unos resultados muy aceptables, dadas las limitaciones del hardware descritas y el haber utilizado imágenes de un tamaño relativamente reducido.

Personalmente, me hubiera gustado saber hasta donde habría llegado la mejora de haber podido entrenar mejor la GAN y de haber tenido el hardware para poder entrenar otros modelos más modernos y potentes.

Pero, teniendo en cuenta el tiempo disponible, los recursos, mi formación inicial en la materia y el bajo coste del proyecto, estoy muy satisfecho.

Además, considero que la solución tiene mucho margen de mejora. Esto se detallará más adelante en la sección de “Líneas de trabajo futuro”.

Como reflexiones personales lo que destaco es que además de haber aprendido mucho desde el punto de vista técnico, cosa que tengo que agradecer al director de este trabajo, el profesor Longlong Yu, también he podido adentrarme el mundo del machine learning, las competiciones de Kaggle.com [44] o la inmensa fuente de información que es paperswithcode.com [45]

He trabajado por primera vez con PyTorch y con PyTorch Lightning y me parece asombrosa la cantidad de información disponible y la de gente dispuesta a proporcionar material y ayuda a los demás. Siendo proyectos *open-source* tienen un respaldo muy impresionante.

Como lecciones aprendidas puedo enumerar las siguientes:

- Encontrar *datasets* para entrenar una red es complicado, si nos apartamos de los canónicos tipo ImageNet o MNIST

- Se puede usar gratuitamente el hardware de otros, pero no es algo óptimo, especialmente si son cuentas con uso limitado como Colab en su versión gratuita.

- Derivado de lo anterior, para utilizar redes grandes o entrenamientos muy exigentes es necesario contar con hardware propio o una cuenta profesional de Colab, que no se encuentra disponible en España.

- Hay mucho software *open-source* disponible y que además de ser muy potente, tiene un gran soporte y mucha información y ejemplos disponibles.

- Que la técnica de aprendizaje transferido es muy potente y puede mejorar mucho los resultados cuando el conjunto de datos original no es muy completo o está desbalanceado

- El aprendizaje profundo es un tema en ebullición. Hay muchísima bibliografía sobre el tema, y a cada momento parece que el ritmo de aparición de nuevos artículos no para de crecer.

Como reflexión final, bajo mi punto de vista, el tiempo de realización del TFM se acaba haciendo corto. Desde que se decide la temática y se planifica hasta que se hace la segunda entrega y se empieza a escribir la memoria final apenas tenemos 10 semanas. Esto es un poco abrumador y a veces implica forzar los plazos de la investigación, pero entiendo que el tiempo es el que es, y siempre se puede continuar con la línea de trabajo después de la finalización de los estudios.

5.2 Planificación y metodología

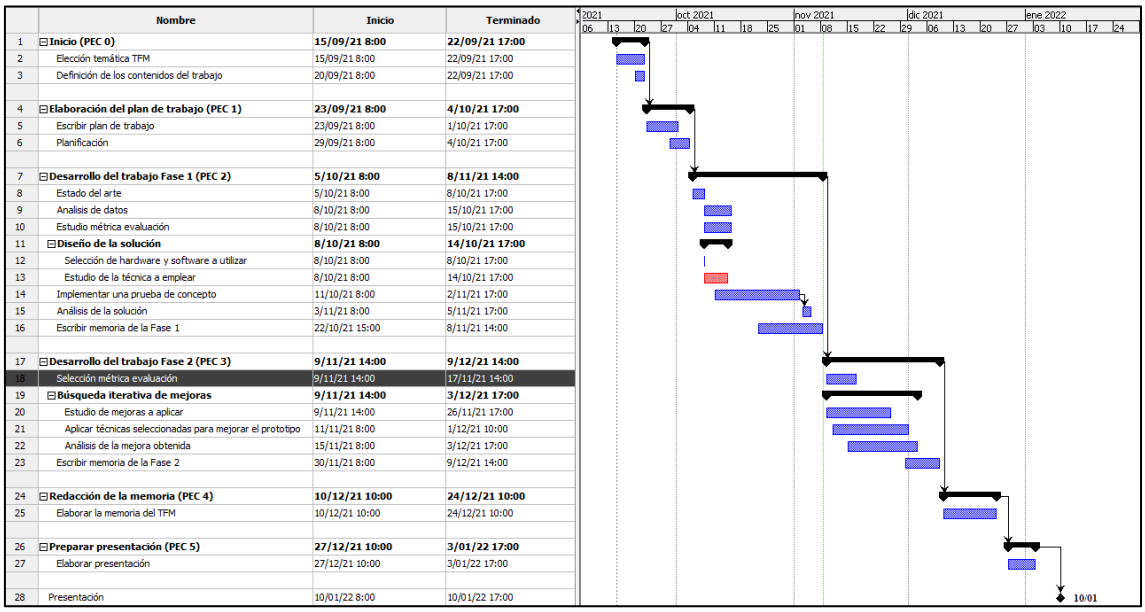
Este trabajo se ha planificado siguiendo los hitos marcados por la asignatura; esto ha sido una ayuda ya que se han podido realizar entregas parciales e ir avanzando en el desarrollo del trabajo.

Este tipo de metodología permite avanzar y corregir desviaciones. También es una forma de forzar al estudiante a pensar en cómo dividir el trabajo y enfrentarse a los problemas uno por uno.

En el caso de mi trabajo en particular, además de las entregas pautadas en la asignatura, se han realizado *sprints* semanales, en los que se le presentaban al director los avances y problemas encontrados, y se marcaban los objetivos de la semana siguiente. Esto ha permitido llevar un ritmo muy constante de trabajo sin descanso que no ha sido fácil, pero creo que ha sido muy efectivo.

Se han respetado las planificaciones y la única desviación detectada es la selección de una métrica para el modelo que estaba prevista para la Fase 1 y finalmente se ha realizado en la primera semana de la Fase 2.

Esto ha sido reflejado en el diagrama de Gantt corregido presentado en la entrega correspondiente.



	Nombre	Inicio	Terminado
1	**Inicio (PEC 0)**	**15/09/21 8:00**	**22/09/21 17:00**
2	Elección temática TFM	15/09/21 8:00	22/09/21 17:00
3	Definición de los contenidos del trabajo	20/09/21 8:00	22/09/21 17:00
4	**Elaboración del plan de trabajo (PEC 1)**	**23/09/21 8:00**	**4/10/21 17:00**
5	Escribir plan de trabajo	23/09/21 8:00	1/10/21 17:00
6	Planificación	29/09/21 8:00	4/10/21 17:00
7	**Desarrollo del trabajo Fase 1 (PEC 2)**	**5/10/21 8:00**	**8/11/21 14:00**
8	Estado del arte	5/10/21 8:00	8/10/21 17:00
9	Analisis de datos	8/10/21 8:00	15/10/21 17:00
10	Estudio métrica evaluación	8/10/21 8:00	15/10/21 17:00
11	**Diseño de la solución**	**8/10/21 8:00**	**14/10/21 17:00**
12	Selección de hardware y software a utilizar	8/10/21 8:00	8/10/21 17:00
13	Estudio de la técnica a emplear	8/10/21 8:00	14/10/21 17:00
14	Implementar una prueba de concepto	11/10/21 8:00	2/11/21 17:00
15	Análisis de la solución	3/11/21 8:00	5/11/21 17:00
16	Escribir memoria de la Fase 1	22/10/21 15:00	8/11/21 14:00
17	**Desarrollo del trabajo Fase 2 (PEC 3)**	**9/11/21 14:00**	**9/12/21 14:00**
18	Selección métrica evaluación	9/11/21 14:00	17/11/21 14:00
19	**Búsqueda iterativa de mejoras**	**9/11/21 14:00**	**3/12/21 17:00**
20	Estudio de mejoras a aplicar	9/11/21 14:00	26/11/21 17:00
21	Aplicar técnicas seleccionadas para mejorar el prototipo	11/11/21 8:00	1/12/21 10:00
22	Análisis de la mejora obtenida	15/11/21 8:00	3/12/21 17:00
23	Escribir memoria de la Fase 2	30/11/21 8:00	9/12/21 14:00
24	**Redacción de la memoria (PEC 4)**	**10/12/21 10:00**	**24/12/21 10:00**
25	Elaborar la memoria del TFM	10/12/21 10:00	24/12/21 10:00
26	**Preparar presentación (PEC 5)**	**27/12/21 10:00**	**3/01/22 17:00**
27	Elaborar presentación	27/12/21 10:00	3/01/22 17:00
28	Presentación	10/01/22 8:00	10/01/22 17:00

64. Planificación con la desviación corregida
Fuente: Elaboración propia

5.3 Líneas de trabajo futuro

Mejora del Hardware

Realizar un trabajo como este utilizando únicamente una herramienta como Google Colab es al mismo tiempo maravilloso y complicado.

Por un lado, permite que cualquier investigador del mundo pueda acceder a este hardware de una manera gratuita. A mí me ha permitido hacer un trabajo que con los medios que tenía a mi disposición, me habría sido totalmente imposible. Por lo tanto, no puedo estar más que agradecido a esta plataforma.

Sin embargo, es cierto que realizar determinados entrenamientos es complicado, especialmente en el caso de las GANs, que requieren mucha potencia de GPU durante mucho tiempo.

Colab está diseñado como una herramienta interactiva, por lo que cada cierto tiempo pregunta con un *captcha* si el investigador está presente. Esto es realmente estresante, porque hay que estar muy pendiente de los entrenamientos y cuando son largos, no es la situación ideal. Además, los recursos de GPU que ofrece Colab están limitados por usuario, de forma que un uso muy alto de recursos penaliza en el número de horas al día que se puede utilizar la plataforma.

Existe una versión avanzada y de pago de la plataforma llamada Colab Pro, que permite a usuarios con necesitades altas de recursos beneficiarse del hardware durante mucho más tiempo, sin embargo, está limitado a un número pequeño de países y por desgracia, España todavía no está entre ellos.

Country of your billing address:	Google entity:
United States	Google LLC 1600 Amphitheatre Parkway Mountain View, CA USA 94043
Brazil	Google LLC 1600 Amphitheatre Parkway Mountain View, CA USA 94043
Canada	Google LLC 1600 Amphitheatre Parkway Mountain View, CA USA 94043
France	Google Commerce Limited ("GCL") Gordon House, Barrow Street Dublin 4, Ireland
Germany	Google Commerce Limited ("GCL") Gordon House, Barrow Street Dublin 4, Ireland
India	Google Asia Pacific Pte. Ltd. ("GAP") 70 Pasir Panjang Road, #03-71 Mapletree Business City II, Singapore 117371
Japan	Google Asia Pacific Pte. Ltd. ("GAP") 70 Pasir Panjang Road, #03-71 Mapletree Business City II, Singapore 117371
Russia	Google Commerce Limited ("GCL") Gordon House, Barrow Street Dublin 4, Ireland
Thailand	Google Asia Pacific Pte. Ltd. ("GAP") 70 Pasir Panjang Road, #03-71 Mapletree Business City II, Singapore 117371
United Kingdom	Google Commerce Limited ("GCL") Gordon House, Barrow Street Dublin 4, Ireland

65. Paises donde se puede utilizar Colab Pro
Fuente: https://colab.research.google.com/pro/terms. Acceso: 19/12/21

Por lo tanto, una de las cosas más importantes a la hora de continuar este trabajo sería poder contar con una cuenta de Colab Pro o un hardware local lo suficientemente potente como para poder entrenar modelos de red más grandes y avanzados.

Escalar EfficientNet

El escalado de la red es la base del diseño mismo de EfficientNet, la red que es el corazón de este trabajo. El título del artículo que la describe ya nos indica esta idea: ***EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*** [20].

Una vez llegado al límite el modelo b0, una de las formas naturales de mejorar el desempeño del modelo obtenido sería aumentar su tamaño y su número de parámetros.

En el artículo citado, los autores describen una mejora desde alrededor del 77% de Accuracy en Imagenet con la versión b0, hasta un 84.3% utilizando la versión b7, por lo que se podría esperar una mejora también en el HAM 10000.

Utilizar las imágenes con su dimensión original

Debido a las limitaciones del hardware utilizado para el entrenamiento y a que la versión b0 de EfficientNet está optimizada para un tamaño de 224x224 píxeles, se ha reducido el tamaño de las imágenes utilizadas para el entrenamiento de la red. En paralelo al escalado descrito en el apartado anterior, se podría aumentar el tamaño de las imágenes, ya que versiones más avanzadas de EfficientNet, están diseñadas para trabajar con tamaños de imagen mayores.

Sería muy interesante saber qué resultados se podrán obtener repitiendo todo el proceso llevado a cabo en este trabajo con las imágenes originales del *dataset*.

Mejorar el entrenamiento de la SAGAN

Ya se ha comentado que el entrenamiento de modelos pesados en Google Colab es complicado. El modelo que se ha utilizado de la SAGAN para generar las imágenes fue conseguido con un ciclo de entrenamiento de 16.000 epochs.

Con mayor capacidad de hardware y más tiempo, probablemente se podrían obtener mejores resultados. Por ejemplo, en su paper ***Generating Highly Realistic Images of Skin Lesions with GANs*** [46] Baur, Albarqouni y Navab entrenan una PGAN durante 3 millones de *epochs* y obtienen grandes resultados con imágenes del HAM 10000.

Usar otras GANs

En los últimos años han aparecido modelos que han superado a la SAGAN en resultados [47]. Redes modernas basadas en el concepto de redes residuales grandes como por ejemplo REACGAN [48], con unos resultados muy buenos en la tarea de generación de imágenes.

Hoy en día, la REACGAN representa el estado del arte de las GAN condicionales y sería el mejor candidato para este proyecto si no fuera por las necesidades tan altas de hardware que requiere su entrenamiento. El resultado del entrenamiento en el entorno de Google Colab fue el siguiente:

```
[INFO] 2021-11-24 22:07:42 > Start training!
[INFO] 2021-11-24 22:35:53 > Step:     100 Progress: 0.0% Elapsed: 0:28:10 Gen_loss: 3.34 Dis_loss: 4.41 Cls_loss: 2.743 Topk:  N/A ada_p: N/A
[INFO] 2021-11-24 23:04:14 > Step:     200 Progress: 0.0% Elapsed: 0:56:31 Gen_loss: 2.402 Dis_loss: 4.699 Cls_loss: 2.795 Topk:  N/A ada_p: N/A
[INFO] 2021-11-24 23:32:36 > Step:     300 Progress: 0.1% Elapsed: 1:24:53 Gen_loss: 3.896 Dis_loss: 3.975 Cls_loss: 2.658 Topk:  N/A ada_p: N/A
[INFO] 2021-11-25 00:00:51 > Step:     400 Progress: 0.1% Elapsed: 1:53:08 Gen_loss: 1.824 Dis_loss: 3.98 Cls_loss: 2.619 Topk:  N/A ada_p: N/A
```

66. Detalle log entrenamiento REACGAN
Fuente: Elaboración propia

Como se puede apreciar, apenas 400 *epoch* ya requieren de casi dos horas de máquina, por lo que se desechó la idea de usar REACGAN y se enfocó todo el esfuerzo del trabajo hacia el uso de SAGAN.

Utilizar técnicas tradicionales de *data augmentation*

En este trabajo se ha utilizado una GAN para obtener un aumento de los datos de entrenamiento. Una solución complementaria es el uso de otras técnicas de *data augmentation* tradicionales basadas en pequeñas alteraciones del conjunto de datos original [49].

Estas modificaciones pueden ser translaciones, giros, cambios de escala o en la luz de las imágenes y han demostrado ser una técnica computacionalmente poco costosa y muy útil a la hora de mejorar un entrenamiento con un *dataset* reducido. Por ejemplo, para el *dataset* MNIST se ha obtenido mejores resultados que utilizando GANs.[50]

Dogs vs Goldfish	
Augmentation	Val. Acc.
None	0.855
Traditional	0.890
GANs	0.865
Neural + No Loss	**0.915**
Neural + Content Loss	0.900
Neural + Style	0.890
Control	0.840

Table I: Quantitative Results on Dogs vs Goldfish

Dogs vs Cat	
Augmentation	Val. Acc.
None	0.705
Traditional	**0.775**
GANs	0.720
Neural + No Loss	0.765
Neural + Content Loss	0.770
Neural + Style	0.740
Control	0.710

Table II: Quantitative Results on Dogs vs Cats

MNIST 0's and 8's	
Augmentation	Val. Acc.
None	0.972
Neural + No Loss	**0.975**
Neural + Content Loss	0.968

Table III: MNIST

67. Resultados con distintas data augmentation en el dataset MNIST
Fuente: The Effectiveness of Data Augmentation in Image Classification using Deep Learning [50]. Acceso 20/12/21

Varios diagnósticos

Otra forma de mejorar el desempeño de este sistema podría ser entrenar distintos modelos de redes y con distintos tipos de entrenamientos para, al final, emitir un diagnóstico que sea una media ponderada de todos ellos.

Aplicaciones

Para concluir se procede a enumerar posibles aplicaciones prácticas del modelo que se ha entrenado.

Las imágenes con las que se ha entrenado el modelo son imágenes dermatológicas. Este tipo de imagen presenta unas características determinadas de procesado que sirven para realzar los detalles importantes en el diagnóstico.

Todo este procesado también es susceptible de ser automatizado, por lo que podemos pensar en una aplicación de móvil que tome fotografías de una calidad razonable, y que se podrían evaluar después con la red entrenada para emitir un diagnóstico automatizado. De este modo tendríamos un sistema de diagnóstico en un dispositivo muy asequible.

Ese protocolo de diagnóstico se podría integrar a nivel sanitario, de forma que los médicos de atención primaria pudieran contar con un diagnóstico inmediato que les orientara sobre cómo proceder con un paciente que va a la consulta con una lesión en la piel.

Pero también se podría pensar en sistemas automatizados disponibles para el gran público, en forma de aplicaciones de móvil, por ejemplo, con las que la población pudiera tener acceso a una herramienta de autodiagnóstico.

Por supuesto, esta opción se plantea con todas las reservas y matices que un software de tipo clínico conllevaría: indicando que el diagnóstico automatizado no substituye a un diagnóstico colegiado, los márgenes de acierto del sistema, que se podrían incluso desglosar por tipo de diagnóstico, y siempre incluyendo la aceptación de descargos de responsabilidad en el acuerdo de licencia de usuario final.

Para este tipo de aplicación, probablemente la capacidad de diagnóstico del sistema tendría que ser mucho más alta de lo que se ha conseguido en este trabajo. Hablamos de porcentajes de acierto superiores al 99%, al tratarse de imagen médica, pero es un camino que probablemente se andará, ya que antes o después se publicarán modelos que alcancen esos resultados.

6. Glosario

Accuracy: Es una métrica que en aprendizaje profundo describe el comportamiento de un modelo. Se calcula como la ratio entre el número de predicciones correctas y el total de predicciones realizadas. Se utiliza principalmente en sistemas con las clases balanceadas.

Baseline: Se le denomina así al resultado que produce un modelo básico. Es práctica común en machine learning el crear primero una solución inicial no muy compleja para después añadir complejidad buscando mejorar esa solución.

Captcha: Prueba utilizada para saber si el usuario es un humano.

Dataset: En castellano conjunto de datos En aprendizaje automático se refiere a un conjunto de datos utilizado en el aprendizaje de una red.

Data augmentation: Se refiere al conjunto de técnicas que se utilizan para incrementar el número de datos disponibles mediante la adición de copias de los datos existentes con ligeras modificaciones o mediante técnicas generativas.

Deep Learning: En castellano Aprendizaje Profundo. Es una rama del Machine Learning o Aprendizaje Automático en la cual el aprendizaje de la máquina se refiere a realizar tareas propias de los organismos vivos y en particular de los seres humanos, como reconocimiento de imágenes, del habla, hacer clasificaciones, predicciones, etc. Se utiliza el término inglés “Deep” (profundo) porque emplea para su objetivo redes neuronales profundas. Las redes neuronales profundas hacen referencia a que contienen varias capas ocultas entre las capas de entrada y salida.

Dermatología: Es la especialidad de la medicina dedicada al estudio de las enfermedades de la piel.

Epoch: En machine learning corresponde a un paso del conjunto de entrenamiento completo por el algoritmo de entrenamiento.

F1-score: Es una métrica que se utiliza en deep learning y se calcula como la media armónica entre precisión (*precision*) y la exhaustividad (*recall*).

GPU: Es el núcleo de procesado de una tarjeta gráfica. Se utiliza en entrenamiento de sistemas de machine Learning debido a su capacidad de procesado en paralelo.

Internet Of Things (comúnmente IoT): En castellano, Internet de las cosas. Es un concepto que se utiliza para describir la interconexión mediante Internet de objetos de uso cotidiano.

Logging: El logging es la acción de guardar en un fichero de registro o de log, los acontecimientos que afectan a un proceso.

Machine Learning: En castellano aprendizaje automático. Es una disciplina de la inteligencia artificial que describe sistemas que son capaces de aprender automáticamente.

Outlier: Es un valor atípico en una observación que parece incompatible con los otros datos.

Precision: Es una métrica que se utiliza en deep learning y se calcula como la ratio entre el número de verdaderos positivos y el número de verdaderos positivos y falsos positivos.

Recall: En castellano exhaustividad. Es una métrica que se utiliza en deep learning y se calcula como la ratio entre el número de verdaderos positivos y el número de verdaderos positivos y falsos negativos.

Red Convolucional: Se refiere a una red neuronal en la cual algunas de sus capas internas son capas de convolución. Esto significa que emplean la operación matemática de convolución a sus entradas, para obtener sus salidas.

Red neuronal o neural: Es un modelo formado por nodos interconectados que intentan imitar la estructura de un cerebro biológico.

Seed: En castellano semilla. Es un número aleatorio que se utiliza para inicializar una operación de generación de números aleatorios (o pseudoaleatorios en este caso).

Sprint: En metodologías ágiles, en particular Scrum, un sprint es un ciclo de ejecución corto, de entre una semana y un mes, en el que se realiza una parte de un proyecto, que tiene que tener resultados propios.

Tensor: En computación se denomina tensor a un vector o matriz que contienen datos.

7. Bibliografía

[1] https://medium.com/neuralmagic/2012-a-breakthrough-year-for-deep-learning-2a31a6796e73, 12/2021

[2] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, Commun. ACM 60, 2017

[3] N Cascinelli, M Ferrario, T Tonelli, E Leo, A possible new tool for clinical diagnosis of melanoma: the computer, 361-367, J Am Acad Dermatol, 6(2 Pt 1), 1987, doi: 10.1016/s0190-9622(87)70050-4

[4] Binder, M. et al. Application of an artificial neural network in epiluminescence microscopy pattern analysis of pigmented skin lesions: a pilot study, pp. 460-465, Br J Dermatol, 130, 1994

[5] https://www.cancer.org/es/cancer/cancer-de-piel-tipo-melanoma.html, Fecha de acceso: 23/12/2021

[6] Linares, M, Zakaria A, Nizran P, Skin Cancer, 645-659 Prim Care, 42, 2015, Doi : 10.1016/j.pop.2015.07.006

[7] https://www.muycomputer.com/2021/08/24/precios-de-las-tarjetas-graficas, Fecha de acceso: 14/12/2021

[8] Tschandl, P., Rosendahl, C, Kittler, H. “The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions”. Sci Data 5, 180161,2018, https://doi.org/10.1038/sdata.2018.161

[9] Steppan J, Hanke S, Analysis of skin lesion images with deep Learning, arXiv:2101.03814v1 [eess.IV], 2021

[10] Krizhevsky A, Sutskever I, Hinton G.E., Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems, pp. 1097-1105, 2012

[11] Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M et al., Imagenet large scale visual recognition challenge, International journal of computer vision, vol. 115, no. 3, pp. 211-252, 2015.

[12] https://image-net.org/, Fecha de acceso: 19/12/2021

[13] Dai Z, Liu H, Le QV, Tan M, CoAtNet: Marrying Convolution and Attention for All Data Sizes, 2021

[14] Tschandl P., Rosendahl C, Kittler H. The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions, Sci Data 5, 180161,2018, https://doi.org/10.1038/sdata.2018.161

[15] Codella N, Rotemberg V, Tschandl P, Emre Celebi M, Dusza S, Gutman D, Helba B, Kalloo A, Liopyris K, Marchetti M, Kittler H, Halpern A, Skin Lesion Analysis Toward Melanoma Detection 2018: A Challenge Hosted by the International Skin Imaging Collaboration (ISIC) https://arxiv.org/abs/1902.03368

[16] https://challenge2018.isic-archive.com, Fecha de acceso: 10/12/2021

[17] https://research.google.com/colaboratory/intl/es/faq.html, Fecha de acceso: 15/12/2021

[18] https://es.wikipedia.org/wiki/Proyecto_Jupyter, Fecha de acceso: 5/12/2021

[19] https://es.wikipedia.org/wiki/Unidad_de_procesamiento_tensorial, Fecha de acceso: 16/12/2021

[20] https://es.wikipedia.org/wiki/Anaconda_(distribuci%C3%B3n_de_Python), Fecha de acceso: 16/12/2021

[21] Tan M, Le QV, EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, International Conference on Machine Learning, 2019, arXiv:1905.11946 [cs.LG]

[22] https://es.wikipedia.org/wiki/PyTorch, Fecha de acceso: 16/12/2021

[23] https://itelligent.es/es/conoces-pytorch-herramienta-open-source-la-puedes-crear-redes-neuronales/, Fecha de acceso: 16/12/2021

[24] https://github.com/PyTorchLightning/pytorch-lightning, Fecha de acceso: 16/12/2021

[25] https://fastai.github.io/timmdocs/, Fecha de acceso: 16/12/2021

[26] https://github.com/fastai/timmdocs, Fecha de acceso: 16/12/2021

[27] https://github.com/rwightman, Fecha de acceso: 16/12/2021

[28] https://scikit-learn.org/stable/, Fecha de acceso: 17/12/2021

[29] https://torchmetrics.readthedocs.io/en/latest/, Fecha de acceso: 17/12/2021

[30] https://programmerclick.com/article/16281462425/, Fecha de acceso: 18/12/2021

[31] https://www.pytorchlightning.ai/blog/dataloaders-explained, Fecha de acceso: 5/10/2021

[32] Mahbod A, Shaefer G, Wang C, Ecker R, Dorffner G, Ellinger I, Investigating and Exploiting Image Resolution for Transfer Learning-based Skin Lesion Classification, 2020, https://arxiv.org/abs/2006.14715v1

[33] Haenssle HA, et al. Man against machine: diagnostic performance of a deep learning convolutional neural network for dermoscopic melanoma recognition in comparison to 58 dermatologists, Annals of Oncology 29.8, 2018, pp.1836-1842.

[34] Szegedy C,Ioffe S, Vanhoucke V, Alemi A, Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, 2016, arXiv:1602.07261 [cs.CV]

[35] https://en.wikipedia.org/wiki/Sensitivity_and_specificity, Fecha de acceso: 19/12/2021

[36] https://medium.com/gumgum-tech/handling-class-imbalance-by-introducing-sample-weighting-in-the-loss-function-3bdebd8203b4, Fecha de acceso: 20/10/2021

[37] Goodfellow, I., Pouget-Abadie, J, Mirza, M, Xu, B, Warde-Farley, D, Ozair S, Courville A, Bengio Y, Generative adversarial networks, 2014, arXiv:1406.2661 [stat.ML]

[38] Han Zhang H, Goodfellow I, Metaxas D, Odena A, Self-Attention Generative Adversarial Networks, 2019, arXiv:1805.08318

[39] https://wandb.ai/site, Fecha de acceso: 10/11/2021

[40] https://en.wikipedia.org/wiki/Fr%C3%A9chet_distance, Fecha de acceso:19/12/2021

[41] Brinker et al., Skin Cancer Classification Using Convolutional Neural Networks:Systematic Review, J Med Internet Res 2018,vol. 20, iss. 10, e11936, pp 2

[42] Ara A, Deserno TM, A systematic review of automated melanoma detection in dermatoscopic images and its ground truth data, Proc SPIE Int Soc Opt Eng 2012, doi: 10.1117/12.912389

[43] Fabbrocini G, De Vita V, Pastore F, D'Arco V, Mazzella C, Annunziata MC, et al. Teledermatology: From prevention to diagnosis of nonmelanoma and melanoma skin cancer. Int J Telemed Appl, 2011, doi: 10.1155/2011/125762]

[44] https://www.kaggle.com/, Fecha de acceso: 10/10/2021

[45] https://paperswithcode.com/, Fecha de acceso: 22/12/2021

[46] Baur C, Albarqouni S, Navab N, Generating Highly Realistic Images of Skin Lesions with GANs, 2018, arXiv:1809.01410 [cs.CV]

[47] Skandarani Y, Jodoin PM, Lalande A , GANs for Medical Image Synthesis: An Empirical Study, 2021, arXiv:2105.05318v2 [eess.IV]

[48] Kang M, Shim, W, Cho M, Park J, Rebooting ACGAN: Auxiliary Classifier GANs with Stable Training, 2021, arXiv:2111.01118 [cs.CV]

[49] https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/, Fecha de acceso:21/12/2021

[50] Wang J, Perez L, The Effectiveness of Data Augmentation in Image Classification using Deep Learning, 2017, arXiv:1712.04621 [cs.CV]

8. Anexos

8.1 Código fuente

Los distintos modelos que se han implementado se pueden revisar en el repositorio de Github: https://github.com/agusbarba/TFM_shared

El trabajo consta de dos sistemas diferenciados. Por un lado, la red neural que realiza la tarea de clasificación principal, y por otro la red tipo GAN con la que se han generado las imágenes complementarias para mejorar el desempeño del clasificador.

Se han utilizado libretas Jupyter de Google Colab para la realización de ambas.

Código para entrenamiento de clasificador

A continuación, se presenta el código utilizado para los entrenamientos del clasificador. También puede ser consultado en:

https://github.com/agusbarba/TFM_shared/blob/main/TFM.ipynb

Instrucciones

Para lanzar un entrenamiento es necesario lanzar los distintos pasos que lo componen:

0. Conexión con Google Drive (opcional)
1. Instalación de bibliotecas
2. Importación de bibliotecas
3. Se fijan las semillas de los procesos aleatorios
4. Se definen los parámetros del experimento que se va a lanzar
5. Definición de los dataloaders.

   En función del tipo de experimento se utilizará una celda diferente:

   - 5.0 Dataloader dinámico
   - 5.1 Dataloader standard
   - 5.2 Dataloader para carga de datos de la GAN
   - 5.3 Dataloaders reentrenamiento GANS

6. Definición de la red neural (efficentnet)
7. Definición de la función de pérdida

   En función del tipo de experimento se utilizará una celda diferente:

   - 7.0 Standard
   - 7.1 Usando datos de la GAN

8. Definición de las métricas a emplear
9. Definición del modelo de test
10. Definición del modulo de datos:

    En función del tipo de experimento se utilizará una celda diferente:

    - 10.0 Módulo de datos standard
    - 10.1 Módulo de datos dinámico
    - 10.2 Módulo de datos dinámico para 50 epochs
    - 10.3 Módulo de datos dinámico para datos de la GAN
    - 10.4 Modulo de datos dinámico para reentrenamiento GAN

11. Creación del modelo
12. Entrenamiento

0. Drive

En esta sección se realiza la conexión con Google Drive, de donde se leerán los conjuntos de datos y se escribirán los resultados

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

1. Dependencias

En esta sección se instalan y actualizan todas las bibliotecas que se van a utilizar

In [ ]:
```
!pip install torch torchvision
!pip install torchmetrics
!pip install pytorch-lightning
!pip install --upgrade timm
!pip install timm_vis
!nvidia-smi
```

2. Bibliotecas

Sección donde se importan las bibliotecas que se van a usar

In [ ]:
```
import pytorch_lightning as pl
import torch
import torchmetrics
import os
import cv2
import gc
import PIL
import math
import shutil
import pandas as pd
import seaborn as sn
import torch.utils.data
import numpy as np
import timm
import sklearn
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt

from timm_vis.methods import *
from torchvision import transforms, models, datasets
from sklearn.preprocessing import LabelEncoder
from torch.utils.data import SubsetRandomSampler
from torchmetrics import Metric
from collections import Counter
from pytorch_lightning.loggers import TensorBoardLogger, CSVLogger
from torchmetrics.functional import accuracy
from sklearn.metrics import average_precision_score
from sklearn.preprocessing import label_binarize
from tqdm import tqdm
from torchmetrics import ConfusionMatrix
from sklearn.metrics import classification_report
```

3. Reproducibilidad

Se definen los distintos seeds para controlar la reproducibilidad del experimento

In [ ]:
```
def seed_everything(seed):
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = True

seed_everything(123)
```

4. Configuración Experimento

En este apartado definimos el tipo de experimento que se quiere realizar

In [ ]:
```
#MODE
VERBOSE_MODE = False

#TYPE
PRETRAINED = True
FREEZE_MODEL = False
GAN_MODE = False

#SAMPLING
WEIGHTED_SAMPLER = False
SHOW_WEIGHTED_SELECTION = False
OVERSAMPLE = False
NUMBER_OF_SAMPLES = 35000
DYNAMIC_SAMPLING = False

#TRAINING
EPOCHS = 16

#LOSS
CALCULATE_CLASS_WEIGHTS = True
USE_WEIGTHED_CROSS_ENTROPY_LOSS = True

if DYNAMIC_SAMPLING:
  CALCULATE_CLASS_WEIGHTS = False
  USE_WEIGTHED_CROSS_ENTROPY_LOSS = False

#DATASETS
DATASET_NAME =          '224'
MAIN_DATA_DIR =         '/content/drive/My Drive/Master/TFM/'
LOGS_FOLDER =           '/content/drive/My Drive/Master/TFM/tb_logs/'
MODELS_FOLDER =         '/content/drive/My Drive/Master/TFM/models/'
CHECKPOINTS_FOLDER =    '/content/drive/My Drive/Master/TFM/models/checkpoints/'
TRAIN_DIR = MAIN_DATA_DIR + DATASET_NAME

#TRAINING - TEST DISTRIBUTION
BATCH_SIZE = 80
TEST_SIZE = 0.15
VALID_SIZE = 0.15
```

```
#PARAMETERS
AVAIL_GPUS = min(1, torch.cuda.device_count())
GPU_MODE = True if AVAIL_GPUS else False
TRAIN_SIZE = 1 - TEST_SIZE - VALID_SIZE
NUM_WORKERS = os.cpu_count()
LR = 3e-4

#LOG
COMMENT = ""
if WEIGHTED_SAMPLER:
  COMMENT = COMMENT + "WS-"

if OVERSAMPLE:
  COMMENT = COMMENT + "OS-"

if USE_WEIGTHED_CROSS_ENTROPY_LOSS:
  COMMENT = COMMENT + "WL-"

if PRETRAINED:
  COMMENT = COMMENT + "TR-"

if DYNAMIC_SAMPLING:
  COMMENT = COMMENT + "DS-"
if GAN_MODE:
  COMMENT = COMMENT + "GAN-"

if FREEZE_MODEL:
  COMMENT = COMMENT + "FM"


NAME_OF_DATASET = DATASET_NAME + "_" + str(BATCH_SIZE) + "_" + str(LR) + "(" + str(int(TRAIN_SIZE * 100))
+ ":" + str(int(VALID_SIZE * 100)) + ":" + str(int(TEST_SIZE * 100)) + ")_" + COMMENT

#CONSOLE
print("MODE GPU:", GPU_MODE)
print("AVAIL_GPUS" ,AVAIL_GPUS)
print("NAME_OF_DATASET:" + NAME_OF_DATASET)
```

5.0 Dynamic Data Loaders

In [ ]:
```
trainDir = TRAIN_DIR
test_size = TEST_SIZE
valid_size = VALID_SIZE
batch_size = BATCH_SIZE
num_workers = NUM_WORKERS

print('TRAINING DIR:',  TRAIN_DIR)

np.seterr(divide='ignore', invalid='ignore')

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
])

def normalizeWeights(inputTensor):
  mean = torch.mean(inputTensor)
  normalization_factor = 1/mean
  output = inputTensor * normalization_factor
  return output

def getSampleWeights():

  length = len(imageDataset)
  indexes = list(range(length))
  classes = [imageDataset.targets[i] for i in train_idx]
  print("Classes:" , classes)
  class_sample_count = Counter(classes)
  print(class_sample_count)
  print("Number of classes: " , class_sample_count)
  class_count = np.array([class_sample_count[0],class_sample_count[1],
                          class_sample_count[2],class_sample_count[3],
                          class_sample_count[4],class_sample_count[5],
                          class_sample_count[6]])
  print("Class count ", class_count)
  weight = 1. / class_count
  print("weight", weight)
  samples_weight = np.array([weight[t] for t in classes])
  samples_weight=torch.from_numpy(samples_weight)
  print("samples_weight", samples_weight)
  return samples_weight
```

```
#GET IMAGES
imageDataset = datasets.ImageFolder(trainDir, transform = transform)
length = len(imageDataset)

print("Dataset Length: ",  length)

#CREATE SPLITS
indexes = list(range(length))
np.random.seed(2)
np.random.shuffle(indexes)
split1 = int(np.floor(valid_size * length))
split2 =int(np.floor((test_size + valid_size) * length))
train_idx, test_idx, valid_idx = indexes[split2:], indexes[:split1], indexes[split1:split2]

print(train_idx)

train_sampler = SubsetRandomSampler(train_idx)
test_sampler = SubsetRandomSampler(test_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

trainDataset = torch.utils.data.Subset(imageDataset, train_idx)

getSampleWeights()
```

5.1 Data Loaders

In [ ]:
```
trainDir = TRAIN_DIR
test_size = TEST_SIZE
valid_size = VALID_SIZE
batch_size = BATCH_SIZE
num_workers = NUM_WORKERS

np.seterr(divide='ignore', invalid='ignore')

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
])

def createWeightedSampler():

  length = len(imageDataset)
  indexes = list(range(length))
  classes = [imageDataset.targets[i] for i in train_idx]
  class_sample_count = Counter(classes)
  print(class_sample_count)
  class_count = np.array([class_sample_count[0],class_sample_count[1],
                          class_sample_count[2],class_sample_count[3],
                          class_sample_count[4],class_sample_count[5],
                          class_sample_count[6]])
  weight = 1. / class_count
  samples_weight = np.array([weight[t] for t in classes])
  samples_weight=torch.from_numpy(samples_weight)
  print("samples_weight", samples_weight)
  samples_weight_length = len(samples_weight)

  if OVERSAMPLE:
    samples_weight_length = NUMBER_OF_SAMPLES  #Oversample to this number


  weighted_train_sampler = torch.utils.data.sampler.WeightedRandomSampler(samples_weight,
                                                                          samples_weight_length,
                                                                          replacement=True)
  return weighted_train_sampler
```

```
#GET IMAGES
imageDataset = datasets.ImageFolder(trainDir, transform = transform)
length = len(imageDataset)

#CREATE SPLITS
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
split1 = int(np.floor(valid_size * length))
split2 =int(np.floor((test_size + valid_size) * length))
train_idx, test_idx, valid_idx = indexes[split2:], indexes[:split1], indexes[split1:split2]

if VERBOSE_MODE:
  print("Data set Info:**********************************")
  print("Train indexes")
  print(train_idx)
  print("Validation indexes")
  print(valid_idx)
  print("Test indexes")
  print(test_idx)

  print("Train images (20)")
  for i, imageIndex in enumerate(train_idx):
    if i == 20:
      break
    print(imageDataset.imgs[imageIndex])

  print("Validation images (20)")
  for i, imageIndex in enumerate(valid_idx):
    if i == 20:
      break
    print(imageDataset.imgs[imageIndex])

  print("Test images (20)")
  for i, imageIndex in enumerate(test_idx):
    if i == 20:
      break
    print(imageDataset.imgs[imageIndex])

  print("************************************************")

train_sampler = SubsetRandomSampler(train_idx)
test_sampler = SubsetRandomSampler(test_idx)
valid_sampler = SubsetRandomSampler(valid_idx)
```

```
#CREATE LOADERS

if WEIGHTED_SAMPLER:
  testDataset = torch.utils.data.Subset(imageDataset, train_idx)
  train_loader = torch.utils.data.DataLoader(testDataset, batch_size=batch_size,
                                             drop_last= True,
                                             sampler=createWeightedSampler(),
                                             pin_memory=GPU_MODE)
  print("Train Loader created successfully")
else:
  train_loader = torch.utils.data.DataLoader(imageDataset, batch_size=batch_size,
                                             drop_last= True,
                                             sampler=train_sampler,
                                             pin_memory=GPU_MODE)

valid_loader = torch.utils.data.DataLoader(imageDataset,
                                           batch_size=batch_size,
                                           drop_last= True,
                                           sampler=valid_sampler,
                                           shuffle=False,
                                           num_workers=num_workers,
                                           pin_memory=GPU_MODE)
print("Validation Loader created successfully")
test_loader = torch.utils.data.DataLoader(imageDataset,
                                          batch_size=batch_size,
                                          drop_last= True,
                                          sampler=test_sampler,
                                          shuffle=False,
                                          num_workers=num_workers,
                                          pin_memory=GPU_MODE)
print("Test Loader created successfully")
#SHOW WEIGHTS
```

```
if SHOW_WEIGHTED_SELECTION & WEIGHTED_SAMPLER:
  print("Train Set distribution sample")
  for i, (image, target) in enumerate(train_loader):
      if i == 5:
        break
      print("batch index {}, 0/1/2/3/4/5/6: {}/{}/{}/{}/{}/{}/{}".format(
          i,
          np.count_nonzero(target.numpy() == 0),
          np.count_nonzero(target.numpy() == 1),
          np.count_nonzero(target.numpy() == 2),
          np.count_nonzero(target.numpy() == 3),
          np.count_nonzero(target.numpy() == 4),
          np.count_nonzero(target.numpy() == 5),
          np.count_nonzero(target.numpy() == 6)))

  print("Validation Set distribution sample:")
  for i, (image, target) in enumerate(valid_loader):
      if i == 5:
        break
      print("batch index {}, 0/1/2/3/4/5/6: {}/{}/{}/{}/{}/{}/{}".format(
          i,
          np.count_nonzero(target.numpy() == 0),
          np.count_nonzero(target.numpy() == 1),
          np.count_nonzero(target.numpy() == 2),
          np.count_nonzero(target.numpy() == 3),
          np.count_nonzero(target.numpy() == 4),
          np.count_nonzero(target.numpy() == 5),
          np.count_nonzero(target.numpy() == 6)))

if CALCULATE_CLASS_WEIGHTS:

  c0 = 0
  c1 = 0
  c2 = 0
  c3 = 0
  c4 = 0
  c5 = 0
  c6 = 0
```

```
  for i, (image, target) in enumerate(train_loader):
    t = target.numpy()
    print(i)
    c0 += np.count_nonzero(t == 0)
    c1 += np.count_nonzero(t == 1)
    c2 += np.count_nonzero(t == 2)
    c3 += np.count_nonzero(t == 3)
    c4 += np.count_nonzero(t == 4)
    c5 += np.count_nonzero(t == 5)
    c6 += np.count_nonzero(t == 6)

  total_samples = c0+c1+c2+c3+c4+c5+c6
  train_classes = {0: c0, 1:c1, 2:c2, 3:c3, 4:c4, 5:c5, 6:c6}

  train_class_weights ={0:total_samples/c0, 1:total_samples/c1,
                        2:total_samples/c2, 3:total_samples/c3,
                        4:total_samples/c4, 5:total_samples/c5,
                        6:total_samples/c6}
  print("Weights for the experiment")
  print(train_class_weights)

else:
   #weights from oversampled (35000) dataset
   #train_class_weights = {0: 6.9489167163585766, 1: 6.797588955862337, 2: 7.059773828756058,
   #3: 7.180119120969398, 4: 7.075490791337786, 5: 6.996197718631179, 6: 6.954445991645116}
   #weights from standard dataset
   train_class_weights = {0: 30.12987012987013, 1: 19.885714285714286, 2: 9.121887287024903,
                          3: 91.57894736842105, 4: 9.25531914893617, 5: 1.4852752880921896,
                          6: 68.23529411764706}

if CALCULATE_CLASS_WEIGHTS:
  print("Train Set")
  print(train_classes)
  print(total_samples)

print("Validation Set")
print("Size: " , len(valid_idx))
classes = [imageDataset.targets[i] for i in valid_sampler.indices]
print(Counter(classes))

print("Test Set")
print("Size: " , len(test_idx))
classes = [imageDataset.targets[i] for i in test_sampler.indices]
print(Counter(classes))
print(train_loader)
```

5.2 DataLoaders GANS

In [ ]:

```
valid_Dir = '/content/drive/MyDrive/Master/TFM/224_fixed/224/valid'
test_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/224/test'

batch_size = BATCH_SIZE
num_workers = NUM_WORKERS

transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
])


validImageDataset = datasets.ImageFolder(valid_Dir, transform = transform)
testImageDataset = datasets.ImageFolder(test_Dir, transform = transform)

length = len(validImageDataset)
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
valid_idx = indexes

length = len(testImageDataset)
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
test_idx = indexes


test_sampler = SubsetRandomSampler(test_idx)
valid_sampler = SubsetRandomSampler(valid_idx)
valid_loader = torch.utils.data.DataLoader(validImageDataset,
                                           batch_size=batch_size,
                                           sampler=valid_sampler,
                                           drop_last= True,
                                           num_workers=num_workers,
                                           pin_memory=GPU_MODE,
                                           shuffle  = False)
test_loader  = torch.utils.data.DataLoader(testImageDataset,
                                           batch_size=batch_size,
                                           sampler=test_sampler,
                                           drop_last= True,
                                           num_workers=num_workers,
                                           pin_memory=GPU_MODE,
                                           shuffle = False)
```

5.3 Dataloaders reentrenamiento GANS

In [ ]:
```
train_Dir = '/content/drive/MyDrive/Master/TFM/224_fixed/224/train'
valid_Dir = '/content/drive/MyDrive/Master/TFM/224_fixed/224/valid'
test_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/224/test'

batch_size = BATCH_SIZE
num_workers = NUM_WORKERS

transform = transforms.Compose([
  transforms.ToTensor(),
  transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
])


trainImageDataset = datasets.ImageFolder(train_Dir, transform = transform)
validImageDataset = datasets.ImageFolder(valid_Dir, transform = transform)
testImageDataset = datasets.ImageFolder(test_Dir, transform = transform)

length = len(trainImageDataset)
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
train_idx = indexes


length = len(validImageDataset)
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
valid_idx = indexes

length = len(testImageDataset)
indexes = list(range(length))
np.random.seed(123)
np.random.shuffle(indexes)
test_idx = indexes

train_sampler = SubsetRandomSampler(train_idx)
test_sampler = SubsetRandomSampler(test_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(trainImageDataset,
                                           batch_size=batch_size,
                                           sampler=train_sampler,
                                           drop_last= True,
                                           num_workers=num_workers,
                                           pin_memory=GPU_MODE)
valid_loader = torch.utils.data.DataLoader(validImageDataset,
                                           batch_size=batch_size,
                                           sampler=valid_sampler,
                                           drop_last= True,
                                           num_workers=num_workers,
                                           pin_memory=GPU_MODE,
                                           shuffle  = False)
test_loader  = torch.utils.data.DataLoader(testImageDataset,
                                           batch_size=batch_size,
                                           sampler=test_sampler,
                                           drop_last= True,
                                           num_workers=num_workers,
                                           pin_memory=GPU_MODE,
                                           shuffle = False)
```

6. Creación de la red

In [ ]:
```
def createModel():
  model = timm.create_model('tf_efficientnet_b0', pretrained = PRETRAINED, num_classes=7)

  if FREEZE_MODEL:
    count = 0;
    print("Freezing model")
    for param in model.parameters():
      count = count + 1
      if count < 212: #212 to freeze the last two #211 last 3
        param.requires_grad = False

    for param in model.parameters():
      print(param)

  return model


test = createModel()
```

7.0 Función de loss

In [ ]:
```
def normalizeWeights(inputTensor):
  mean = torch.mean(inputTensor)
  normalization_factor = 1/mean
  output = inputTensor * normalization_factor
  return output

print("Using: ")
if USE_WEIGTHED_CROSS_ENTROPY_LOSS:

    classes = [imageDataset.targets[i] for i in train_sampler.indices]
    class_tensor = torch.FloatTensor(classes)

    validation_classes = [imageDataset.targets[i] for i in valid_sampler.indices]
    validation_class_tensor = torch.FloatTensor(validation_classes)

    test_classes = [imageDataset.targets[i] for i in test_sampler.indices]
    test_class_tensor = torch.FloatTensor(test_classes)

    if WEIGHTED_SAMPLER:
       class_weights = torch.tensor([0.148, 0.144,  0.142, 0.141, 0.140, 0.141 , 0.145])
       print("oversampled train class weights", class_weights)
    else:
      class_weights = sklearn.utils.class_weight.compute_class_weight('balanced',
                                                                      classes =np.unique(class_tensor),
                                                                      y = class_tensor.numpy())

      class_weights=torch.tensor(class_weights,dtype=torch.float)
      class_weights = normalizeWeights(class_weights)
      print("train class_weights", class_weights)


    validation_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',
                                                                    classes = np.unique(validation_class_tensor),
                                                                    y = validation_class_tensor.numpy())
    validation_class_weights=torch.tensor(validation_class_weights,dtype=torch.float)
    validation_class_weights = normalizeWeights(validation_class_weights)
    print("validation_class_weights", validation_class_weights)
    test_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',
                                                                     classes =np.unique(test_class_tensor),
                                                                     y = test_class_tensor.numpy())
    test_class_weights=torch.tensor(test_class_weights,dtype=torch.float)
    test_class_weights = normalizeWeights(test_class_weights)
    print("test_class_weights", test_class_weights)

    if GPU_MODE:
      trainLossFunction = nn.CrossEntropyLoss(class_weights.cuda())
      validationLossFunction = nn.CrossEntropyLoss(validation_class_weights.cuda())
      testLossFunction = nn.CrossEntropyLoss(test_class_weights.cuda())
    else:
      trainLossFunction = nn.CrossEntropyLoss(class_weights)
      validationLossFunction = nn.CrossEntropyLoss(validation_class_weights)
      testLossFunction = nn.CrossEntropyLoss(test_class_weights)
    print("Using Weighted Loss Function")

else:
  print("nn.CrossEntropyLoss()")
  trainLossFunction = nn.CrossEntropyLoss()
  validationLossFunction = nn.CrossEntropyLoss()
  testLossFunction = nn.CrossEntropyLoss()
```

7.1 Función de Loss GAN

In [ ]:
```
def normalizeWeights(inputTensor):
  mean = torch.mean(inputTensor)
  normalization_factor = 1/mean
  output = inputTensor * normalization_factor
  return output


if USE_WEIGTHED_CROSS_ENTROPY_LOSS:

    classes = [trainImageDataset.targets[i] for i in train_sampler.indices]
    class_tensor = torch.FloatTensor(classes)

    validation_classes = [validImageDataset.targets[i] for i in valid_sampler.indices]
    validation_class_tensor = torch.FloatTensor(validation_classes)

    test_classes = [testImageDataset.targets[i] for i in test_sampler.indices]
    test_class_tensor = torch.FloatTensor(test_classes)

    if WEIGHTED_SAMPLER:
       class_weights = torch.tensor([0.148, 0.144,  0.142, 0.141, 0.140, 0.141 , 0.145])
       print("oversampled train class weights", class_weights)
    else:
      class_weights = sklearn.utils.class_weight.compute_class_weight('balanced',
                                                                      classes =np.unique(class_tensor),
                                                                      y = class_tensor.numpy())
      class_weights=torch.tensor(class_weights,dtype=torch.float)
      class_weights = normalizeWeights(class_weights)
      print("train class_weights", class_weights)


    validation_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',
                                                                    classes = np.unique(validation_class_tensor),
                                                                    y = validation_class_tensor.numpy())
    validation_class_weights=torch.tensor(validation_class_weights,dtype=torch.float)
    validation_class_weights = normalizeWeights(validation_class_weights)
    print("validation_class_weights", validation_class_weights)
    test_class_weights=sklearn.utils.class_weight.compute_class_weight('balanced',
                                                                       classes =np.unique(test_class_tensor),
                                                                       y = test_class_tensor.numpy())
    test_class_weights=torch.tensor(test_class_weights,dtype=torch.float)
    test_class_weights = normalizeWeights(test_class_weights)
    print("test_class_weights", test_class_weights)
```

```
    if GPU_MODE:
      trainLossFunction = nn.CrossEntropyLoss(class_weights.cuda())
      validationLossFunction = nn.CrossEntropyLoss(validation_class_weights.cuda())
      testLossFunction = nn.CrossEntropyLoss(test_class_weights.cuda())
    else:
      trainLossFunction = nn.CrossEntropyLoss(class_weights)
      validationLossFunction = nn.CrossEntropyLoss(validation_class_weights)
      testLossFunction = nn.CrossEntropyLoss(test_class_weights)
    print("Using Weighted Loss Function")

else:
  print("nn.CrossEntropyLoss()")
  trainLossFunction = nn.CrossEntropyLoss()
  validationLossFunction = nn.CrossEntropyLoss()
  testLossFunction = nn.CrossEntropyLoss()
```

8. Métricas

In [ ]:
```
metric = torchmetrics.F1()
confusionMatrix = ConfusionMatrix(num_classes=7,  multilabel=False)
```

9. Definición del Modelo de Test

In [ ]:
```
class TFMModel(pl.LightningModule):

    def __init__(self, lr = LR, input_size=None):
        super().__init__()
        self.save_hyperparameters()
        self.model = createModel()
        self.train_acc = metric
        self.valid_acc = metric

    def forward(self, x):
        return self.model(x)

    def training_step(self, batch, batch_idx):
        #visualize_filters(self.model)
        inputs, labels = batch
        y_pred = self(inputs)
        loss = trainLossFunction(y_pred, labels)
        #if VERBOSE_MODE:
          #print("**************TRAIN LOG******************")
          #print("LABELS ", labels)
          #print("Y pred ", y_pred)
          #print("LOSS:", loss)
        self.log('train_loss', loss)
        ap = self.train_acc(y_pred, labels)
        self.log('train_acc', ap)
        return loss
```

```
def evaluate(self, batch, stage=None):
    inputs, labels = batch
    y_pred = self(inputs)
    if stage == "val":
      loss = validationLossFunction(y_pred, labels)
    else:
      loss = testLossFunction(y_pred, labels)
    if VERBOSE_MODE:
      print("**************",stage, "********************")
      print("LABELS ", labels)
      print("Y pred ", y_pred)
      print("Loss:", loss)
    self.train_acc(y_pred, labels)
    if (stage):
      self.log(f"{stage}_loss", loss, prog_bar = True)
      self.log(f"{stage}_acc", self.train_acc, prog_bar = True)
    return {'loss': loss, 'preds': y_pred, 'target': labels}

def validation_step(self, batch, batch_idx):
    outputs = self.evaluate(batch, "val")
    return outputs
```

```
def validation_epoch_end(self, outputs):
    preds = torch.cat([tmp['preds'] for tmp in outputs])
    targets = torch.cat([tmp['target'] for tmp in outputs])
    confusion_matrix = confusionMatrix(preds.cpu(), targets.cpu())
    print("Validation")
    print(confusion_matrix)
    conf_mat = confusion_matrix.detach().cpu().numpy().astype(np.int)
    df_cm = pd.DataFrame(
      conf_mat,
      index=np.arange(7),
      columns=np.arange(7))
    sn.heatmap(df_cm/np.sum(df_cm),
               annot=True, xticklabels = ['akiec','bcc','bkl','df','mel','nv','vasc'],
               yticklabels = ['akiec','bcc','bkl','df','mel','nv','vasc'],
               fmt='.2%', cbar_kws={'format': '%.0f%%', 'ticks': [0, 100]},linewidths=.5,  cmap="YlGnBu")
    plt.show()

    target_names = ['akiec','bcc','bkl','df','mel','nv','vasc']
    if GPU_MODE:
      predsCPU = preds.cpu()
      targetCPU = targets.cpu()
    else:
      predsCPU = preds
      targetCPU = targets
    binary_ground_truth = label_binarize(targetCPU, classes=[0, 1, 2, 3, 4, 5, 6])
    predictions = predsCPU.detach().numpy()
    predictions = np.argmax(predictions, axis=1)
    if VERBOSE_MODE:
      print(targetCPU)
      print(predictions)
    try:
      print(classification_report(targetCPU, predictions, target_names=target_names))
    except:
      print("Problem printing classification report")


    CURRENT_EPOCH = self.current_epoch
    print("CURRENT_EPOCH", CURRENT_EPOCH)
```

```
    def test_epoch_end(self, outputs):
        preds = torch.cat([tmp['preds'] for tmp in outputs])
        targets = torch.cat([tmp['target'] for tmp in outputs])
        confusion_matrix = confusionMatrix(preds.cpu(), targets.cpu())
        conf_mat = confusion_matrix.detach().cpu().numpy().astype(np.int)
        print("Test")
        df_cm = pd.DataFrame(
          conf_mat,
          index=np.arange(7),
          columns=np.arange(7))
        sn.heatmap(df_cm/np.sum(df_cm), annot=True,
                   xticklabels = ['akiec','bcc','bkl','df','mel','nv','vasc'],
                   yticklabels = ['akiec','bcc','bkl','df','mel','nv','vasc'],
                   fmt='.2%', cbar_kws={'format': '%.0f%%', 'ticks': [0, 100]},linewidths=.5,  cmap="YlGnBu")
        plt.figure()
        plt.show()

        target_names = ['akiec','bcc','bkl','df','mel','nv','vasc']

        if GPU_MODE:
          predsCPU = preds.cpu()
          targetCPU = targets.cpu()
        else:
          predsCPU = preds
          targetCPU = targets

        binary_ground_truth = label_binarize(targetCPU, classes=[0, 1, 2, 3, 4, 5, 6])
        predictions = predsCPU.detach().numpy()
        predictions = np.argmax(predictions, axis=1)

        if VERBOSE_MODE:
          print(targetCPU)
          print(predictions)

        try:
          print(classification_report(targetCPU, predictions, target_names=target_names))
        except:
          print("Problem printing classification report")

    def test_step(self, batch, batch_idx):
        outputs = self.evaluate(batch, "test")
        return outputs

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.hparams['lr'])
        return optimizer
```

10.0 Módulo de datos

In [ ]:
```
class Ham10000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):

      if DYNAMIC_SAMPLING:
        print("Using dynamic sampling")
        train_sampler = createDynamicWeightedSampler(0)
        train_loader = torch.utils.data.DataLoader(trainDataset, batch_size = batch_size,
                                                   drop_last = True, sampler = train_sampler,
                                                   pin_memory = GPU_MODE)
      else:
        trainDataset = torch.utils.data.Subset(imageDataset, train_idx)
        train_loader = torch.utils.data.DataLoader(trainDataset, batch_size=batch_size, drop_last= True,
                                                   sampler=createWeightedSampler(), pin_memory=GPU_MODE)

      return  train_loader
    def val_dataloader(self):
      return valid_loader

    def test_dataloader(self):
      return test_loader


ham10000_dm = Ham10000DataModule(batch_size=BATCH_SIZE)
```

10.1 Módulo de datos dinámico

In [ ]:
```
class Ham10000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):

      samples_weight = getSampleWeights()

      print("self.trainer.current_epoch", self.trainer.current_epoch)
      if self.trainer.current_epoch < 2:
       samples_weight_length = 35000  #Oversample to this number
      elif  2 <= self.trainer.current_epoch <4:
        samples_weight_length = 25000
      elif 4 <= self.trainer.current_epoch < 6:
       samples_weight_length = 15000
      elif 6 <= self.trainer.current_epoch < 8:
        samples_weight_length = 12000
      else:
        samples_weight_length = len(samples_weight)

      print("samples_weight_length:" ,  samples_weight_length)

      weighted_train_sampler = torch.utils.data.sampler.WeightedRandomSampler(samples_weight,
                                                                               samples_weight_length,
                                                                               replacement=True)

      train_loader = torch.utils.data.DataLoader(trainDataset, batch_size = batch_size, drop_last = True,
                                                 sampler = weighted_train_sampler, pin_memory = GPU_MODE)
      return  train_loader
    def val_dataloader(self):
      return torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, shuffle=False, drop_last= True,
                                         sampler=valid_sampler, num_workers=num_workers, pin_memory=GPU_MODE)

    def test_dataloader(self):
      return torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, shuffle=False, drop_last= True,
                                         sampler=test_sampler, num_workers=num_workers, pin_memory=GPU_MODE)


ham10000_dm = Ham10000DataModule(batch_size=BATCH_SIZE)
```

10.2 Módulo de datos dinámico 50 epoch

In [ ]:
```
class Ham10000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):

      samples_weight = getSampleWeights()

      print("self.trainer.current_epoch", self.trainer.current_epoch)
      if self.trainer.current_epoch < 4:
       samples_weight_length = 35000  #Oversample to this number
      elif  4 <= self.trainer.current_epoch <8:
        samples_weight_length = 25000
      elif 8 <= self.trainer.current_epoch < 12:
       samples_weight_length = 15000
      elif 12 <= self.trainer.current_epoch < 18:
        samples_weight_length = 12000
      else:
        samples_weight_length = len(samples_weight)

      print("samples_weight_length:" ,  samples_weight_length)

      weighted_train_sampler = torch.utils.data.sampler.WeightedRandomSampler(samples_weight,
                                                                               samples_weight_length,
                                                                               replacement=True)

      train_loader = torch.utils.data.DataLoader(trainDataset, batch_size = batch_size, drop_last = True,
                                                 sampler = weighted_train_sampler, pin_memory = GPU_MODE)
      return  train_loader
    def val_dataloader(self):
      return torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, shuffle=False, drop_last= True,
                                         sampler=valid_sampler, num_workers=num_workers, pin_memory=GPU_MODE)

    def test_dataloader(self):
      return torch.utils.data.DataLoader(imageDataset, batch_size=batch_size, shuffle=False, drop_last= True,
                                         sampler=test_sampler, num_workers=num_workers, pin_memory=GPU_MODE)


ham10000_dm = Ham10000DataModule(batch_size=BATCH_SIZE)
```

10.3 Módulo de datos dinámico GAN

In [ ]:
```
class Ham10000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):

      transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.7649, 0.5422, 0.5683), (0.1376, 0.1590, 0.1754))
      ])

      print("self.trainer.current_epoch", self.trainer.current_epoch)
      if self.trainer.current_epoch < 2:
       train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/4700'
       print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/4700")
      elif  2 <= self.trainer.current_epoch < 4:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/4000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/4000")
      elif 4 <= self.trainer.current_epoch < 6:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/3000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/3000")
      elif 6 <= self.trainer.current_epoch < 8:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/2000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/2000")
      elif 8 <= self.trainer.current_epoch < 10:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/1000'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/1000")
      elif 10 <= self.trainer.current_epoch < 12:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/500'
        print("Using training dir: " , "/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/500")
      else:
        train_Dir  = '/content/drive/MyDrive/Master/TFM/224_fixed/dopped_33000/train'
        print("Using original training dir:")

      trainImageDataset = datasets.ImageFolder(train_Dir, transform = transform)
      length = len(trainImageDataset)
      indexes = list(range(length))
      np.random.seed(123)
      np.random.shuffle(indexes)
      train_idx = indexes
```

```
      train_sampler = SubsetRandomSampler(train_idx)
      train_loader = torch.utils.data.DataLoader(trainImageDataset, batch_size=batch_size,
                                                 sampler=train_sampler, pin_memory=GPU_MODE)
      return  train_loader


    def val_dataloader(self):
      return valid_loader
    def test_dataloader(self):
      return test_loader


ham10000_dm = Ham10000DataModule(batch_size=BATCH_SIZE)
```

10.4 Modulo de datos dinámico para reentrenamiento GAN

In [ ]:
```
class Ham10000DataModule(pl.LightningDataModule):

    def __init__(self, batch_size: int = BATCH_SIZE):
        super().__init__()

    def train_dataloader(self):
      return train_loader
    def val_dataloader(self):
      return valid_loader
    def test_dataloader(self):
      return test_loader


ham10000_dm = Ham10000DataModule(batch_size=BATCH_SIZE)
```

11. Crea Modelo

In [ ]:
```
model = TFMModel(LR)
```

12. Entrenamiento

In [ ]:
```
from time import strftime

TIME = strftime("%Y-%m-%d %H:%M:%S")
name = NAME_OF_DATASET + "-" + TIME
EPOCHS = 10
print("Training dataset ", name)
print("EPOCHS: " , EPOCHS)
print("LR: ", LR)
tb_logger = TensorBoardLogger(LOGS_FOLDER, name = name)
csv_logger = CSVLogger(LOGS_FOLDER, name = name)
gc.collect()
torch.cuda.empty_cache()
RESUME = False
CHECKPOINT = '/content/drive/MyDrive/Master/TFM/models/checkpoints/224_80_0.0003(70:15:15)_TR-GAN--2021-12-06 15

if GPU_MODE:
  if RESUME:
      trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      precision = 16,
                      logger=[tb_logger, csv_logger],
                      gpus=-1,
                      max_epochs=EPOCHS,
                      log_every_n_steps=10,
                      deterministic=True,
                      reload_dataloaders_every_n_epochs=2,
                      default_root_dir=CHECKPOINTS_FOLDER,
                      resume_from_checkpoint = CHECKPOINT)
  else:
    trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      precision = 16,
                      logger=[tb_logger, csv_logger],
                      gpus=-1,
                      max_epochs=EPOCHS,
                      log_every_n_steps=50,
                      reload_dataloaders_every_n_epochs=2,
                      deterministic=True,
                      default_root_dir=CHECKPOINTS_FOLDER)
```

```
else:
  if RESUME:
    trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      logger=[tb_logger, csv_logger],
                      gpus=0,
                      max_epochs=EPOCHS,
                      log_every_n_steps=50,
                      deterministic=True,
                      default_root_dir=CHECKPOINTS_FOLDER,
                      resume_from_checkpoint = CHECKPOINT)
  else:

    trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      logger=[tb_logger, csv_logger],
                      gpus=0,
                      max_epochs=EPOCHS,
                      log_every_n_steps=5,
                      reload_dataloaders_every_n_epochs=2,
                      deterministic=True,
                      default_root_dir=CHECKPOINTS_FOLDER)
  """
  trainer = pl.Trainer(progress_bar_refresh_rate = 10,
                      logger=[tb_logger, csv_logger],
                      gpus=0,
                      max_epochs=10,
                      log_every_n_steps=50,
                      resume_from_checkpoint=CHECKPOINTS_FOLDER + "224.ckpt")

#trainer = pl.Trainer(gpus=1,  logger = logger, resume_from_checkpoint = "path/to/ckpt/file/checkopoint.ckpt")
"""
trainer.fit(model, datamodule=ham10000_dm)
trainer.test(model, datamodule=ham10000_dm)

trainer.save_checkpoint(CHECKPOINTS_FOLDER + NAME_OF_DATASET + ".ckpt")
```

In [ ]:
```
%reload_ext tensorboard
%tensorboard --logdir /content/drive/MyDrive/Master/TFM/tb_logs
```

Código para entrenamiento de GAN

Para la creación y entrenamiento de la GAN, se ha utilizado como base la librería Pytorch StudioGAN.

Para su utilización con el conjunto de entrenamiento Ham 10000 ha habido que añadir algunas configuraciones. Esto se ha hecho en un *fork* del proyecto original que se puede consultar en: https://github.com/agusbarba/PyTorch-StudioGAN

Además, para su uso en el entorno de Google Colab ha habido que realizar algunas adaptaciones. El código completo de dichas adaptaciones y de los comandos de entrenamiento utilizados se lista a continuación y también puede consultarse en:

https://github.com/agusbarba/TFM_shared/blob/main/TestStudioGAN.ipynb

Drive

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

Dependencias

In [ ]:
```
!pip3 install tqdm ninja h5py kornia matplotlib pandas sklearn scipy seaborn wandb
!pip3 install PyYaml click requests pyspng imageio-ffmpeg prdc
!git clone https://github.com/agusbarba/PyTorch-StudioGAN.git
!pip3 install torch==1.10.0+cu111 torchvision==0.11.1+cu111 torchaudio==0.10.0+cu111
              -f https://download.pytorch.org/whl/cu111/torch_stable.html
!pip install --upgrade scikit-learn
!pip install -U PyYAML
```

Setup

In [ ]:
```
import torch
torch.cuda.empty_cache()
CUDA_VISIBLE_DEVICES = 0
!wandb login '6db71e761011c9987686e2f307a2825e505ca398'
```

Instrucciones

In [ ]:
```
!python3 PyTorch-StudioGAN/src/main.py
```

Entrenamiento

In [ ]:
```
!python3 PyTorch-StudioGAN/src/main.py -t -e
    -cfg "/content/PyTorch-StudioGAN/src/configs/HAM10000/My SAGAN-Size 128.yaml"
    -data "/content/drive/My Drive/Master/TFM/128" -save "/content/drive/My Drive/Master/TFM/results"
    --num_workers 2
```

Generar imágenes

In [ ]:
```
!python3 PyTorch-StudioGAN/src/main.py -s -best -resume_ct
-cfg "/content/PyTorch-StudioGAN/src/configs/HAM10000/My SAGAN-Size 128.yaml"
-ckpt "/content/drive/MyDrive/Master/TFM/results/checkpoints/Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10"
-data "/content/drive/MyDrive/Master/TFM/128" -save "/content/drive/MyDrive/Master/TFM/results/"
--num_workers 2
```

Continuar Entrenamiento

```
In [ ]:

!python3 PyTorch-StudioGAN/src/main.py -t -best -e
-cfg "/content/PyTorch-StudioGAN/src/configs/HAM10000/My SAGAN-Size 128.yaml"
-ckpt "/content/drive/MyDrive/Master/TFM/results/checkpoints/Ham10000-My SAGAN-Size 128-train-2021_11_19_22_23_10"
-data "/content/drive/MyDrive/Master/TFM/128"
-save "/content/drive/MyDrive/Master/TFM/results"
--num_workers 2
```

Para el entrenamiento de la red se ha utilizado el *dataset* original HAM 10000, redimensionado a 128x128. El la imagen inferior se puede ver el fichero de configuración utilizado.

master PyTorch-StudioGAN / src / configs / HAM10000 / **My SAGAN-Size 128.yaml** Go to file ...

agusbarba Update My SAGAN-Size 128.yaml Latest commit 34d1ba6 20 days ago History

1 contributor

22 lines (22 sloc) | 359 Bytes Raw Blame

```
DATA:
  name: "Ham10000"
  img_size: 128
  num_classes: 7
MODEL:
  g_cond_mtd: "cBN"
  d_cond_mtd: "PD"
  apply_g_sn: True
  apply_d_sn: True
  apply_attn: True
  attn_g_loc: [4]
  attn_d_loc: [1]
LOSS:
  adv_loss: "hinge"
OPTIMIZATION:
  batch_size: 20
  g_lr: 0.0001
  d_lr: 0.0004
  beta1: 0.0
  beta2: 0.999
  d_updates_per_step: 1
  total_steps: 1000000
```

Scripts auxiliares

Para la preparación de los distintos *datasets*, se ha utilizado el IDE de Python, Spyder 4.2.5 trabajando sobre una instalación de Anaconda. Los scripts utilizados han sido los siguientes:

Para realizar el estudio de las imágenes se ha utilizado:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Oct  9 19:35:21 2021

@author: abarsan
"""
import torch
from torch.utils.data import DataLoader
from torchvision import transforms, datasets

def obtain_mean_and_std(dataloader):

    channels_sum, channels_squared_sum, num_batches = 0, 0, 0

    print("Starting the calculation")

    for data, _ in dataloader:
        channels_sum += torch.mean(data, dim=[0,2,3])
        channels_squared_sum += torch.mean(data**2, dim=[0,2,3])
        num_batches += 1

    mean = channels_sum / num_batches
    std = (channels_squared_sum / num_batches - mean ** 2) ** 0.5
    return mean, std

DATASET_NAME = "224"
MAIN_DATA_DIR = 'C:/Users/Usuario/Google Drive/Master/TFM/'
TRAIN_DIR = MAIN_DATA_DIR + DATASET_NAME

transform = transforms.Compose([
    transforms.ToTensor(),
])

imageDataset = datasets.ImageFolder(TRAIN_DIR, transform = transform)
dataLoader    =    DataLoader(imageDataset,       batch_size=len(imageDataset),    shuffle=False,
num_workers=1)
obtain_mean_and_std(dataLoader)
```

Para reducir el tamaño de las imágenes, utilizando escalado biliniear y *croping* de los laterales se ha utilizado el siguiente script:

```
"""
Created on Sat Oct  9 12:44:15 2021
@author: abarsan
"""
from PIL import Image
from PIL import ImageOps
import os
basedir = "w:\\Ham10000\Images"
inside = os.listdir(basedir)
resolutions = [(60,45),(100, 75),(140, 115),(200, 150),(220, 165)]

resolutionsCenterCrop = [64,96,128,192,224]



for i in inside:
    fullPath = basedir + "\\" + i
    if os.path.isfile(fullPath):

        originalImage = Image.open(fullPath)
        border = (75, 0, 75, 0)
        originalImage = ImageOps.crop(originalImage, border)
        name = os.path.basename(fullPath)

        name, extension = os.path.splitext(name)


        for x in resolutionsCenterCrop:
            width = x
            outputDir = basedir + "\\" + str(width)
            im = originalImage.resize((width, width), Image.BILINEAR)
            im.save(outputDir + "\\" + name + ".png", 'PNG')
            print(name)
```

Para aumentar el tamaño desde las 128x128 que devuelve la GAN hasta el 224x224 que utiliza el clasificador se ha utilizado:

```
"""
Created on Sat Oct  9 12:44:15 2021
@author: abarsan
"""
from PIL import Image
import os
#baseDir =   "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/16000/fake3/"
#outputDir = "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/16000/fake3/bcc/"

baseDir =   "C:/Users/Usuario/Desktop/33000/"
outputDir = "C:/Users/Usuario/Desktop/33000_resized/"

inside = os.listdir(baseDir)
for classes in range(7):
    classFolder = baseDir + str(classes)
    outputClassFolder = outputDir + str(classes)
    print("Starting Folder", classFolder)
    print("Output Folder", outputClassFolder)

    inside = os.listdir(classFolder)
    for i in inside:
        print(i)
        fullPath = classFolder + "/" + i
        if os.path.isfile(fullPath):

            originalImage = Image.open(fullPath)
            width = 224
            height = 224
            im = originalImage.resize((width, height), Image.BILINEAR)
            im.save(outputClassFolder + "/" + i, 'PNG')
```

Finalmente, para generar los distintos *datasets* con distintos % de dopaje se ha utilizado el siguiente script.

```
"""
Created on Fri Nov 19 20:05:56 2021
@author: abarsan
"""
import os, shutil, random

origin = "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/33000_resized/"
destination = "C:/Users/Usuario/Google Drive/Master/TFM/224_fixed/dopped_33000/"

classes = ["akiec", "bcc", "bkl", "df", "mel", "vasc"]
lenghOfRealSets = {"akiec":229, "bcc":360, "bkl":769, "df": 81, "mel": 779, "nv": 4693, "vasc":
99}

finalSize = 3000
for classPath in classes:

    fromDir = origin + classPath
    toDir = destination + str(finalSize) + "/" + classPath

    files = os.listdir(fromDir)
    length = len(files)
    print("Available files ", length)
    print("Real Images for class" ,  classPath, " is ", lenghOfRealSets[classPath])
    testLength = finalSize - lenghOfRealSets[classPath]
    print("Adding ", testLength, " images")
    print("Creating  ", testLength, " files ")

    if testLength > 0:
        filenames = random.sample(files, testLength)
        for fname in filenames:
            srcpath = os.path.join(fromDir, fname)
            shutil.copy(srcpath, toDir)
            print("Copying file ", srcpath)
            print("to", toDir)
```