Proyecto Final
Visión por Computador

Álvaro Beltrán Camacho
Sergio Cabezas González de Lara

Doble Grado de Ingeniería Informática y Matemáticas

Universidad de Granada
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

29 de enero de 2021

Índice

Brain Tumor Detector

1.1. Definición del problema

1.1.1. Introducción

Un tumor cerebral está considerado como una de las enfermedades más agresivas que existen, tanto en adultos como en niños. Representa el 85-90 % de los tumores del sistema nervioso central que se tratan cada año. De hecho, cada año alrededor de 11.700 personas son diagnosticadas con un tumor cerebral. La tasa de supervivencia a cinco años es aproximadamente del 34 % para hombres y del 36 % para mujeres. Los tumores cerebrales pueden ser malignos o benignos. Y se debe implementar un tratamiento, una planificación y un diagnóstico precisos para mejorar la vida de los pacientes. La mejor técnica para detectar tumores cerebrales es la resonancia magnética. Estas resonancias generan una gran cantidad de datos en imágenes, el radiólogo examina estas imágenes de forma manual, pero este examen puede ser propenso a errores debido al nivel de complejidad involucrado.

La aplicación de técnicas de clasificación automatizada que utiliza Machine Learning e Inteligencia Artificial, han demostrado consistentemente una mayor precisión que la clasificación manual. Por lo tanto, proponer un sistema que realice la detección y clasificación mediante el uso de algoritmos de aprendizaje profundo, utilizando redes neuronales convolucionales, sería útil para los médicos de todo el mundo.

1.1.2. Descripción del problema

Vamos a clasificar imágenes de resonancias magnéticas cerebrales para responder a si estas imágenes contienen o no un tumor. Además, clasificaremos también en base a los tipos de tumor con las siguientes clases: no tumor, glioma, meningioma y tumor pituitario.

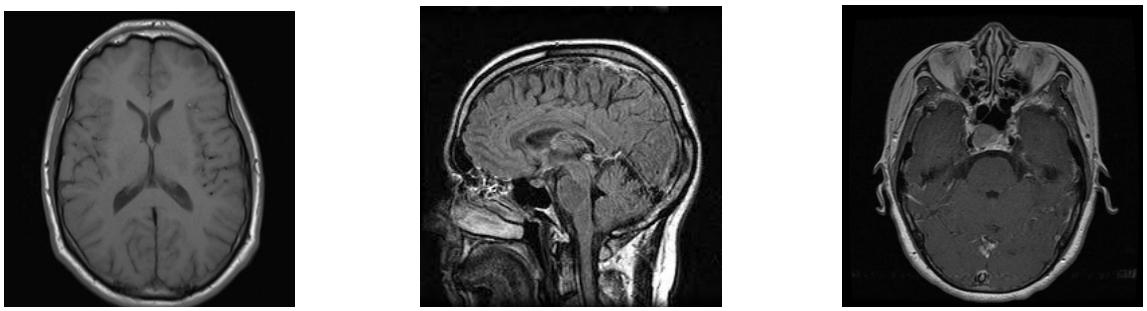
Para ello, vamos a subdividir el problema en dos partes: decidir si el paciente tiene o no un tumor, para finalmente acabar clasificando entre las cuatro clases especificadas anteriormente donde entran los tipos de tumores.

Hemos elegido la red DenseNet121 preentrenada con IMAGENET para clasificación para abordar los problemas anteriores, debido a su versatilidad y a su buen funcionamiento en general con problemas en el tratado de imágenes.

1.1.3. Descripción de la base de datos

La base de datos elegida para abordar el problema se llama Brain Tumor Classification (MRI) [1]. Nos encontramos ante una base de datos de imágenes médicas, concretamente, de resonancias magnéticas cerebrales. La base de datos está dividida en dos conjuntos de training y test. Y cada división está subdividida en cuatro conjuntos referentes a las clases especificadas anteriormente (no tumor, glioma, meningioma y tumor pituitario). Las imágenes de la base de datos no son todas del mismo tamaño, se encuentran tomadas desde diferentes perfiles y en formato RGB.

En total la base de datos aporta 3264 imágenes. De las cuales corresponden 2870 para training (no tumor 395, glioma 826, meningioma 822 y tumor pituitario 827) y 394 para test (no tumor 105, glioma 100, meningioma 115 y tumor pituitario 74).



(a) Imagen sin tumor (b) Imagen sin tumor (c) Imagen con tumor pituitario

Figura 1.1: Ejemplos de la Base de datos

1.1.4. Descripción del código

Los códigos se adjuntarán junto con la memoria como cuadernos de python ejecutados en GoogleColab.

1.2. Binary Brain Tumor Detector

1.2.1. Datos

Teniendo en cuenta que la base de datos nos aporta 500 imágenes en total sin tumor, entre training y test. Para crear la clase de imágenes con tumor, hemos decidido extraer 500 imágenes repartidas entre los distintos tipos de tumores. Teniendo en total un conjunto de 1000 imágenes. Para las etiquetas hemos decidido generar dos pares (0,1) para imágenes sin tumor y (1,0) para imágenes con tumor. Finalmente, tras un barajado de las imágenes, escogemos un 75 % para training y un 25 % para test.

Ajustamos el formato de las imágenes que cargamos a RGB y al tamaño 224x224 con la función resize de OpenCV, con tal de que todas las imágenes tengan las mismas dimensiones.

1.2.2. Modelos

Vamos a usar de modelo base DenseNet121 a la cual le iremos proponiendo en las siguientes secciones una serie de mejoras.

Aplicaremos a los datos de entrada, tanto de training como de test, la función de preprocesado correspondiente al modelo usado. Lo que hacemos es adecuar nuestros datos con respecto a la media y desviación típica de ImageNet, que es la base de datos con la que DenseNet ha sido preentrenada.

DenseNet121 preentrenado [Modelo Simple]

Lo primero que hacemos es usar la red DenseNet121 preentrenada con los datos de ImageNet, quitándole su última capa para usar la red como extractor de características, añadiendo una capa final de GlobalAveragePooling para transformar el tensor resultado de (7, 7, 1024) en un vector de características de tamaño 1024 y posteriormente una capa Dense con activación softmax de tamaño 2, que corresponde a las dos clases en las que queremos clasificar las imágenes.

DenseNet121 preentrenado mejorado

Para mejorar el modelo anterior, hemos extraído características con la red DenseNet121 preentrenada igual que antes, pero esta vez no le hemos añadido la capa final de GlobalAveragePooling, por lo que la salida de la red es un tensor de (7, 7, 1024). Entonces, además de usar la capa Dense con activación softmax de tamaño 2 como salida del problema de clasificación, vamos a añadir capas intermedias justo después de la salida de la red. Concretamente añadimos una capa convolucional, una capa de pooling y una capa Dense más, junto con capas del tipo BatchNormalization y Dropout que se usan para regularización.

Estos cambios van enfocados a mejorar los resultados del modelo, ya que estamos aumentando el número de capas que entrenamos con nuestros datos.

Layer No.	**Layer Type**	**Kernel size**	**Input | Output dimension**	**Input | Output channels**
1	Conv2D	3	7 | 7	1024 | 2048
2	BatchNorm	-	7 | 7	-
3	Relu	-	7 | 7	-
4	Dropout (0.5)	-	7 | 7	-
5	GlobalAvgPooling2D	7	7 | 1	-
6	Linear	-	2048 | 200	-
7	Relu	-	200 | 200	-
8	Dropout (0.5)	-	200 | 200	-
9	Linear	-	200 | 2	-

Tabla 1.1: Arquitectura del Modelo Mejorado.

1.2.3. Entrenamiento

Para entrenar los modelos, lo primero que hacemos es predecir las características con el modelo DenseNet121 indicado, mediante la función predict de Keras. Y luego entrenamos las arquitecturas que hemos definido anteriormente con la salida de estas predicciones.

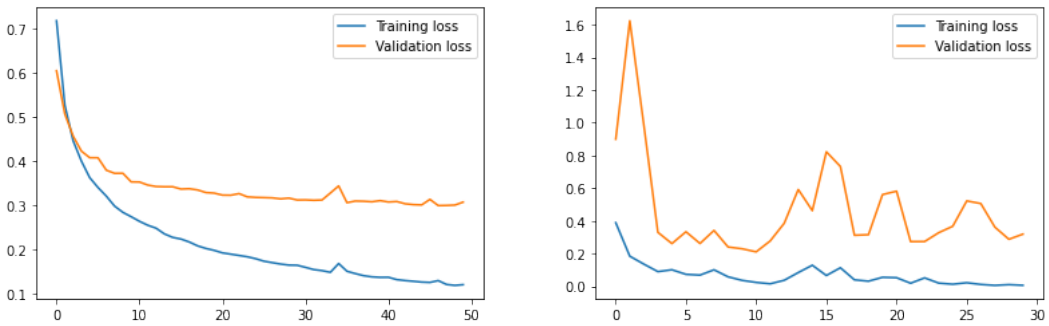
Los parámetros usados para el entrenamiento son los siguientes:

- **BatchSize.** Hemos usado batch size de 32 debido al tamaño de nuestra muestra.
- **Función de Pérdida.** Para la función de pérdida hemos usado Binary Crossentropy (Función de pérdida de entropía cruzada para el caso binario), porque nos encontramos ante un problema de clasificación binaria.
- **Optimizador.** Como optimizador hemos usado Gradiente Descendiente Estocástico (SGD), con un learning rate de 0.01, un momentum de 0.9, un decay de 1e-6 y con la opción de nesterov activa. Todos los parámetros han sido escogidos experimentalmente observando mejoras en el entrenamiento.
- **Métrica.** Hemos decidido escoger la métrica Accuracy para maximizar el número de instancias bien clasificadas.
- **Conjunto Validación.** Usamos la opción de validation_split que nos da Keras para definir el conjunto de validación sobre un porcentaje del conjunto de training, el porcentaje elegido ha sido 10 %.
- **Épocas.** Hemos entrenado durante 50 épocas el modelo.

Finalmente, en el modelo mejorado, hemos añadido como callback Early Stopping ante la función de pérdida y la métrica. Con una paciencia de 20 épocas y la opción de restaurar los mejores pesos activos.

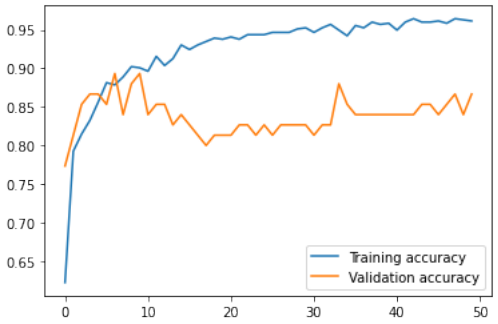
1.2.4. Resultados

Tras entrenar el modelo simple y el modelo mejorado, obtenemos los siguientes resultados para el problema:

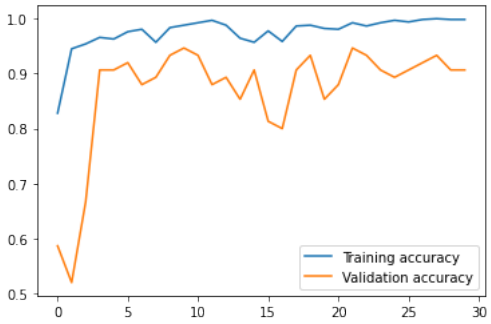


(a) Función de Pérdida Modelo Simple. (b) Función de Pérdida Modelo Mejorado.

Figura 1.2: Gráficas de la Función de Pérdida.



(a) Accuracy Modelo Simple.



(b) Accuracy Modelo Mejorado.

Figura 1.3: Gráficas de Accuracy.

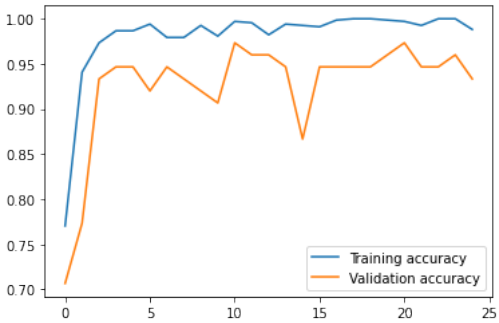
	Test Loss	**Test Accuracy**
Modelo Simple	0.14	0.948
Modelo Mejorado	0.0887	0.9639

Tabla 1.2: Datos resultantes al evaluar el modelo con el conjunto de Test

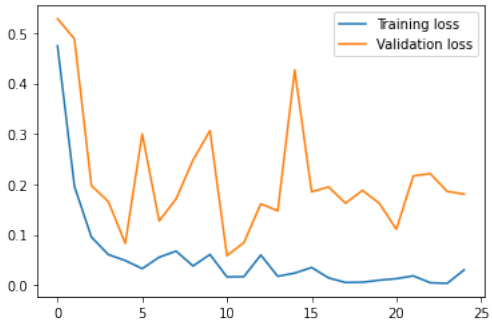
1.2.5. Fine Tuning

Una vez probado que el modelo mejorado da mejor resultado por el hecho de entrenar unas capas más que el modelo base con nuestros datos. Para mejorar aún más este modelo mejorado aplicaremos lo que denominamos Fine Tuning, que se trata de reentrenar el modelo completo (incluidas las capas de DenseNet) con nuestros datos del problema.

Tras reentrenar el modelo 25 épocas con esta técnica, se obtienen los siguientes resultados:



(a) Accuracy Modelo con Fine Tuning.



(b) Pérdida Modelo con Fine Tuning.

Figura 1.4: Gráficas Modelo con Fine Tuning.

	Test Loss	Test Accuracy
Modelo Mejorado	0.0887	0.9639
Modelo Mejorado Fine Tuning	0.0781	0.984

Tabla 1.3: Datos resultantes al evaluar el modelo con el conjunto de Test

Como era de esperar, se han mejorado los resultados que se habían obtenido anteriormente, por lo que nos quedamos con esta última opción para resolver nuestro problema.

Además, hemos comprobado que usando como métrica F1 en vez de accuracy, también se obtiene con el entrenamiento un valor de F1 muy alto.

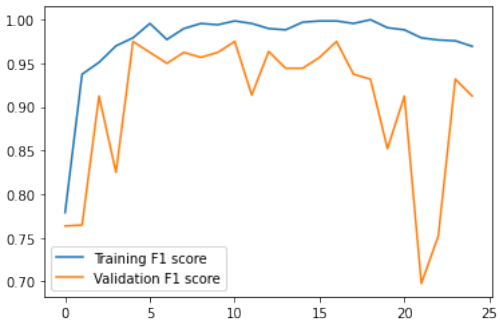
En primer lugar la métrica F1 se define como:

$$recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{1.1}$$

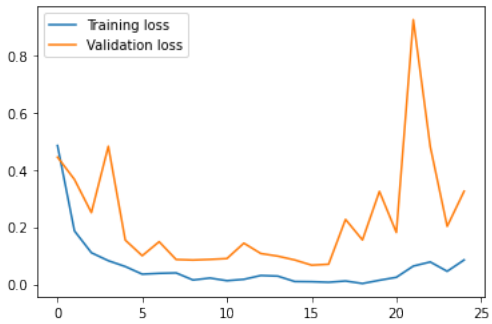
$$precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{1.2}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{1.3}$$

Y los resultados para el modelo mejorado con Fine Tuning con esta métrica son:



(a) Métrica F1 Modelo con Fine Tuning.



(b) Pérdida Modelo con Fine Tuning.

Figura 1.5: Gráficas Modelo con Fine Tuning.

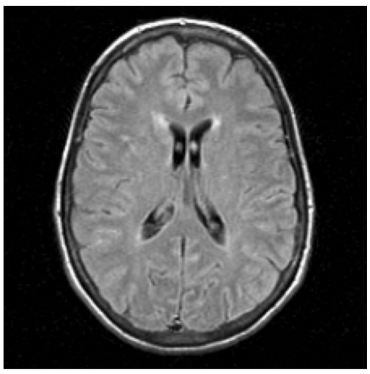
	Test Loss	Test Accuracy
Modelo Mejorado Fine Tuning F1	0.100	0.9639

Tabla 1.4: Datos resultantes al evaluar el modelo con el conjunto de Test

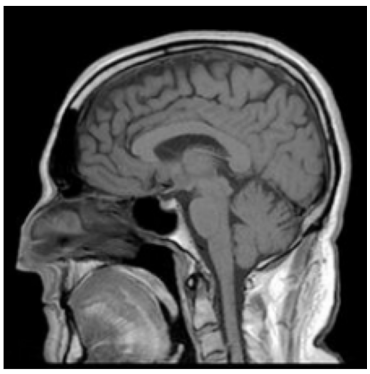
1.2.6. Conclusión

Observando los resultados obtenidos, se demuestra que estas técnicas son una solución al problema de detectar tumores en imágenes médicas debido a que son capaces de mejorar el porcentaje de acierto en el diagnóstico de la existencia de un tumor cerebral dado por el ojo humano de un médico.

Finalmente, como comprobación de nuestro detector hemos extraído unos ejemplos del conjunto de test y comprobamos observando su etiqueta que efectivamente nuestro detector acierta en su diagnóstico.

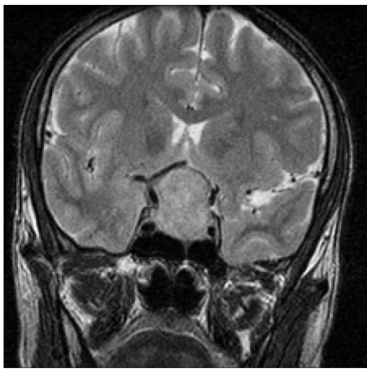


(a) label : (0,1) [non-tumor]
prediction label : (9.55e-10 , 1.000)

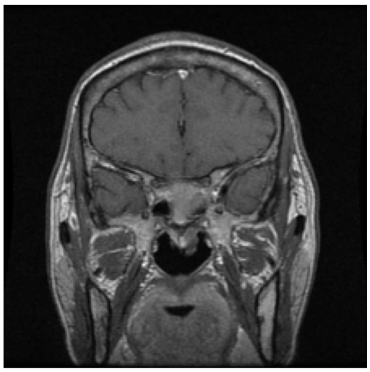


(b) label : (0,1) [non-tumor]
prediction label : (5.71e-8 , 1.000)

Figura 1.6: Resultados en imágenes sin tumor



(a) label : (1,0) [tumor]
prediction label : (1.000 , 5.28e-8)



(b) label : (1,0) [tumor]
prediction label : (1.000 , 1.05e-9)

Figura 1.7: Resultados en imágenes con tumor

1.3. Multiclass Brain Tumor Detector

1.3.1. Datos

Como dijimos anteriormente, la base de datos nos aporta 500 imágenes sin tumor, pero ahora queremos clasificar en 4 tipos (no tumor, glioma, meningioma y tumor pituitario), por lo que hemos cogido 500 imágenes de cada tipo haciendo un total de 2000 imágenes. De las cuales, tras un barajado, hemos escogido el 25 % para test y el resto para training. Además, hemos decidido asignar como etiquetas: (1,0,0,0) para no tumor, (0,1,0,0) glioma, (0,0,1,0) meningioma y (0,0,0,1) pituitario.

También, hemos ajustado el formato de las imágenes igual que en el problema anterior.

1.3.2. Modelos

Como en el problema binario, usaremos para el modelo base DenseNet121 y al igual que antes también le iremos proponiendo una serie de mejoras.

En primer lugar aplicamos a los datos de training y de test la función de preprocesado correspondiente al modelo usado para adecuar nuestros datos a la red DenseNet ya preentrenada con Imagenet.

DenseNet121 preentrenado [Modelo Simple]

Este modelo es el modelo simple 1.2.2 definido para el problema binario excepto que la capa Dense ahora es de tamaño 4.

DenseNet121 preentrenado mejorado

Para mejorar el modelo anterior, hemos extraído características con la red DenseNet121 preentrenada con Imagenet, pero esta vez no le hemos añadido la capa final de GlobalAverage-Pooling, por lo que la salida de la red es un tensor de (7, 7, 1024). Entonces, además de usar la capa Dense con activación softmax de tamaño 4 como salida del problema de clasificacíón, vamos a añadir capas intermedias justo después de la salida de la red. Especificamos la arquitectura a continuación.

Layer No.	**Layer Type**	**Kernel size**	**Input | Output dimension**	**Input | Output channels**
1	Conv2D	3	7 | 5	1024 | 512
2	BatchNorm	-	5 | 5	-
3	Relu	-	5 | 5	-
4	Conv2D	3	5 | 5	512 | 1024
5	BatchNorm	-	5 | 5	-
6	Relu	-	5 | 5	-
7	MaxPooling2D	2	5 | 3	
8	Dropout (0.5)	-	3 | 3	-
9	Linear	-	9216 | 200	-
10	Relu	-	200 | 200	-
11	Dropout (0.5)	-	200 | 200	-
12	Linear	-	200 | 4	-

Tabla 1.5: Arquitectura del Modelo Mejorado.

DenseNet121 + Módulo Inception modificado

Para ayudarnos a la implementación del módulo Inception modificado y realizar una serie de cambios nos hemos ayudado del artículo de Inception-v4 [3] y de la página web [2] para la implementación.

El módulo Inception creado se basa en la concatenación de una convolución 1x1, una convolución 3x3, una convolución 5x5 y un Maxpooling. Además, hemos implementado la opción de poder concatenar la capa de entrada al módulo para controlar el desvanecimiento del gradiente.

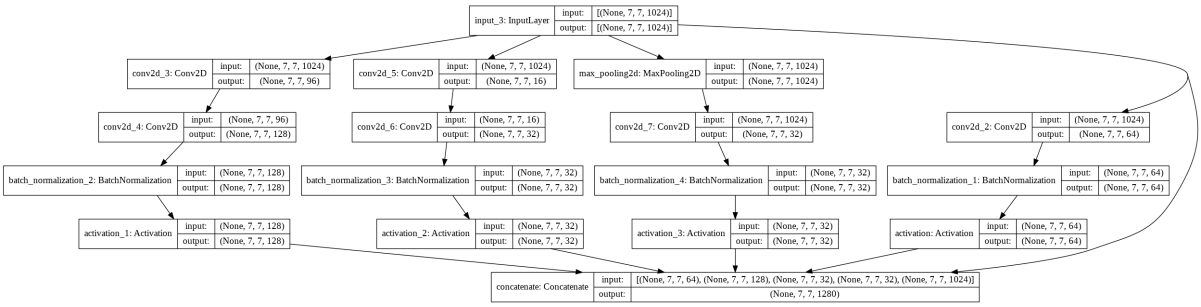


Figura 1.8: Imagen Módulo Inception modificado

La arquitectura que hemos creado para tratar el tensor de tamaño (7, 7, 1024) obtenido tras predecir las imágenes con DenseNet121 se compone de dos módulos Inception modificados, el primero con un arco hacia la primera capa y el segundo sin arco. La salida tras los módulos Inception es un tensor de (7, 7, 480). Además, este último tensor será tratado por la siguiente arquitectura secuencial:

Layer No.	**Layer Type**	**Kernel size**	**Input | Output dimension**	**Input | Output channels**
1	Dropout (0.25)	-	7 | 7	-
2	GlobalAvgPooling	7	7 | 1	-
3	Linear	-	480 | 200	-
4	Relu	-	200 | 200	-
5	Dropout (0.5)	-	200 | 200	-
6	Linear	-	200 | 4	-

Tabla 1.6: Arquitectura de la parte secuencial del modelo con Inception modificado.

Finalmente, la arquitectura resultante es la siguiente:

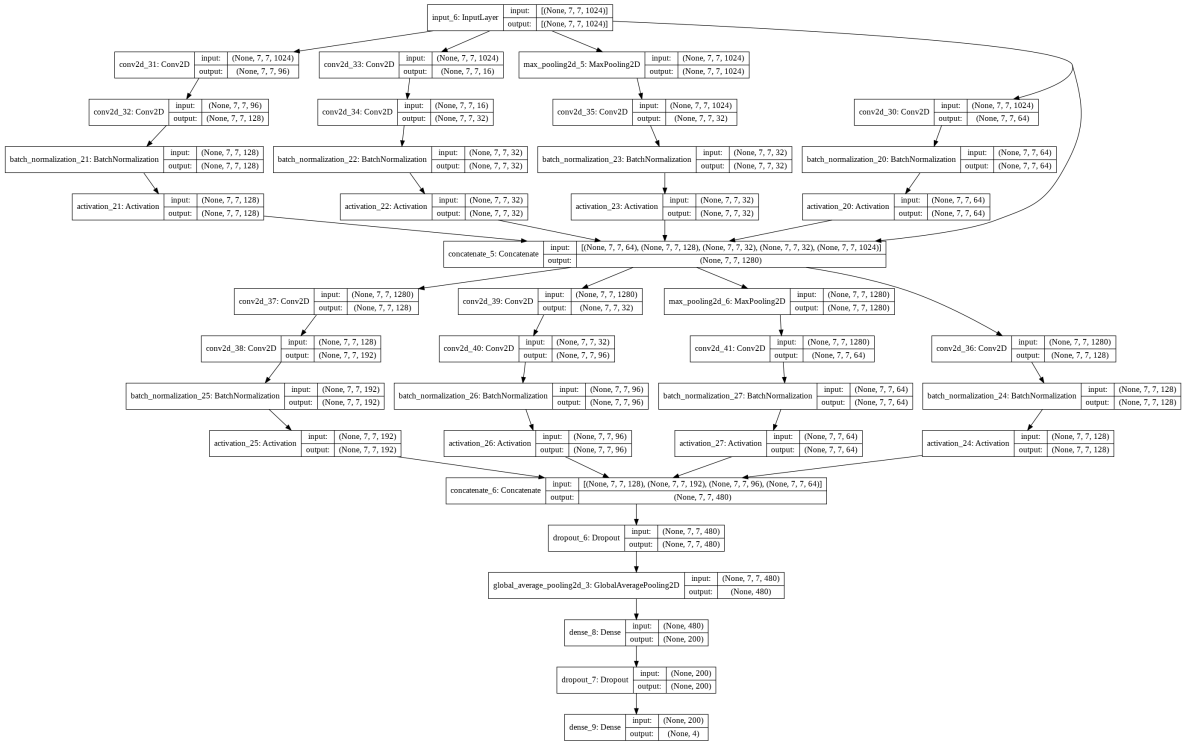


Figura 1.9: Arquitectura

1.3.3. Entrenamiento

Para entrenar los modelos, primero predecimos las características con el modelo DenseNet121 al igual que hicimos para el problema de clasificación binaria.

Los parámetros usados para el entrenamiento son los siguientes:

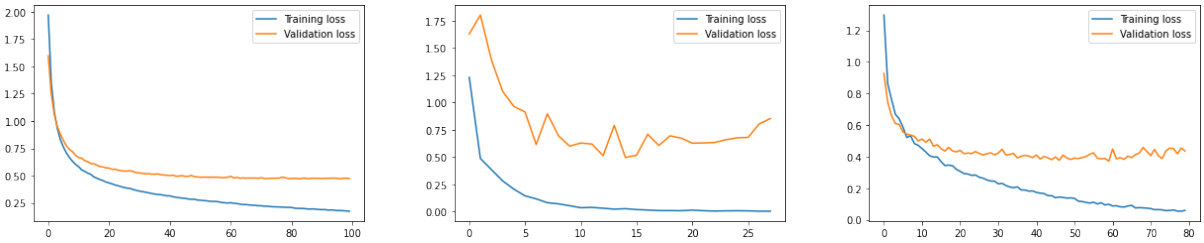
- **BatchSize.** Hemos usado batch size de 128, excepto para el modelo con Inception modificado que hemos usado como tamaño 32.
- **Función de Pérdida.** Para la función de pérdida hemos usado Categorical Crossentropy (Función de pérdida de entropía cruzada categórica), porque nos encontramos ante un problema de clasificación multiclase.
- **Optimizador.** Como optimizador hemos usado Gradiente Descendiente Estocástico (SGD) con un momentum de 0.9, un decay de 1e-6 y con la opción de nesterov activa. El learning rate se ha adecuado experimentalmente a cada modelo, debido a que se producían errores de convergencia.
- **Métrica.** Hemos decidido escoger la métrica Accuracy para maximizar el número de instancias bien clasificadas.
- **Conjunto Validación.** Usamos la opción de validation_split que nos da Keras para definir el conjunto de validación sobre un porcentaje del conjunto de training, el porcentaje elegido ha sido 10 %.

- **Épocas.** Hemos entrenado durante un máximo de 100 épocas cada modelo.

Finalmente, en el modelo mejorado y en el modelo con Inception modificado, hemos añadido como callback Early Stopping ante la función de pérdida y la métrica. Con una paciencia de 20 épocas y la opción de restaurar los mejores pesos activos.

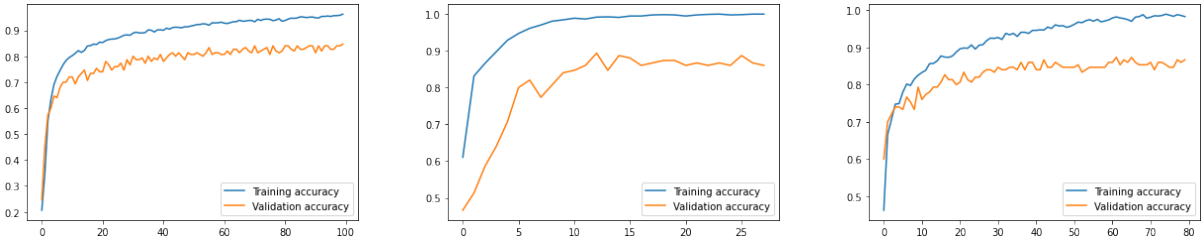
1.3.4. Resultados

Tras entrenar los modelos obtenemos los siguientes resultados:



(a) Densenet121 Simple (b) Densenet121 Mejorado (c) DN + Inception Modificado

Figura 1.10: Función de Pérdida



(a) Densenet121 Simple (b) Densenet121 Mejorado (c) DN + Inception Modificado

Figura 1.11: Métrica Accuracy

	Test Loss	**Test Accuracy**
Modelo Simple	0.4097	0.8500
Modelo Mejorado	0.4884	0.8719
Modelo con Inception Modificado	0.3797	0.8780

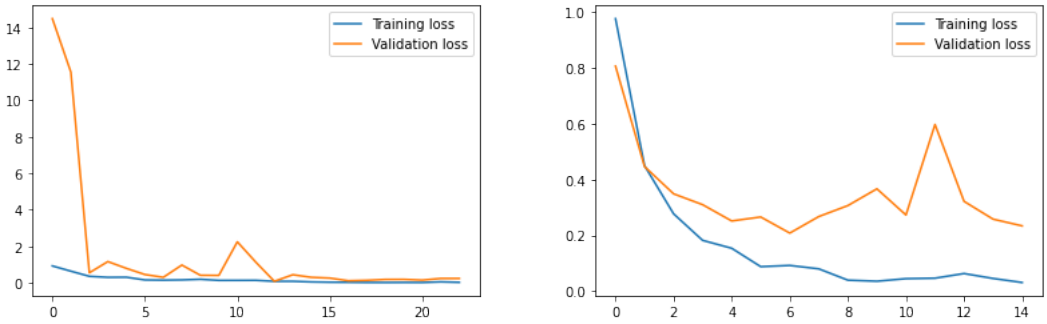
Tabla 1.7: Datos resultantes al evaluar el modelo con el conjunto de Test

Tras observar los resultados de las gráficas, vemos que el mejor modelo es el modelo con Inception modificado, ya que tiene la menor función de pérdida y el mayor valor de accuracy. Además, podemos apreciar que existe una asíntota en 0.9 de accuracy y 0.35 en la función de pérdida. Y por tanto, no vamos a conseguir mucha más mejoría siguiendo con este enfoque. Aplicaremos ahora Fine Tuning.

1.3.5. Fine Tuning

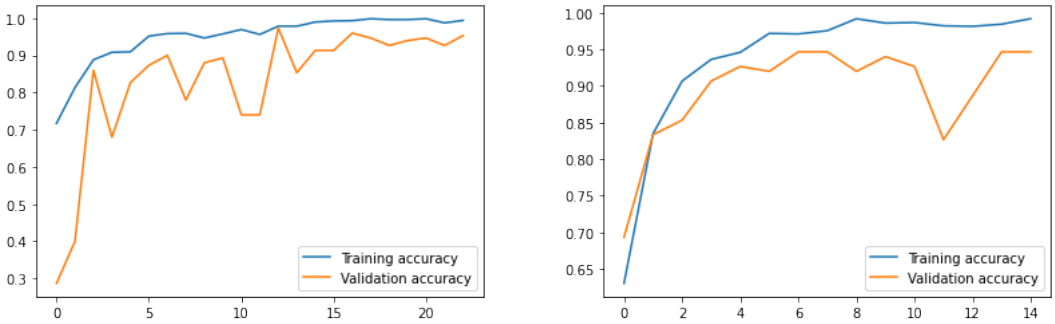
Para intentar traspasar la asíntota que tenemos aplicamoslo que denominamos Fine Tuning, por lo que reentrenamos el modelo completo (incluidas las capas de DenseNet) con nuestros datos del problema. Aplicaremos esta técnica al modelo simple y al modelo con Inception modificado para comprobar la mejoría que existe entre los dos modelos. En esta ocasión, el modelo DenseNet con Inception modificado contendrá un único módulo Inception modificado más la parte secuencial ya definida.

Reentrenamos el modelo simple 30 épocas y el modelo con Inception modificado 15 épocas. No entrenamos el modelo con Inception modificado un número mayor de épocas debido a la complejidad en tiempo del entrenamiento. Finalmente, obtenemos los siguientes resultados:



(a) DenseNet Simple (b) DN + Inception Modificado

Figura 1.12: Función de Pérdida con Fine Tuning



(a) DenseNet Simple (b) DN + Inception Modificado

Figura 1.13: Métrica Accuracy con Fine Tuning

	Test Loss	**Test Accuracy**
Modelo Simple FT	0.3512	0.9120
Modelo con Inception Modificado FT	0.2887	0.9440

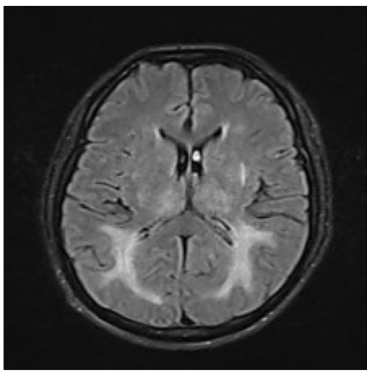
Tabla 1.8: Datos resultantes al evaluar el modelo con el conjunto de Test

Se observa que con Fine Tuning hemos conseguido superar la asíntota claramente definida en 0.9 para accuracy y 0.35 para la función de pérdida.

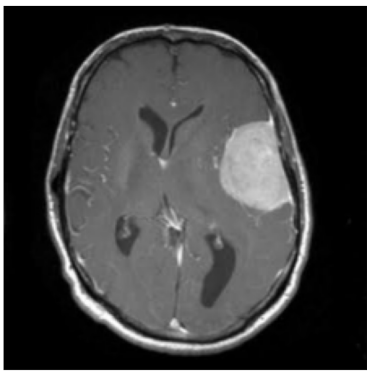
1.3.6. Conclusión

Como solución al problema multiclase hemos conseguido un modelo con 94.4 % de precisión (accuracy) para el conjunto de test. Esto demuestra que estas técnicas de Inteligencia Artificial aplicadas al problema multiclase de detección de tumores cerebrales, son una buena herramienta no sólo para detectar si hay un tumor o no, sino que son capaces también de detectar el tipo de tumor en caso de que lo haya.

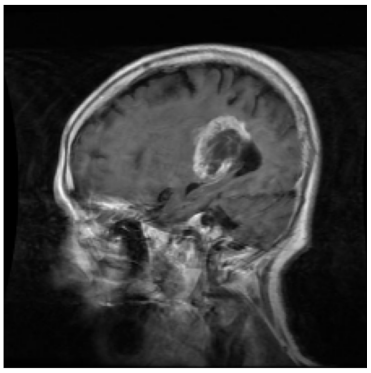
Finalmente, como comprobación de nuestro detector hemos extraído unos ejemplos del conjunto de test y comprobamos observando su etiqueta que efectivamente nuestro detector acierta en su diagnóstico.



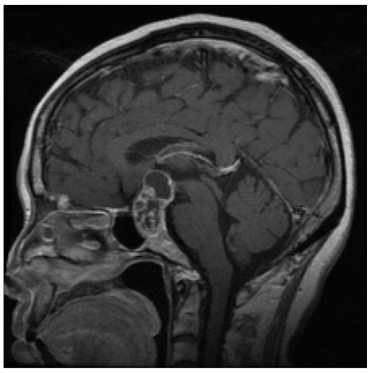
(a) label : (1,0,0,0) [non-tumor]
prediction label :
(9.99e-01, 1.17e-05, 6.89e-06, 1.71e-06)



(b) label : label : (0,0,1,0) [meningioma]
prediction label :
(4.38e-06, 2.29e-04, 9.99e-01, 6.41e-06)



(c) label : (0,1,0,0) [glioma]
prediction label :
(8.32e-05, 9.41e-01, 5.83e-02, 1.69e-05)



(d) label : (0,0,0,1) [pituitary]
prediction label :
(7.08e-07, 4.03e-07, 1.45e-06, 9.99e-01)

Figura 1.14: Resultados en las imágenes.

1.4. Trabajos Futuros

A la hora de que este proyecto sea útil para profesionales médicos con escasos o nulos conocimientos de visión por computador, sería fácil crear un programa utilizando el modelo ya entrenado para ser usado como una caja negra. En la que la entrada sea una imagen cerebral tomada mediante una resonancia magnética, y la salida un diagnóstico tumoral sobre la resonancia. El profesional médico cargaría la imagen tomada del paciente en el programa y se le devolvería la predicción.

En cuanto a las mejoras de la técnica presentada, proponemos dos. La primera sería aumentar el volumen de datos entrenados y la segunda probar con otras arquitecturas como modelo. Para ambas opciones podría ser necesario un computador más potente que el que se dispone para este proyecto.

Para este problema parece interesante de cara al futuro abordar no solo el problema de clasificación de imágenes, sino los problemas de detección y segmentación.

Bibliografía

[1] Sartaj Bhuvaji. *Brain Tumor Classification (MRI)*. https://www.kaggle.com/sartajbhuvaji/brain-tumor-classification-mri. 2020.

[2] Jason Brownlee. *How to Develop VGG, Inception and ResNet Modules from Scratch in Keras*. https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/. 2019.

[3] Christian Szegedy y col. «Inception-v4, inception-resnet and the impact of residual connections on learning». En: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.