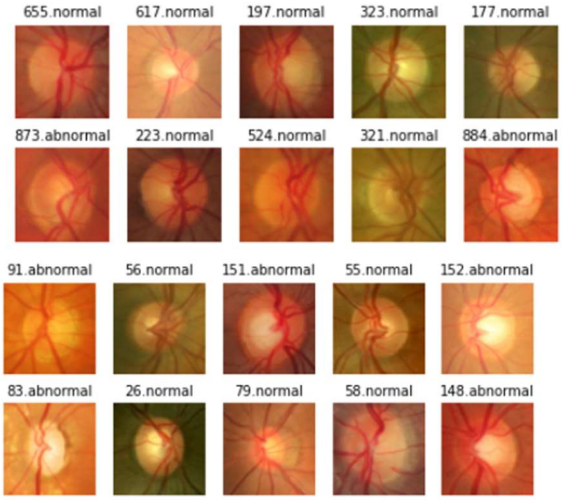
Práctica: Clasificador Neuronal para la detección de Glaucoma

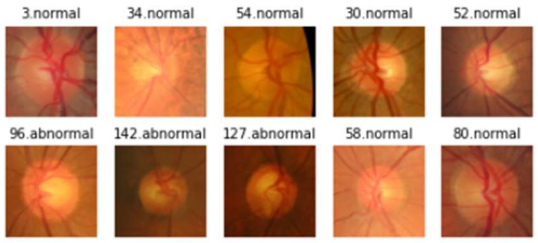
El glaucoma es una patología que afecta al nervio óptico y cuyos orígenes son diversos, es la segunda causa de ceguera por detrás de la diabetes y los efectos en la pérdida de visión son irreversibles. Las causas que lo producen se pueden tratar si la patología es detectada a tiempo.

El objetivo de esta práctica es proponer y entrenar un algoritmo que sea capaz de detectar adecuadamente ojos con glaucoma frente a otros sanos. Los algoritmos de reconocimiento de imágenes se están implementando en la práctica clínica, integrándose en ocasiones directamente en el hardware que se utiliza para la exploración (por ejemplo, en los ecógrafos).

Disponemos de una base de datos formada por imágenes en color de 224x224 píxeles dividida en 10 particiones distintas que se usarán para aplicar un método de cross validation con el objetivo de minimizar errores estadísticos. Cada una de estas particiones, a su vez, contiene tres subconjuntos: train, test y valid. Las imágenes a su vez están etiquetadas de dos formas: normal o abnormal.

Podemos visualizar algunas de las imágenes correspondientes a la partición Fold0 (conjunto train, test y valid respectivamente, 10 imágenes de cada una) con sus etiquetas:






```
(1379, 224, 224, 3)
(1379, 1)
(174, 224, 224, 3)
(174, 1)
(154, 224, 224, 3)
(154, 1)
```

Podemos observar el número de imágenes de cada uno de los conjuntos, divididos en imágenes y etiquetas. He decidido emplear el conjunto de datos “test” para el entrenamiento, ya que consta de más imágenes que el conjunto “valid”, no es una diferencia muy significativa, pero puede ayudar ligeramente.

En todos los modelos propuestos se ofrece una muestra de los datos y la cantidad de imágenes de cada uno de los conjuntos, incluido el método cross validation, en el que cada vez que se carguen datos de una partición diferente se indican el número de imágenes cargadas en cada conjunto y el directorio correspondiente.

Podemos ver que las imágenes están en modo RGB, cada color se forma por combinación de tres canales, cada canal se corresponde con un color primario: Red (rojo), Green (verde), y Blue (azul), se asigna un valor de intensidad a cada color que oscila entre 0 y 255. De la combinación surgen hasta 16,7 millones de colores. Cargamos las imágenes con la librería “cv2”, el valor de

la intensidad lo guardamos en formato 'unit8', números enteros, es importante ya que al emplear un modelo preentrenado, EfficientNet B0, debemos entender el funcionamiento de dicho modelo, si consultamos dicho modelo observamos que este normaliza las imágenes que recibe (https://github.com/keras-team/keras/blob/v2.7.0/keras/applications/efficientnet.py#L320-L322), por tanto, no sería correcto normalizarlas previamente, de ahí el formato, números enteros.

Una vez entendido el formato y la disposición de nuestros datos pasamos al entrenamiento de la red neuronal con las 3 configuraciones aportadas en la práctica. Las 3 primeras configuraciones se deben implementar en un modelo basado en EfficientNet B0, preentrenado con los pesos de Imagenet, al que se le sustituye su capa de clasificación por una capa de GlobalAveragePolling2D, una capa de BatchNormalization, una capa de dropout con probabilidad del 20%, y finalmente, una capa fully connected. Las 3 configuraciones son las siguientes:

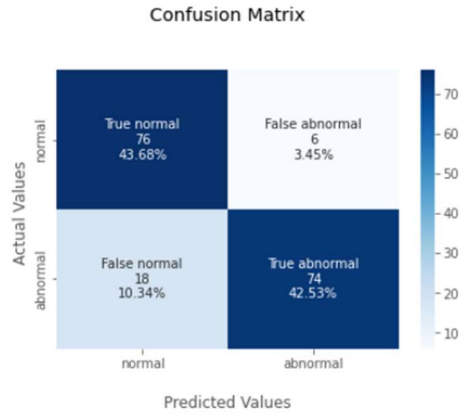
- En la primera se entrena el modelo congelando todas las capas menos las que se han añadido al final:

  Dado que se trata de un problema de clasificación binaria, emplearemos una función de activación final sigmoide (capa fully connected), y una función de coste "binary_crossentropy", en cuanto al optimizador, no existen normas preestablecidas, si consultamos la documentación de keras nos hace una advertencia sobre RMSprop, pero tras diferentes pruebas (Adam, SGD…), curiosamente, los mejores resultados los he obtenido con dicho optimizador, también buscando información al respecto he podido encontrar un estudio previo enfocado en el mismo caso (glaucomas) en el que los resultados con dicho optimizador no son malos:

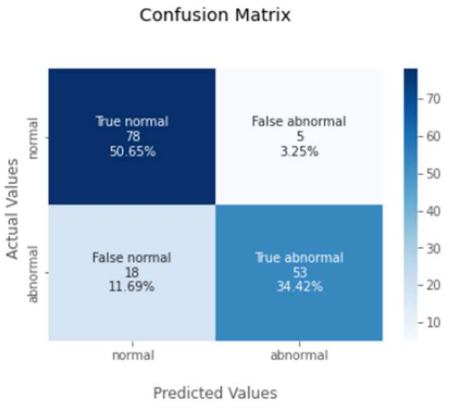
  https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8962290/ (Tabla 14 y 15)

  Tras numerosas pruebas se aprecia una ligera mejoría para el optimizador RMSprop, por tanto, es el que emplearemos. En cuanto al batch_size elegido, también nos advierte en la documentación de keras que un batch_size pequeño puede ayudar a la precisión en el conjunto de validación, pero en mi caso no existe una diferencia notable, por tanto, optaré por un valor de 128, que no sature la memoria de la GPU, en mi caso el código se ejecuta localmente, con una GPU Nvidia RTX 2080 con 8Gb de memoria, por tanto, la elección se basa en no saturar dicha memoria, en cuento a las epochs, se añade la función EarlyStopping(), monitorizando "val_loss" (perdida del conjunto de validación), en mode "min", por tanto, la ejecución parará cuando el valor claramente ya no decrezca, por motivos de tiempo de ejecución también añadiremos el valor epochs=200, para no alargarla mucho en el tiempo (se han debido realizar numerosas pruebas y con tiempos de ejecución mayores sería muy lento) y "patience=70", para esperar un mínimo de epochs antes de parar. También añadimos la función ModelCheckpoint() para guardar el modelo con mejor resultado en el conjunto de validación, de esta forma simplificamos un poco el proceso, enfocandonos en el que mejor resultado nos ofrezca ("save_best_only=True"). Uno de los parámetros más importantes es "learning_rate", de nuevo, no existe una norma establecida que determine un valor óptimo, pero podemos ayudarnos de las gráficas de precisión y pérdida (accuracy y loss) para elegir un valor adecuado, queremos una curva de aprendizaje ascendente a un ritmo decreciente, y una curva de pérdida inversa, ni muy plana ni muy "angulada", tras diferentes pruebas un valor de "learning_rate=1e-3" nos ofrece curvas decentes en un tiempo aceptable:

Observamos un comienzo ligeramente abrupto que podríamos corregir con un valor menor, pero como hemos dicho, también se han tenido en cuenta los tiempos de ejecución, tratando de equilibrar el tiempo que aumenta con la mejora que nos ofrece. Los resultados obtenidos son los siguientes (ya he comentado que el conjunto de datos empleado en el entrenamiento es train y test, luego hemos comprobado el comportamiento con el conjunto valid, también hemos comentado el motivo). A la izquierda vemos test y a la derecha valid:



Classification report (test) :	precision	recall	f1-score	support
0	0.93	0.81	0.86	94
1	0.80	0.93	0.86	80
accuracy			0.86	174
macro avg	0.87	0.87	0.86	174
weighted avg	0.87	0.86	0.86	174



Classification report (test) :	precision	recall	f1-score	support
0	0.94	0.81	0.87	96
1	0.75	0.91	0.82	58
accuracy			0.85	154
macro avg	0.84	0.86	0.85	154
weighted avg	0.87	0.85	0.85	154

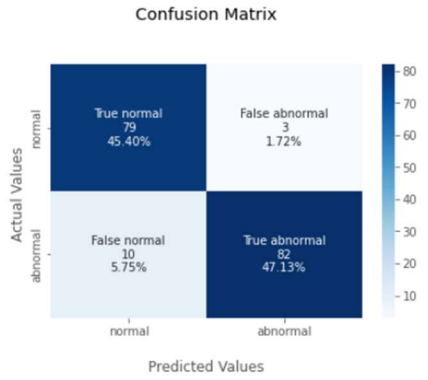
En las gráficas anteriores ya veíamos claramente una tendencia a overfitting del modelo, que vemos demostrada en las matrices de confusión, especialmente tenemos un problema en los falsos negativos, debido a la naturaleza del problema debemos enfocar nuestros esfuerzos en minimizar estos errores, obviamente clasificar un caso de glaucoma como sano en una situación en la que no queremos estar, es preferible un mayor número de falsos positivos siempre. En cualquier caso, es un primer modelo que nos servirá como base para los siguientes.

- Una segunda configuración, a partir de los pesos encontrados para el modelo 1, se entrena descongelando las últimas 20 capas, pero dejando las capas de BatchNorm congeladas:

Para esto crearemos las función unfreeze_model(), que nos permitirá hacer las últimas 20 capas entrenables (layer.trainable = True) menos las BatchNorm (not isinstance(layer, layers.BatchNormalization)). Los motivos en la elección de parámetros seguirán la misma lógica que en el modelo anterior, pero esta vez elegimos un learning_rate=1e-4, veamos las curvas:

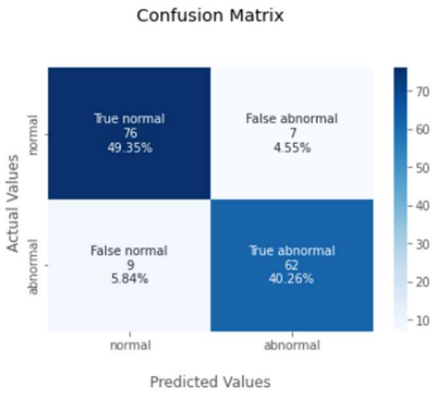
Las curvas de train parecen adecuadas, mientras que las de test siguen un patrón muy diferente al visto anteriormente, la variación tan elevada entre distintas epochs puede hacernos pensar que un valor de learning rate menor podría ayudarnos, pero también hay que tener en cuenta el número reducido de imágenes que tenemos en el conjunto de test para el entrenamiento, esto también puede ser motivo de dicha variación, en cualquier cao podemos comprobar un valor menor (1e-5):

El tiempo de entrenamiento aumenta considerablemente y vemos que los resultados no son significativamente mejores, por tanto, he decidido no reducirlo, también podemos comprobar las matrices de confusión para test y valid (de izquierda a derecha de nuevo). Vemos una mejora considerable respecto al modelo anterior, de nuevo, y como veiamos en las gráficas, existe overfitting y clasifica ligeramente mejor par el conjunto valid que para test, y de nuevo vemos la problemática de los falsos negativos (abordaremos el problema en los modelos finales):



Classification report (test) :

	precision	recall	f1-score	support
0	0.96	0.89	0.92	89
1	0.89	0.96	0.93	85
accuracy			0.93	174
macro avg	0.93	0.93	0.93	174
weighted avg	0.93	0.93	0.93	174



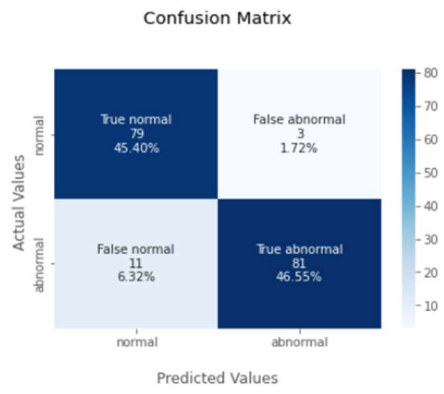
Classification report (valid) :

	precision	recall	f1-score	support
0	0.92	0.89	0.90	85
1	0.87	0.90	0.89	69
accuracy			0.90	154
macro avg	0.89	0.90	0.90	154
weighted avg	0.90	0.90	0.90	154

- En la tercera configuración, a partir de los pesos del modelo 2, se descongelan todas las capas y se entrena la red en su totalidad:

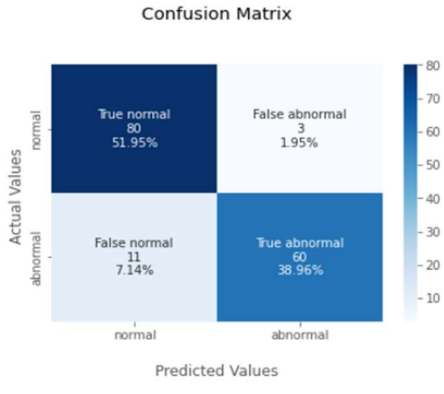
De nuevo, misma lógica en la elección de hiperparámetros, en este caso elegimos un learning_rate= 1e-5, veamos los resultados:

La presencia de overfitting es muy obvia, pero en el caso de un learning rate menor estamos en el mismo caso que el modelo anterior, ya en este los tiempos de ejecución son bastante más elevados (por el hecho de descongelar todas las capas) y la ganancia es mínima, en los notebooks podemos observar gráficas con diferentes valores de learning rate para comprobar el motivo de la elección de dicho valor, no añadiré capturas de más gráficas para no extenderme en exceso. El overfitting es el problema con el que trataremos de lidiar en los siguientes modelos, aunque la cantidad de datos es limitada y esto puede suponer un problema difícil de solventar, pero trataremos de mejorar la predicción del modelo. Observemos las matrices de confusión del modelo resultante, de nuevo a la izquierda test (empleado para entrenar) y a la derecha valid:



Classification report (test) :

	precision	recall	f1-score	support
0	0.96	0.88	0.92	90
1	0.88	0.96	0.92	84
accuracy			0.92	174
macro avg	0.92	0.92	0.92	174
weighted avg	0.92	0.92	0.92	174



Classification report (test) :

	precision	recall	f1-score	support
0	0.96	0.88	0.92	91
1	0.85	0.95	0.90	63
accuracy			0.91	154
macro avg	0.90	0.92	0.91	154
weighted avg	0.92	0.91	0.91	154

No se aprecia una mejora muy considerable repecto al segundo modelo y seguimos manteniendo la problemática de los falsos negativos.

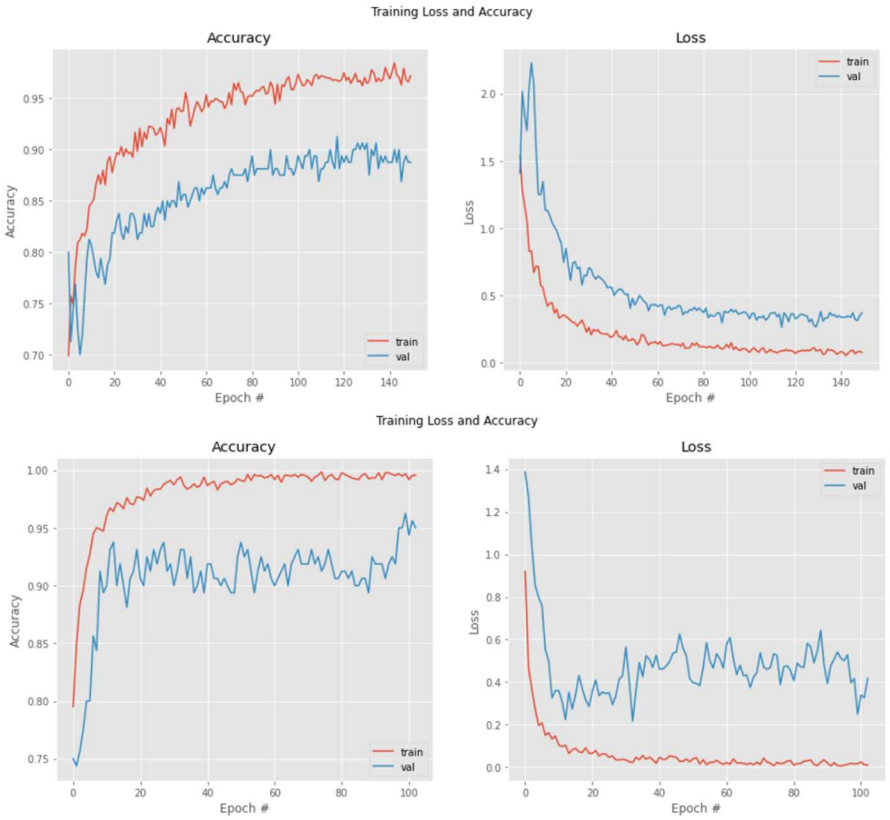
- Para los últimos modelos he decidido crear una red propia (sin entrenamiento previo), para comparar los resultados, es de esperar que los resultados sean peores, pero me parece interesante comparar el desempeño con un modelo preentrenado, y por último he decidido crear diferentes modelos basados en los 3 anteriores, tratando de lidiar con el problema de overfitting y los falsos negativos, podría haber empleado otra red neuronal preentrenada (VGG16, Inception…) y comparar sus resultados, pero me ha parecido más interesante incidir sobre la red EfficientNet B0, tratando de mejorar al máximo posible el modelo resultante.

En el caso de la red EfficientNet B0, he creado distintos modelos con ligeras modificaciones, lo primero sería añadir el proceso ImageDataGenerator(), hemos visto en la documentación de la asignatura que generar imágenes a partir de las que tenemos (rotando, aumentando, descentrando…) puede ser muy útil, no sólo aumenta el número de imágenes, si no que aumenta la variación en el conjunto, facilitando al modelo generalizar, para ello aplicaremos la función mencionada con los siguientes parámetros: rotation_range=20, zoom_range=0.15, width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15, horizontal_flip=True, fill_mode="nearest", validation_split=0.2 y dtype = 'uint8' para mantener el formato de número entero, para las imágenes de test no haremos transformaciones, para no afectar a los resultados. Otro elemento importante en estos modelos puede ser el implementar un learning_rate que se modifique a lo largo del tiempo, para ello haremos uso de la función ExponentialDecay(), que nos permite reducir su valor a lo largo del entrenamiento definiendo la cantidad que decae y los tiempos, según el modelo emplearemos parámetros distintos. En cuanto al modelo base, la idea es similar al propuesto, pero también probaremos con un modelo nuevo en el que añadimos algunas capas fully connected antes de la salida para ver si es capaz de extraer nuevas características, se han probado modelos con una única capa fully_connected de 1024 nodos o neuronas y una capa dropout con probabilidad del 20%, con varias capas fully connected con capas BatchNorm y Dropout (0.5 y 0.2)con 1024, 512 y 256 nodos o neuronas, reduciendo la complejidad de la salida sucesivamente, y 3 capas fully connected

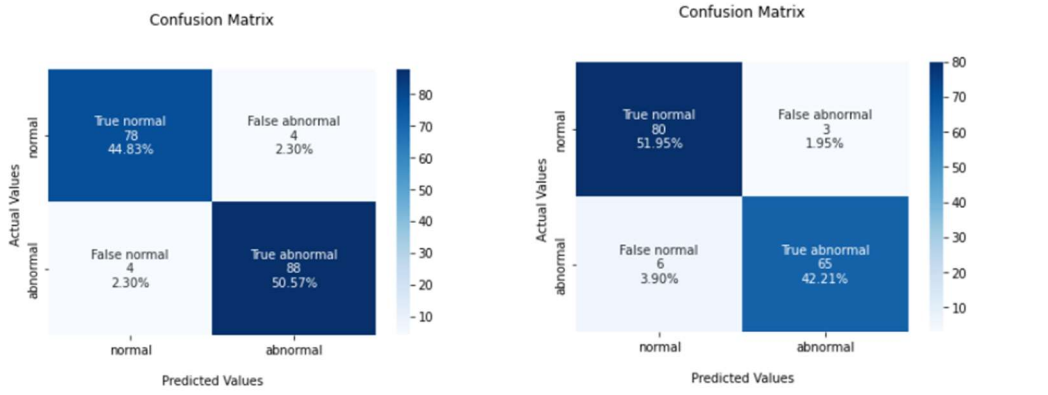
esta vez sin capas BatchNorm y Dropout intermedias, finalmente el modelo elegido ha sido el segundo, con capas intermedias. Podemos apreciar diferencias en las gráficas de accuracy y loss respecto a los modelos anteriores, en una primera fase con todas las capas de EfficientNet B0 congeladas:

Destacar el impacto de la frecuencia con la que reducimos el learning_rate, lo calculamos en base a los steps que hay en cada epoch, con una reducción menos frecuente obtenemos resultados diferentes, con menores tiempos de ejecución, y aunque las gráficas son distintas el impacto en los resultados no es muy notable:

Continuamos entrenando el modelo siguiendo los pasos anteriores, llegando a un modelo en el que parece que hemos podido mejorar considerablemente los resultados, de nuevo la elección de hiperparámetros ha seguido la lógica de los modelos anteriores, priorizando un buen resultado con un tiempo de ejecución aceptable, por ejemplo, en el entrenamiento de todo el modelo, con las capas descongeladas vemos los parámetros elegidos frente a otro modelo posiblemente mejor, pero con un coste computacional mayor, debido a que tendremos que hacer un cross validation con muchos datos elegimos un modelo un poco más rápido, la ejecución ya supera las 4 horas, esto deja un margen de mejora a nuestro modelo, vemos la comparativa (modelo más complejo arriba y elegido (más rápido) abajo, curiosamente también ha ofrecido mejores resultados):



También vemos los resultados de las matrices de confusión, resultados prometedores. También hemos probado el modelo original sin añadir capas, incorporando unicamente la generación de imágenes y la corrección del learning_rate y los resultados también han sido mejores, similares a los vistos, no aportaré capturas para no extenderme en exceso, podemos ver los notebooks (PRAC_1_con_datagenerator_sin_capas añadidas.ipynb).



Classification report (test) :

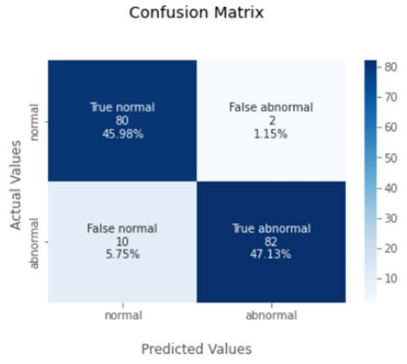
	precision	recall	f1-score	support
0	0.95	0.95	0.95	82
1	0.96	0.96	0.96	92
accuracy			0.95	174
macro avg	0.95	0.95	0.95	174
weighted avg	0.95	0.95	0.95	174

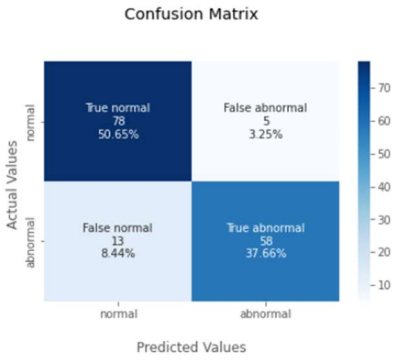
Classification report (test) :

	precision	recall	f1-score	support
0	0.96	0.93	0.95	86
1	0.92	0.96	0.94	68
accuracy			0.94	154
macro avg	0.94	0.94	0.94	154
weighted avg	0.94	0.94	0.94	154

El último modelo es una red propia que aprende directamente de los datos, podemos ver la arquitectura de dicha red en el notebook “PRAC_1_own_model.ipynb”, también hemos añadido la función ImageDataGenerator(), los resultados son peores que en el modelo anterior pero mejores de lo esperado, aunque los tiempos de ejecución son muy elevados

en comparación, tras 400 epochs (el modelo podría seguir mejorando pero los tiempos son muy elevados y en las pruebas realizadas no parece que vaya a superar los resultados del último modelo basado en EfficientNet B0), vemos los resultados de las matrices de confusión en test y valid, de izquierda a derecha:





Classification report (test) :

	precision	recall	f1-score	support
0	0.98	0.89	0.93	90
1	0.89	0.98	0.93	84
accuracy			0.93	174
macro avg	0.93	0.93	0.93	174
weighted avg	0.93	0.93	0.93	174

Classification report (test) :

	precision	recall	f1-score	support
0	0.94	0.86	0.90	91
1	0.82	0.92	0.87	63
accuracy			0.88	154
macro avg	0.88	0.89	0.88	154
weighted avg	0.89	0.88	0.88	154

Por último, en la validación cruzada, he decidido hacerlo con los dos mejores modelos, genrando imágenes con learning rate variable, con capas añadidas y sin capas añadidas, los resultados son parecidos pero ligeramente mejores para el primero, hemos cerado un dataframe con los valores loss, accuracy, f1, precission y recall tanto en el conjunto de test como en valid (aportaré los resultados del modelo con capas añadidas, en cualquier caso se pueden consultar los resultados de ambos en los notebooks "PRAC_1_cross_validation_sin capas_añadidas.ipynb" y "PRAC_1_cross_validation_con capas_añadidas.ipynb", también vemos la diferencia de resultados en distintas ejecuciones, se aportan 2 en cada notebook):

	Fold	loss_test	accuracy_test	f1_score_test	precision_test	recall_test	loss_valid	accuracy_valid	f1_score_valid	precision_valid	recall_valid
0	Fold0	0.275220	0.931035	0.933379	0.942130	0.930044	0.249733	0.941558	0.934038	0.981818	0.892381
1	Fold1	0.222934	0.942529	0.947359	0.921100	0.978472	0.214963	0.935065	0.924242	0.884089	0.970053
2	Fold2	0.264359	0.948276	0.930160	0.954678	0.911111	0.134972	0.948052	0.938461	0.953846	0.925275
3	Fold3	0.354905	0.954023	0.958806	0.980811	0.940079	0.114609	0.954545	0.940842	0.973333	0.915219
4	Fold4	0.212875	0.948276	0.950968	0.939286	0.967305	0.170450	0.948052	0.950256	0.955556	0.948586
5	Fold5	0.315049	0.925287	0.911367	0.926763	0.903011	0.304562	0.928571	0.930098	0.937431	0.923329
6	Fold6	0.135177	0.971264	0.975496	0.991667	0.961261	0.218851	0.935065	0.930122	0.939346	0.923492
7	Fold7	0.109315	0.977012	0.976586	1.000000	0.956845	0.145869	0.954545	0.951927	0.946282	0.959615
8	Fold8	0.421954	0.919540	0.921107	0.922476	0.921693	0.339244	0.909091	0.907715	0.912222	0.909706
9	Fold9	0.274069	0.965517	0.961611	0.965241	0.961039	0.421873	0.909091	0.905754	0.912920	0.905735

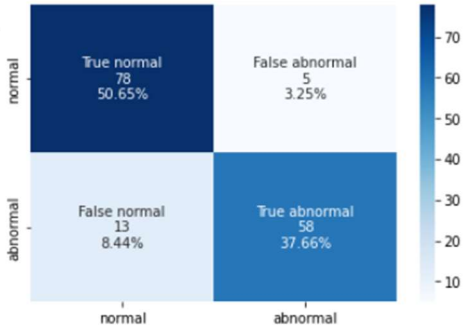
```
El valor promedio de F1 score para las 10 particiones en el conjunto de datos de test es: 0.9466838717460633
El valor promedio de F1 score para las 10 particiones en el conjunto de datos de validación es: 0.9313455820083618
La desviación estándar de F1 score para las 10 particiones en el conjunto de datos de test es: 0.022271844633090093
La desviación estándar de F1 score para las 10 particiones en el conjunto de datos de validación es: 0.01562104155289475
```

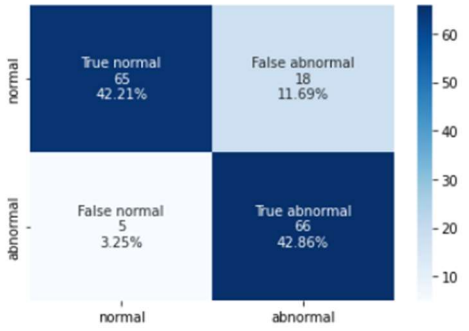
Los resultados en el conjunto valid (no empleado para entrenar) son ligeramente peores, pero con menos desviación, apreciamos las diferencias en las distintas particiones, pero en conjunto el modelo es capaz de generalizar bien, manteniendo un valor de f1 relativamente estable (tanto en test como en valid, 0946 y 0.9313 respectivamente).

En cuanto a los falsos negativos, la red neuronal nos devuelve las probabilidades de pertenecer a una clase y fijamos un límite a partir del cual pertenecerá a una clase u otra,

una forma sencilla de reducirlos sería cambiando este límite, observando los resultados de los valores mal clasificados, pongamos de ejemplo el modelo propio con una red neuronal creada (sin Efficientnet), si nos fijamos en los valores mal clasificados y cambiamos el límite paraa determinar un negativo, por ejemplo a 0.14 en vez de 0.5:

	prob	label	pred_label
6	0.764548	0	1
10	0.176792	1	0
30	0.161777	1	0
34	0.385376	1	0
48	0.217502	1	0
51	0.019673	1	0
56	0.506446	0	1
71	0.141696	1	0
74	0.400963	1	0
82	0.205282	1	0
92	0.024085	1	0
99	0.211348	1	0
100	0.981835	0	1
113	0.002738	1	0
116	0.666219	0	1
121	0.000129	1	0
131	0.584299	0	1
132	0.057060	1	0





Reducimos mucho los falsos negativos, pero a cambio obtenemos un número mucho mayor de falsos positivos, obviamente en un modelo más preciso las cosas serían distintas y podríamos obtener una mayor precisión, también existen otras estrategias de selección de imágenes en el proceso de entrenamiento para mejorar estos errores que podrían ayudar.

Diseño de particiones: En la documentación de la asignatura vemos que una división aleatoria con una distribución de la variable objetivo compensada es la mejor estrategia, en nuestro caso existe un número ligeramente mayor de ejemplos con la etiqueta “normal”, pero no está muy descompensado y se mantiene a lo largo de todas las particiones (está justificado), por tanto, creo que es una estrategia correcta, lo que sería una opción interesante sería, por ejemplo, emplear k-2 particiones para entrenar cada modelo, otra partición para validación y otra para test, e ir iterando sobre distintas combinaciones, tendríamos un mayor número de imágenes de entrenamiento que quizás podría ayudar , pero esto también podría ser un paso posterior al llevado a cabo en la práctica, una vez tenemos cierta confianza de que nuestro algoritmo es consistente, para después pasar a entrenarlo con todo el conjunto de datos.

Conclusión: El uso de modelos preentrenados en la clasificación de imágenes y el proceso de entrenamiento por capas es una opción muy interesante para obtener buenos resultados y con menores tiempos de ejecución, la importancia en la selección de parámetros, el introducir variabilidad añadiendo imágenes modificadas del propio conjunto de entrenamiento y una elección correcta de la velocidad de aprendizaje me han parecido aspectos muy determinantes, hemos conseguido un modelo capaz de generalizar mejor, es decir, menos overefitting, mejorando el modelo base planteado en la práctica y reduciendo los falsos negativos considerablemente, sin un patrón constante en todas las particiones en el que los falsos negativos superen siempre a los falsos positivos, podemos ver en los notebooks las matrices de confusión en cada partición. Por último, destacar el empleo de cross-validation para comprobar la consistencia del modelo, también puede emplearse para la elección de hiperparámetros, pero sin duda es una técnica muy útil. Es muy importante centrar esfuerzos en tratar de corregir la tendencia a overfitting de las redes neuronales, existen otras formas relacionadas con la distribución de las particiones que no hemos explorado en la práctica.