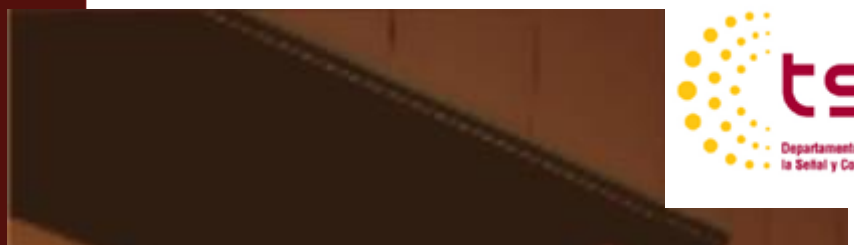
Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Desarrollo de aplicación basada en redes neuronales para detectar imágenes médicas.

Autor: Ismael García Mayorga
Tutor: Alejandro del Real Torres

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

Sevilla, 2023



Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Desarrollo de aplicación basada en redes neuronales para detectar imágenes médicas.

Autor:
Ismael García Mayorga

Tutor:
Alejandro del Real Torres
Profesor titular

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Proyecto Fin de Carrera: Desarrollo de aplicación basada en redes neuronales para detectar imágenes médicas.

Autor: Ismael García Mayorga

Tutor: Alejandro del Real Torres

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

La carrera ha sido una experiencia que recordare con cariño, pero también ha sido dura y lo habría sido mucho más si no hubiese tenido el apoyo de todas las personas que me han ayudado en este proceso. Primero dar las gracias a mi Padre y Madre que me han ayudado incondicionalmente en todo este recorrido, aunque no sean profesores o ingenieros me han enseñado lecciones de vida más importante que la que te puede aportar un libro y estoy seguro de que me ayudarán en mi futuro muchísimo.

Por otro lado, también agradecer a todos los compañeros que he conocido y que me han ayudado a lo largo del camino, gracias a ellos he podido superar muchos inconvenientes que solo no hubiese podido afrontar de la misma manera. Me siento muy orgulloso de las amistades que he formado y es algo que me llevaré para siempre.

Para finalizar, agradecer tanto a Alejandro que ha sido mi tutor del TFG, como Lucía Gálvez que me han orientado durante todo el proceso y han hecho que este proyecto haya sido realizado de la mejor manera. Dar las gracias por haberme brindado la oportunidad de poder introducirme en el mundo de la inteligencia y aprender de manera profunda sobre el tema. Aprecio enormemente la de poder realizar una serie de cursos en la plataforma de Coursera donde he adquirido todos los conocimientos necesarios para poder realizar este TFG.

Resumen

La intención de este proyecto es poder realizar una aplicación de detección de imágenes médicas basadas en redes neuronales. Nuestra aplicación estará orientada a detectar cáncer cerebral en imágenes de resonancia magnética (RMI) y a través de la red neuronal que ha sido entrenada poder generar diagnósticos para los pacientes.

La motivación de este proyector es poder realizar una aplicación que pueda ser desplegada en un entorno real que pueda ayudar a los distintos pacientes a tener una respuesta lo más rápida posible de su enfermedad.

Al principio del Proyecto se realizará una introducción a las redes neuronales, así como una serie de conceptos necesarios para poder entender la aplicación. Una vez comprendido estos conceptos pasaremos a explicar la aplicación y los resultados obtenidos.

Abstract

The purpose of this project is to develop a medical detection application based on neural networks. Out application will be focused on detecting brain cancer in magnetic resonance imaging (MRI) images, and through the trained neural network, it will be able to generate a diagnosis for patients.

The motivation behind this project is to create an application that can deployed in a real-world environment, helping patient receive a prompt response regarding their illness.

At the beginning of the project, an introduction to neural networks will be provided, along with a series of necessary concepts to understand the application. Once these concepts are understood, we will proceed to explain the application and the results obtained.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La inteligencia artificial (IA) nació a mediados del siglo pasado, pero no fue hasta hace pocos años que este campo empezó a tomar gran importancia en la comunidad científica. Si es cierto que durante todo este tiempo se han desarrollado proyectos en distintos ámbitos, pero siempre tenían un delimitador común a la hora de poder desarrollar proyectos a una mayor escala que es la capacidad de computación. Con el gran avance tanto del hardware como del software empezó a ser más viable poder desarrollar todo este tipo de nuevas tecnologías dando lugar a increíbles proyectos basados en IA.

En la actualidad la inteligencia artificial está tomando un papel cada vez más importantes en distintos campos, como la industria, la medicina, la educación y la robótica entre otros. Todo este avance ha permitida el desarrollo de sistemas capaces de tomar decisiones de manera autónoma, lo que ha mejorado significativamente la eficiencia y precisión de muchos procesos. Además, gracias a la IA está siendo utilizada para analizar una gran cantidad de datos de información. Vemos que día a día la cantidad de datos generados está aumentando de una manera exponencial.

Uno de los principales avances ha sido en el campo de la medicina, donde la IA está siendo utilizada para la detección temprana de enfermedades y para el diseño de tratamientos personalizados más efectivos. Gracias al uso de la IA está permitiendo analizar una gran cantidad de información médica ya sea imágenes de TAC o el nivel hormonal de nuestro cuerpo, permitiendo predecir posibles enfermedades que un humano no sería capaz de detectar por el mismo, además, esto mejoraría la detección de forma prematura de muchas de estas enfermedades aumentando el índice de supervivencia de los pacientes en enfermedades como el cáncer. Por estas razones he decidido enfocar mi trabajo de fin de grado en realizar una aplicación médica, más concretamente en una aplicación de detección de cáncer cerebral a partir de imágenes de resonancia magnética.

2 MOTIVACIÓN DE LA APLICACIÓN

La aplicación nace de la necesidad de poder ayudar a los pacientes a prevenir de forma prematura el cáncer cerebral. Este tipo de cáncer representa el 2% de todos los tumores diagnosticados en todo el mundo. Sin embargo, los tumores cerebrales metastásicos, es decir, aquellos que se originan en otra parte del cuerpo y se propagan al cerebro, son más comunes. Se estima que alrededor del 10% a 30% de las personas con cáncer desarrollan tumores cerebrales en algún momento durante el curso de su enfermedad. No tenemos que obviar que este tipo de tumores son muy agresivos y tienen una alta tasa de mortalidad

¿Qué podría aportar la aplicación

Esta aplicación podría ser utilizado en distintos ámbitos de la sanidad. Uno de ellos sería trabajar de manera conjunta con un médico donde ambos podrían discernir en caso de duda un posible resultado sobre el diagnóstico del paciente.

Otro posible uso sería el poder utilizar la aplicación para poder realizar el diagnostico sin necesidad de un médico, mejorando enormemente el tiempo de espera de los resultados de una prueba. Por desgracia, existen muchos países donde el sistema de sanidad se encuentra totalmente colapsado con grandes colas de espera y por ello una aplicación que automatice la resolución de estos problemas puede ayudar enormemente a reducir estas enormes colas de espera.

Si bien es cierto que la gran mayoría de personas aún se encuentran reacias a utilizar este tipo de tecnologías, demostraremos que son muy útiles para este tipo casos.

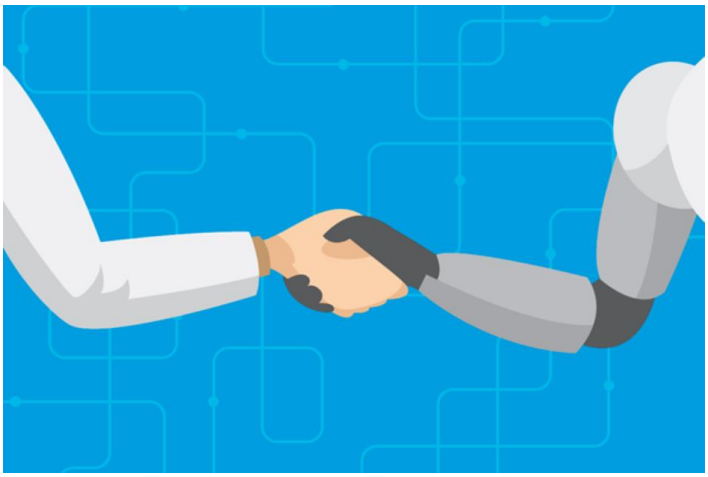


Figura 2-1. IA y Humanos.

3 CONOCIMIENTOS PREVIOS Y ADQUIRIDOS

Los conocimientos necesarios adquiridos para el desarrollo de esta aplicación los he obtenido con la realización de una serie de cursos en la plataforma Coursera impartidos por Andrew Ng. Este curso se llama: Programa especializado: Deeplearning.

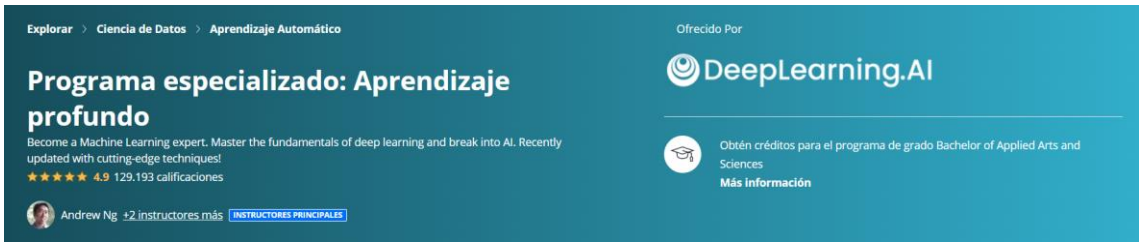


Figura 3-1. Foto del curso principal.

Este programa se divide en 5 cursos, en cada uno de ellos se imparte distintas materias. Cada curso se divide en distintos modulo y cada módulo la explicación se realiza mediante videos, al final de cada módulo hay unas actividades que deberemos superar ya sea test o ejercicios de programación donde pondremos en práctica todo lo aprendido. La duración media de cada curso es de unas 18 horas. Pasamos a describir lo que se imparte en cada uno de ellos.

1. **Neural Networks and Deeplearning:** curso inicial donde se explican los conceptos básicos, estructura y funcionamiento de una red neuronal. Añade ejercicios básicos en Python para programar de forma muy sencilla una red neuronal y aplicar forward propagation y back Ward propagation.
2. **Improving Deep Neural Networks: Hyperparameter tunning, Regulazation and Optimization:** En este curso se explican conceptos más complejos de las redes neuronales como los distintos algoritmos de optimización, los principales problemas que podemos encontrar a la hora de enfrentarnos al entrenamiento de una red y como solucionar estos problemas. En este curso se realiza una introducción a Tensorflow basado en Keras. Este es un Framework orientado al trabajo con redes neuronales, más adelante en la explicación de las tecnologías del proyecto profundizaremos en el funcionamiento de este.
3. **Structuring Machine Learning projects:** En este curso se realiza una introducción a cómo afrontar un proyecto de Deep Learning a gran escala. Entre otras cosas explica como distribuir los datos de entrenamiento, los problemas que con lleva una mala distribución o distintas técnicas que podemos aplicar si tenemos un conjunto de datos de entrenamiento con error o si es demasiado pequeño. Se nos plantean una serie de actividades orientadas a proyectos reales donde se nos expone un problema y nosotros tenemos que dar una posible solución a dicho problema.
4. **Convulational neural Networks.** Curso especializado en la arquitectura de redes convolucionales. Realiza una explicación extensa del funcionamiento de las redes convolucionales, como funcionan y se nos presentan distintas arquitecturas reales. Por otro lado, mediante cursos de programación implementares todas estas redes utilizando el framework Tensorflow que es muy útil para este tipo de redes. Para el propósito de nuestro proyecto este es uno de los cursos más importantes, ya que nuestra aplicación se basa en redes convolucionales.
5. **Sequence Models:** Introducción a las redes convolucionales, explicación del funcionamiento, arquitectura y un entendimiento profundo de las distintas aplicaciones que se pueden lograr con este tipo de redes. Es un curso muy importante para el ámbito de redes neuronales pero nuestro proyecto no se basa en el uso de estas, sin embargo, ofrece muchos conocimientos que podría aplicar en otros proyectos.

4 REDES NEURONALES

Antes de comenzar con el desarrollo de la aplicación vamos a realizar una introducción al aprendizaje máquina y al Deeplearning. Nuestra aplicación se fundamenta en el uso de Deeplearning para poder realizar las predicciones de los diagnósticos. Este es una forma de aprendizaje automático que se basa en el uso de redes neuronales. Deeplearning es un subconjunto dentro del Machine Learning que estes a su vez pertenece a un todo mayor que es la inteligencia artificial.

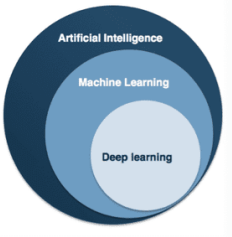


Figura 4-1. Representación división de la inteligencia artificial.

Una red neuronal puede llegar a ser muy compleja y poder entenderla profundamente puede llegar a ser complicado. Por ello realizaremos una introducción para que sea más fácil entender el funcionamiento de nuestra aplicación y poder comprender el por qué las decisiones tomadas en su diseño.

4.1 Estructura de una red neuronal

La arquitectura de una red neuronal y como esta funciona es el principal motivo por el que son capaces de realizar tareas tan complejas.

La unidad mínima en la arquitectura de red neuronal es la neurona, también conocida como nodo.

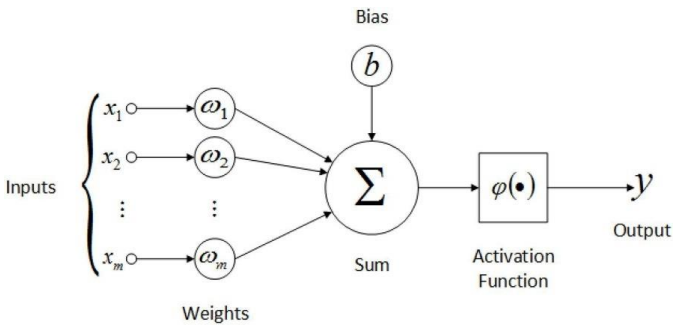


Figura 4-2. Neurona.

Cada una de estas neuronas estas neuronas están compuestas por:

- Una o varias entradas (inputs) que representan las distintas características de los datos de entrada a una neurona
- Pesos (weights) es un parámetro que representa la importancia de cada una de estas características, indican si una entrada será más relevante para el entrenamiento que otra.

- Sesgo (bias) es un parámetro aditivo en la función de la neurona que permite ajustar el umbral de activación, permitiendo a la neurona saber cuándo activarse.

Nuestro objetivo principal a la hora de entrenar una red neuronal será la de ajustar el valor de estos parámetros, para obtener el valor que mejor funcionamiento brinda a nuestra aplicación. Estas entradas y los parámetros están relacionados en una ecuación lineal.

$$Z = \sum_{i=1}^{m} x_i \times w_i \quad + b$$

Esta función lineal introducirá en una función de activación. El objetivo de esta función será introducir no-linealidad en la salida de las neuronas y permitir que la red pueda aprender relaciones no-lineales. Esta función determinará si la salida de una neurona deberá propagarse a la entrada de la siguiente. Se puede representar la salida de una neurona como la siguiente formula.

$$Y = f(Z) = f(\sum_{i=1}^{m} x_i \times w_i \quad + b)$$

Existen muchas funciones de activación como podría ser: Sigmoid, gaussiana, Tanh o Softmax entre otras.

CAPAS

Cada una de estas neuronas se apilarán en distintas capas y están interconectadas a cada una de las neuronas de la siguiente capa. Cada capa representa un nivel de abstracción, en cada capa se aprenderán características de nuestros datos. En las capas iniciales se aprenderán las características más sencillas y según vamos profundizando en la red se irán aprendiendo cada vez características más complejas.

El número de neuronas y la cantidad de capas son hiperparámetros elegido por el que ha diseñado la arquitectura de la red. En cada capa puede tener un número de neuronas distintas a la anterior.

CAPA FINAL

La capa final es un poco diferente a las capas anteriores. Esta es la capa encargada de realizar la predicción final de nuestro modelo. Esta capa estará compuesta por tantas neuronas como clases queramos predecir. Por ejemplo, en nuestro caso queremos predecir 4 posibilidades que son: no tener tumor o en caso de tener cual es el tumor que tiene el paciente. Cada una de estas neuronas estará asignada a cada una de las clases

¿Cómo saber cuál ha sido la predicción obtenida?

Al igual que las capas anteriores la entrada se realiza una serie de operaciones matemáticas, tras esto se pasa

por una función de activación que en este nos daría el resultado final. El resultado de dicha función es la probabilidad de acierto de cada una de las clases, el mayor valor representará la clase resultado de la predicción

De forma general existen 2 tipos de funciones de activación utilizadas para realizar la predicción de esta capa final.

ESTRUCTURA FINAL

La arquitectura de una red neuronal será la unión de estas neuronas con un número determinado de capas.

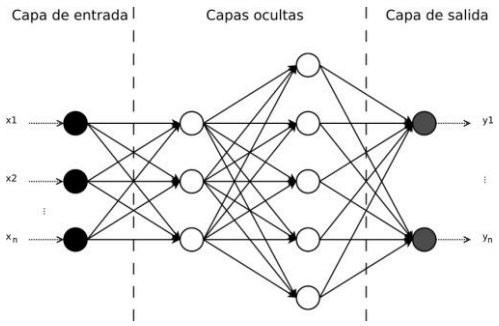


Figura 4-3. Estructura de una red neuronal.

Se dividen en 3 partes fundamentales:

1. Capa de entrada (input layer): Esta será la capa inicial donde se introducirán nuestros datos de entrada, cada una de las neuronas representan cada una de las características iniciales de nuestros datos. Es muy común en muchos casos realizar un preprocesado de los datos para poder permitir a nuestra red un funcionamiento más sencillo

2. Capas ocultas (hidden layers): Las capas ocultas son el núcleo de la red neuronal y son las responsables de procesar los datos de entrada y transformarlos en representación más abstracta y útiles. A medida que la información se propaga a través de las capas ocultas, las neuronas realizan cálculos matemáticos para ajustar el valor de los pesos y sesgo. Como ya se menciona anteriormente cada una de estas capas representan un nivel de abstracción y a medida que profundizamos en nuestra red cada una de estas capas ocultas atenderán a características más complejas de nuestros datos. La cantidad y estructura de las capas ocultas son aspectos importantes en el diseño de la red neuronal e influyen enormemente en el funcionamiento de esta.

3. Capa salida (output layer): Esta es la última capa de la red neuronal y es responsable de general la predicción final de la red. Cada neurona en la capa de salida representa cada una de las distintas clases o valores posibles que puede tomar la predicción del algoritmo. La neurona con el mayor valor representará cual ha sido la predicción de la red.

4.2 APRENDIZAJE DE UNA RED NEURONAL

Para explicar el funcionamiento del aprendizaje cabe destacar que nosotros nos centraremos en el aprendizaje supervisado. En este tipo de aprendizaje se basa en que tenemos unos datos de entrada y las salidas de las predicciones que dichos datos deberían de producir. EL aprendizaje de una red neuronal tiene dos procesos claramente diferenciados uno es forward propagation y el otro es backward propagation. Estos procesos de entrenamiento se repetirán un número de veces determinado, este número se conoce como epoch. El objetivo de estos procesos será el de ajustar el valor de los distintos parámetros de nuestra red.

4.2.1 FORDWARD PROPAGATION

La propagación hacia adelante (Forward propagation) es el proceso en el que se basa el funcionamiento de una red neuronal artificial. El proceso de propagación hacia adelante comienza con la alimentación de los datos de entrada.

Cada neurona en la capa correspondiente recibe unos inputs y realiza las operaciones matemáticas previamente explicadas hasta obtener una salida, la cual será enviada a la siguiente neurona. Este proceso se repetirá hasta alcanzar la capa final.

Una vez en la capa final el valor de la última salida corresponderá con la predicción de nuestro algoritmo. Este resultado de la predicción se comparará con el valor esperado de este. La diferencia entre las dos salidas se conoce como error que será utilizado en el backward propagation para ajustar los parámetros. La función de error o perdidas es la siguiente.

Función de perdidas:

$$L(\hat{Y}, Y) = -(Y \times \log(\hat{Y}) + (1 - Y) \times \log(1 - \hat{Y}))$$

Esta función define cuanto se parece nuestra salida $\hat{Y}$ resultado de la predicción con la salida espera Y.

Este proceso de entrenamiento se repetirá m ejemplos de entrenamiento, calcularemos el promedio de todos los resultados de la función de perdidas. Este promedio se conoce como función de coste.

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{Y}i, Yi)$$

Una vez calculado la función de coste pasaremos al segundo paso del entrenamiento que será el backward propagation.

4.2.2 BACKWARD PROPAGATION

La propagación hacia detrás ocurres después de la propagación hacia adelante donde hemos calculado el valor de la función de coste. El objetivo será ajustar los valores de los parámetros para intentar conseguir el menor valor de la función. Existen distintos tipos de algoritmos que se aplican en esta fase, pero el más conocido es el gradient descent.

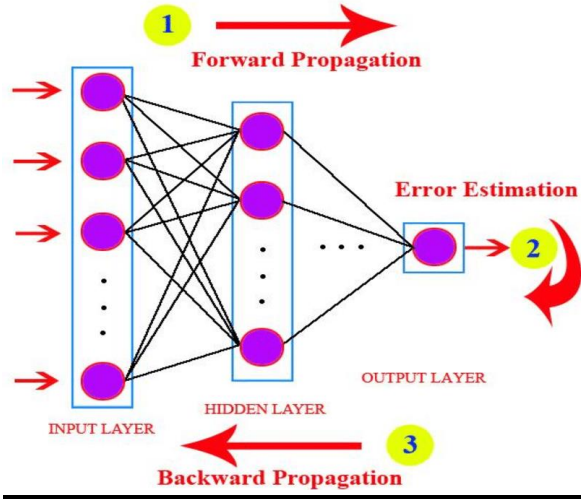


Figura 4-4. Proceso de aprendizaje.

Gradient Descent

El descenso de gradiente o gradient Descent es uno de los algoritmos más utilizados en backward propagation. Nuestro objetivo será el de ajustar los parámetros del modelo. Para ello realizaremos la derivada de la función de coste respecto a cada parámetro para calcular la dirección y tasa de cambio optimas en la que los parámetros deben actualizarse para minimizar la función de coste. La fórmula básica es la siguiente:

Para w (peso)

$$w = w - \alpha \times \frac{\varphi J(w,b)}{\varphi w}$$

Para b (bias)

$$b = b - \alpha \times \frac{\varphi J(w,b)}{\varphi b}$$

Donde:

- α Representa la tasa de aprendizaje o learning rate. Determinara la magnitud de los cambios de los parámetros. El learninig rate es un hiperparámetro cuyo valor es elegido por nosotros o también puede ser objetivo de entrenamiento.

En cada una de las iteraciones de este algoritmo se irán actualizando los valores de los parámetros de toda la red neuronal. Podemos representar la relación entre J, W y B como una función convexa. Una de las características principales de este tipo de función es que solo existe un mínimo óptimo. Nuestro objetivo será alcanzar dicho mínimo desplazando tasa de aprendizaje en cada iteración del algoritmo.

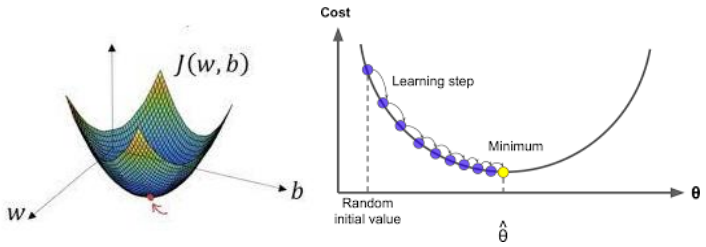


Figura 4-5. Gradient Descent.

4.3 PROBLEMAS EN EL ENTRENAMIENTO Y COMO SOLUCIONARLO

Una vez comprendido los conceptos básicos de como entrenar una red, nuestro siguiente objetivo será entrenarla. Lo más probable que cuando intentemos entrenar cualquier modelo no funcione tan bien como nos esperábamos o que consigamos un buen funcionamiento en el entrenamiento y un pobre rendimiento en la fase de test. Nuestro siguiente objetivo será entender a qué se debe este y como solucionarlo. Es importante poder comprender estos problemas ya que muchos han ocurrido en el desarrollo de nuestra aplicación y será necesario entender cada uno de ellos.

4.3.1 ¿Qué es el overfitting (sobre ajuste) y como solucionarlo?

El overfitting, también conocido como sobreajuste, es un problema bastante común en el aprendizaje de una red, esto se debe a que nuestro modelo se ajusta demasiado a los datos de entrenamiento y no es capaz de generalizar bien a nuevos datos.

El overfitting ocurre cuando un modelo es demasiado complejo en relación con la cantidad de datos de entrenamiento disponibles. Esto provocaría que el modelo memorice estos datos y no sea capaz de aprender patrones nuevos, provocando en un buen funcionamiento en el entrenamiento y en un mal funcionamiento en el test.

Existen varias medidas que podremos utilizar frente al overfitting que son:

- Recopilar más datos de entrenamiento, esto puede ayudar a reducir la complejidad relativa del modelo en relación con la cantidad de datos disponibles. Sin embargo, en muchos casos es muy complicado encontrar nuevos datos de entrenamiento por lo que se optaría en aumentar los que ya tenemos. Un ejemplo claro que se aplica en nuestra aplicación es aumentar el número de imágenes médica

realizando rotaciones, ajustes de tamaño, cambios de brillo y contraste. Permitiendo el modelo generalizar aún más y reduciendo el overfitting.

- Simplificar el modelo: reducir la complejidad del modelo ayudará a reducir el overfitting. Esto puede lograrse reduciendo el número de características utilizada en el modelo, disminuyendo la complejidad de la función de perdidas o reduciendo el número de capas.
- Regulación: Se pueden utilizar técnicas de regulación, como la regresión L1 y L2, para penalizar los coeficientes del modelo y reducir la complejidad de este.
- Normalización o preprocesamiento de datos: Otras de las técnicas más utilizadas para reducir el overfitting es realizar una normalización o un preprocesado de los datos. En el caso de las imágenes podríamos reducir el ruido, recortar zonas que no son de interés o cambiar el tamaño de la imagen para que se ajuste mejor a la capa de entrada del modelo.

4.3.2 Vanishing Gradient y Exploding gradient

Uno de los problemas principales al entrenar redes neuronales muy profundas puede causar que el gradiente alcance valores muy pequeños cercanos al cero provocando que la red sea muy complicada de entrenar, otra posible situación seria justo la contraría que tuvieses un valor del gradiente demasiado grande provocando que nuestra red no sea estable, estos dos fenómenos se conocen como Vanishing Gradient y Exploding Gradient:

Para el fenómeno del Vanishing Gradient el valor del gradiente alcanza valores casi nulos esto se debe fundamentalmente por la naturaleza de las funciones Sigmoid y ReLu. En ambos casos este problema surge durante la propagación hacia atrás donde se calcula las derivadas de las funciones desde la capa final hasta la inicial. Para el caso de la función Sigmoid su derivada suele dar valores pequeños o valores muy grandes pero este problema ocurre cuando alcanza valores muy pequeños. Si se suma que aplicando la regla de la cadena este valor se iría multiplicando por la derivada correspondiente en cada capa, conseguiríamos que se redujese su valor cada vez más y más.

Una posible solución a este problema sería cambiar la función de activación de Sigmoid por ReLu. Esta elección puede arreglar un poco el problema, pero no lo solventa totalmente. La función ReLu como ya sabemos nos devuelve 0 en caso negativo o el valor del input en caso de valor positivo, esto puede desencadenar en que numerosas entradas negativas hagan que la salida siempre sea 0 y el resultado final sería algo parecido a lo que ocurre con la función Sigmoid.

El principal problema que produce el vanishing gradient es que la red haga actualizaciones de los parámetros muy pequeños o prácticamente se deje de actualizar, provocando que la red dejase de aprender. Para solucionar este problema una posible solución es aplicar conexiones residuales, concepto que profundizaremos más adelante.

Para el fenómeno del exploding gradient el problema es el contrario, el valor del gradiente alcanza valores demasiado grandes que hace que nuestra red sea inestable. Esto puede ocurrir debido por varias cosas. Una de ellas es por la función Sigmoid que sus derivadas pueden alcanzar o valores muy pequeños o grandes, en este caso se da cuando alcanza valores muy grandes.

AL igual que antes durante el proceso de propagación hacia detrás, cada vez que fuésemos propagan el gradiente a una capa inferior provocaría que cada una de estas multiplicaciones aumentase desorbitadamente el valor del gradiente. Otro caso posible seria inicializar el valor de los parámetros con un valor demasiado grande que básicamente provocaría lo mismo. Otro factor crucial que provoca este problema sería tener valor del learning rate demasiado grande que provocaría que los parámetros se actualizasen demasiado provocando que el valor del gradiente alcanza valores enormes.

Para solucionar este problema existen técnicas de normalización o uso de umbrales que establecen un tamaño máximo que podría alcanzar el gradiente.

4.4 TRANSFER LEARNING Y FINE TUNNING.

Uno de los principales problemas que nos encontramos a la hora de entrenar una red neuronal desde cero es la capacidad computacional. Cuando nos encontramos frente a una arquitectura muy compleja entrenar desde cero a veces supone un gran reto y no todo el mundo tiene la infraestructura necesaria para poder entrenarlas. Otro gran problema que también puede sumarse a este anterior es que, en muchos casos, tenemos un conjunto de datos de entrenamiento muy pequeño haciendo que nuestro algoritmo no sea capaz de aprender características nuevas. Frente a estos tipos de problemas se aplica una técnica conocida como: transfer learning.

En el transfer learning utilizaremos el valor de los parámetros previamente entrenados en una red que tenga la misma arquitectura a la nuestra o comparte en gran medida su arquitectura. Esta red habrá sido entrenada para una aplicación parecida a la nuestra. Un ejemplo claro de esto se verá más adelante en nuestra aplicación donde hemos aplicado esta técnica, donde cargamos los datos de entrenamiento de una red diseñada para la detección de imágenes. Ya que nuestra red también debe reconocer imágenes las características comunes y más básicas a la hora de detectar una imagen se comparten entre redes.

Otra técnica muy utilizada a la hora de entrenar redes neuronales es fine tunning. Esta técnica se basa en entrenar los parámetros de las últimas capas. Este método se basa en la abstracción de las capas, nuestra intención será entrenar las capas finales de nuestra red dejando congeladas las capas iniciales encargadas de las características más simples. Estas capas iniciales no merecen la pena utilizar la cantidad de recursos necesarios en entrenarlas de nuevo cuando la mayoría de estas capas han sido previamente entrenadas en aplicaciones que comparten características con nuestra red. Por eso nos centraremos únicamente en entrenar las capas encargadas, de las funciones más complejas de nuestra arquitectura.

Las dos técnicas suelen utilizarse de manera conjunta donde en una instancia inicial utilizaríamos transfer learning para precargar el valor de los pesos de todas las neuronas y aplicaríamos fine tunning a las capas finales. En nuestra aplicación hemos aplicado el uso de ambas técnicas.

4.4 TIPOS DE REDES NEURONALES.

Para finalizar pasaremos a realizar una clasificación de los distintitos tipos de redes que existen, cada una de ellas tienen sus puntos fuertes y sus puntos débiles. Cada aplicación requerirá la red que más se le adecue.

Redes Neuronales Feedforward (FNN): son las redes neuronales más básicas y comunes, en la que la información fluye en una sola dirección, sin ciclos ni bucles. Estas redes son utilizadas para tares de clasificación y predicción.

Redes Neuronales Recurrente (RNN): En este tipo de redes, las neuronas están conectadas en forma de bucle, permitiendo que la información fluya en varias direcciones, incluyendo hacia atrás, lo que permite la retroalimentación. Estas redes son útiles en tares de procesamiento de lenguaje natural y predicciones de series temporales.

Redes Neuronales Convolucionales (CNN): Son redes que se utilizan principalmente para tareas de visión artificial, en las que se procesan imágenes y videos. Estas redes están diseñadas para procesar datos con una topología de cuadrícula, como las imágenes. Su capacidad para detectar características en diferentes niveles de abstracción las hace muy útiles para la clasificación de objetos y la detección de patrones en imágenes.

Para nuestra aplicación se ha utilizada las redes convolucionales ya que es la que más se ajusta a lo que queremos a realizar. Su funcionamiento es parecido al que ya hemos explicado, pero vamos a profundizar más en las diferencias fundamentales.

5 REDES CONVOLUCIONALES

Existen varios tipos de redes neuronales, pero vamos a centrarnos en las redes convolucionales. Este tipo de red presenta una arquitectura distinta a las redes neuronales básicas, pero de forma general el funcionamiento y como estas aprenden siguen la metodología anteriormente explicada. Uno los principales usos de las redes convolucionales son la detección de imágenes y patrones en estas, por eso hemos elegido este tipo de red para nuestra aplicación.

5.1 Bases de una red convolucional.

Las redes convolucionales se basan en el uso de la operación matemática convolucional que más adelante pasaremos a su explicación. Los estados en cada capa están organizados según una estructura de cuadrícula. Estas relaciones son heredadas de una capa a la siguiente, cada valor de las distintas características se basa en una pequeña región espacial de la capa anterior.

Cada capa en la red tiene una estructura tridimensional, tienen una altura (height), anchura (wight) y una profundidad (Depth). Esta profundidad no hace referencia al número de capas que tiene una red, si no que se refiere al número de canales de cada capa. Utilizaremos el termino canal (channel) para referirnos a esta profundidad y no causar confusión.

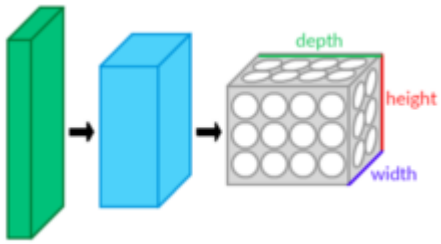


Figura 5-1. Dimensionalidad capas de red convolucional.

La red neuronal convolucional funciona de una manera parecido a como lo haría una red de alimentación directa tradicional (FNN), excepto que cada capa realiza un tipo de operación, al contrario, que en FNN donde todas las capas realizaban las mismas operaciones. Los tres tipos de capas que suelen estar presentes en una red neuronal convolucional son: capa de convolución, pooling y ReLu. La capa ReLu funciona de una manera parecida a como se haría en una FNN. Además, al final de la red existirán una serie de capas totalmente interconectadas que siguen una estructura parecida a la red tradicional. En estas capas se realizarán las predicciones finales de la red. Estas capas suelen disponerse de una determinada manera que explicaremos más tarde, primero definiremos el funcionamiento de estas capas por separado.

5.2 CAPAS EN UNA RED CONVOLUCIONAL

5.2.1 Capa de convolución

La funcionalidad de la capa de convolución es extraer las características o patrones relevante de los datos de entrada. Esto se consigue mediante la aplicación de filtros. Estos filtros son matrices pequeñas que se desplazan por toda la imagen, y en cada posición se realiza la operación matemática de la convolución. El resultado de esta operación es un mapa de características que resalta las partes de la imagen donde se encuentra el patrón representado por el filtro.

Cada uno de estos filtros se encargará de detectar características distintas en cada una de las capas. En las capas iniciales donde se intenta detectar las características más simples de los datos que en este caso es una imagen, los filtros se encargarían de detectar patrones simples como podrían ser líneas verticales u oblicuas. A medida que avanzamos en la red, las características que deberemos detectar serán cada vez más complejas, en este caso los filtros empezarían a detectar patrones más complejos como podría ser un ojo, una boca o una rueda.

El valor que toma cada uno de estos filtros son los parámetros que deberemos de entrenar en nuestra red.

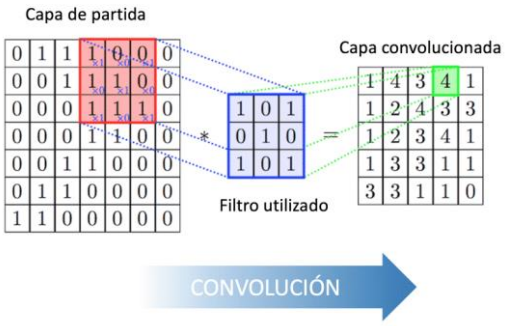


Figura 5-2. Convolución.

5.2.2 Capa de ReLu

La capa Relu es similar a la que nos encontrábamos en las redes tradicionales (FNN). La capa realiza la activación del mapa de características para poder obtener sus propiedades más importantes, la fórmula que la describe es la siguiente.

$$F(x) = \max(threshold, x)$$

La capa compara el valor de entrada si este supera el threshold el valor de salida será x, que se utilizó como entrada. El valor normalmente de este umbral suele ser 0 para eliminar de esta forma los valores negativos del mapa de activación anterior.

Al aplicar esta capa ReLu no cambia la dimensión del mapa de características de entrada. En las redes tradicionales, la función de activación se combina con una transformación lineal con una matriz de pesos para crear la siguiente capa. De forma parecida, en las redes convolucionales normalmente tras una capa ReLu se suele aplicar una capa de convolución.

5.2.3 Capa Pooling

La capa pooling (pooling layer) es la encargada de reducir la dimensionalidad de las características extraídas por las capas de convolución, reduciendo el número de parámetros en la red, ayudando a reducir el overfitting.

La capa de pooling opera en regiones pequeñas de la imagen y reduce la información a un solo valor. Existen dos tipos principales de capas pooling.

- Average Pooling = este tipo de capa extrae la media de la región y reducir su tamaño.
- Max Pooling = En este caso la capa extrae el valor máximo de dicha región.

Ambas técnicas tienen el mismo objetivo, reducir la dimensión de los datos de entrada y conservar la información más relevante para el siguiente paso de la red. Estas capas ayudan a reducir el coste computacional, ya que reduce el número de parámetros de una red.

5.2.4 Capas totalmente conectas (Fully connected Layers)

Las capas totalmente conectadas (Fully connected Layers) son las capas finales en la arquitectura de una red convolucional. Su funcionamiento es similar al de cualquier capa de una red tradicional. Cada neurona está conectada con cada una de las neuronas de la siguiente capa. Estas capas extraen las características obtenidas en las capas de convolución.

Normalmente como función de activación suele utilizarse ReLu para introducir no-linealidad en el modelo y para la capa final funciones de activación binaria como podría ser sigmoid o la más utilizada que es softmax para clasificación múltiple.

5.2.5 Interconexión de las capas

La distribución de cada una de las capas dependerá del tipo de arquitectura, pero de forma general siguen una distribución similar en todas.

Normalmente la primera capa es una capa de convolución, está casi siempre es seguida de una capa ReLu para introducir no-linealidad en el modelo. Esta estructura se repetirá varias veces. Es común a la hora de diseñar una red no indicar que después de una capa de convolución se introduce una capa ReLu, ya que se supone que siempre será así.

Convolución->ReLu->Convolución->Relu->convolución->ReLu

Tras esta estructura se introduciría una capa de pooling para extraer estas características generales más importantes obtenidas hasta ahora y se repetiría la estructura varias veces.

Convolución->ReLu->Convolución->Relu->convolución->ReLu->pooling->Convolución->ReLu->Convolución->Relu->convolución->ReLu->pooling->Convolución->ReLu->Convolución->Relu->convolución->ReLu->pooling.

Esta es la estructura general, al final se conectarán a unas capas totalmente conectadas para extraer las características generales y realizar la predicción final.

Cabe destacar que el tamaño de los filtros, el número de desplazamiento o el tamaño del pooling será distinto en cada una de las arquitecturas de una red CNN.

6 ARQUITECTURA DEL MODELO

6.1 Selección de una arquitectura

La elección de una arquitectura u otra definirá enormemente le funcionamiento de un proyecto de redes neuronales, esta elección es uno de los puntos más críticos del proyecto. Como ya sabemos el entrenamiento de una red neuronal supone un gran coste computacional, en nuestro caso hemos utilizado Google Colab como ya se mencionó anteriormente como el entorno de desarrollo. Aunque Google Colab nos ofrezca recursos suficientes para realizar muchas tareas para el caso de entrenar una red se quedaba corto. Se han estudiado 3 arquitecturas para nuestro proyecto que han sido:

1. U-Net: Es una arquitectura bastante interesante para la detección de imágenes, pero fue descartada principalmente a que este tipo de arquitectura esta más orientada a la segmentación de imágenes. Podría dar un buen funcionamiento en nuestra aplicación, pero tras estudiarla profundamente no ha sido elegida.

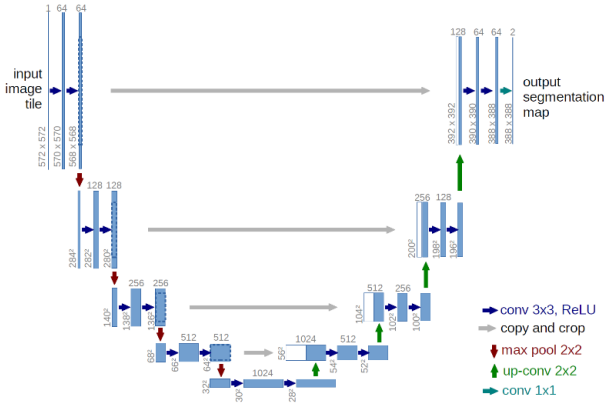


Figura 6-1. Arquitectura U-Net.

2. Vgg16: La siguiente arquitectura que se planteó utilizar fue Vgg16, está compuesta por 138 millones de parámetros y 16 capas. El problema que ofrecía esta arquitectura es que para poder obtener un resultado óptimo teníamos que realizar un largo entrenamiento, esto principalmente se debe a la estructura de la arquitectura en si dificulta el aprendizaje.

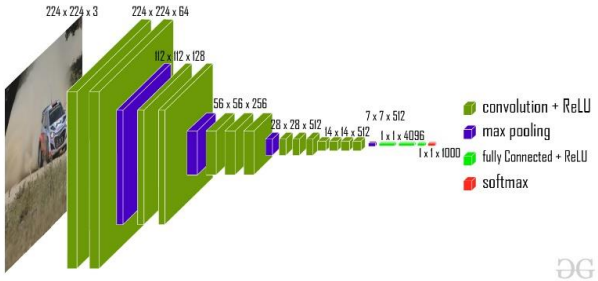


Figura 6-2. Arquitectura Vgg16.

3. ResNet50: La arquitectura está compuesta por 175 capas y 24 millones de parámetros. Más adelante explicaremos en profundidad la arquitectura, pero está arquitectura implementa lo que se conoce como conexiones residuales que permite que la información se propague de forma más sencilla. Aunque esta arquitectura tiene una mejor arquitectura, los recursos que ofrecía Google Colab casi superaban el máximo que nos permitía usar, hemos tenidos que ir ajustándolo para que podamos entrenar la red. En la siguiente imagen puede verse el uso de recursos utilizados de Google colab.

Figura 6-3. Uso de recursos ResNet50.

La elección de estas arquitecturas no se ha seleccionado de forma aleatoria. Primero se ha realizado un estudio previo de distintos artículos, donde hemos intentado buscar aplicaciones de detección de imágenes médicas para entender cuáles son las mejores arquitecturas para nuestro proyecto. Papers estudiados previamente:

- *Detection Of Brain Tumor Using Deep Learning. Hamma Rafiq and Samy S. Abu Naser*
- *Automatic Brain Tumor Detection and Segmentation Using U-Net Bases Fully Convolutional Network. Hao Dong, Guant Yang, et al*
- *Brain Tumor segmentation and grading of lower-grade glioma using deep learning in MRY. Mohamed A. Naser, M. Jamal Deen*

6.2 ResNetV50

ResNetV50 fue diseñada en 2015 por un equipo de investigadores de Microsoft Research Asia, liderado por Kaiming He, en el año 2015. La arquitectura se basa en el uso de conexiones residuales para facilitar el entrenamiento y mejorar el rendimiento en tareas de clasificación de imágenes. Es una de las arquitecturas más utilizadas en el desarrollo de aplicaciones de redes neuronales, esto se debe fundamentalmente al buen rendimiento que aporta y no necesitando una gran cantidad de recursos para ser entrenadas como otras redes más profundas. Pasaremos a explicar en profundidad su arquitectura y cómo funciona.

6.2.1 Conexiones residuales.

¿Por qué utilizar conexiones residuales?

Como ya sabemos el uso de redes demasiada profundas puede provocar grandes problemas a la hora del aprendizaje, uno de estos problemas que ya hemos hablado anteriormente ha sido el problema del vanishing gradient y exploding gradient. Como forma de recordatorio estos fenómenos provocaban que se alcanzasen valores o muy pequeños del gradiente o demasiado grandes provocando un funcionamiento muy pobre de nuestra red. Es por ello por lo que se desarrolló las conexiones residuales para entre otras cosas solucionar muchos de los problemas que implica una red tan profunda.

Se realizo una comparativa entre redes que usan conexiones residuales y redes que no las usan, esta comparación fue realizada por los creadores de la red en el artículo: *Deep Residual Learning for Image*

Recognition. En este articulo demuestra que al aumentar el número de capas de una red provocaría que el error aumentase de forma progresiva a diferencia de la arquitectura ResNet que se beneficiaba del aumento de las capas de la red provocando un error menor. En la siguiente imagen extraída de dicho artículo se puede ver perfectamente.

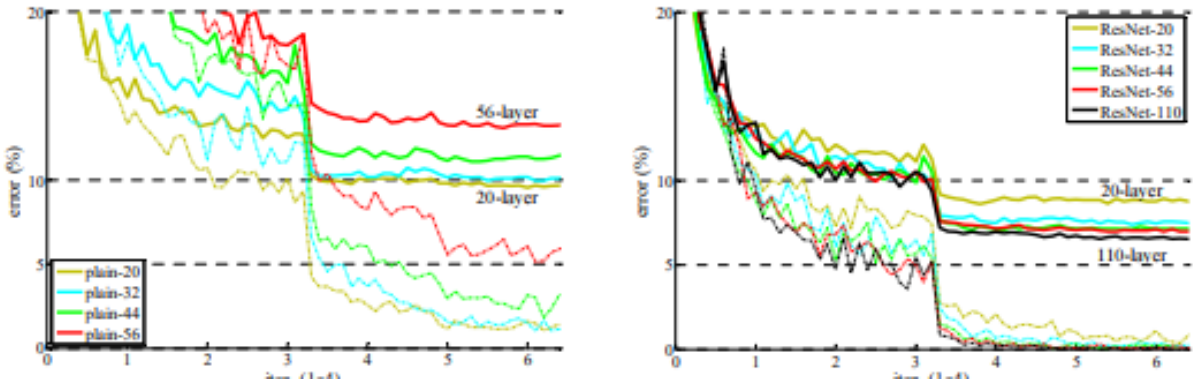


Figura 6-4. Error con y sin conexiones residuales.

A la izquierda tenemos una red plana sin conexiones residuales y a la derecha con conexiones residuales (ResNet). Como se puede observar en la red que no implementa conexiones residuales, a medida que crece el número de capas también aumenta el error, a diferencia de ResNet que se beneficia de esto reduciendo el error.

¿Cómo funcionan las conexiones residuales?

El principal funcionamiento de una conexión residual es enviar la salida de la activación de una capa hacia otra capa más profunda. Esto permite que el gradiente fluya a capas más profundas corrigiendo el error producido por el vanishing gradient, además, mejora la velocidad del funcionamiento de la red y reduce el número de capas necesarias para conseguir un funcionamiento correcto.

Esta es la representación de una conexión residual:

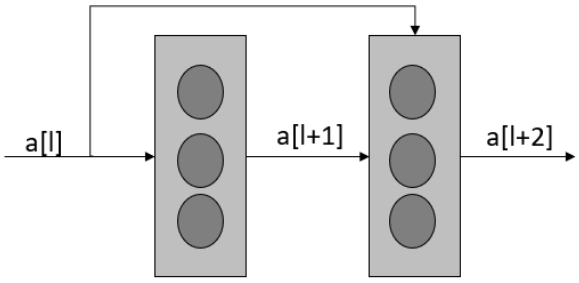


Figura 6-5. Conexión residual.

¿Por qué esta modificación arregla el vanishing gradient?

Esta modificación permite que características de capas anteriores tengan peso en las siguientes capas. En caso de que z[L+2] cayese a valores cercanos a 0 el término a[L] hace que a[L+2] tenga un valor distinto a cero. Si

esta activación fuese nula la alimentación hacia la siguiente capa haría que también fuese nula, es decir, si a[L+2] fuese 0 entonces a[L+3] sería prácticamente 0, este fenómeno sería como una bola de nieve que haría que el resto de las activaciones fuesen prácticamente nulas. Gracias a que nos alimentamos de capas anteriores este problema se eliminaría, ya que siempre existirá un término con peso influyente en activaciones futuras.

Existen dos bloques fundamentales en la arquitectura que se basan en el uso de las conexiones residuales que son:

- Bloque identidad (identity block)
- Bloque de convolución (convolution block)

6.2.2 Bloques residuales.

El bloque identidad es el bloque están en ResNet, corresponde con el caso donde la activación a[L] tiene la misma dimensión que la activación a[L+2]. Podemos ver una representación en la siguiente imagen

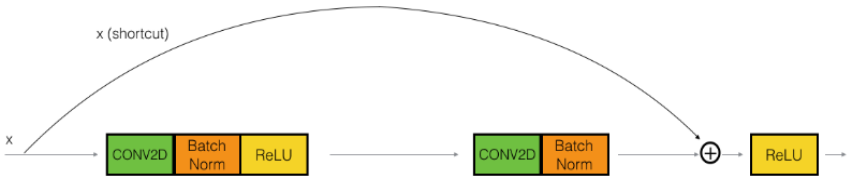


Figura 6-6. IdentityBlock.

El camino superior se conoce como “shortcut path”. El camino inferior como “main path”. Observamos que en cada capa existe una convolución y una capa ReLu como función de activación, además, se ha añadido la capa BatchNorm para acelerar la velocidad en el entrenamiento.

El segundo bloque es el convolution block. Este bloque es utilizado cuando la entrada y la salida no tienen el mismo tamaño. La principal diferencia es que en el shortcut path se ha añadido una capa de convolución y una capa BatchNorm. Representación en la siguiente imagen.

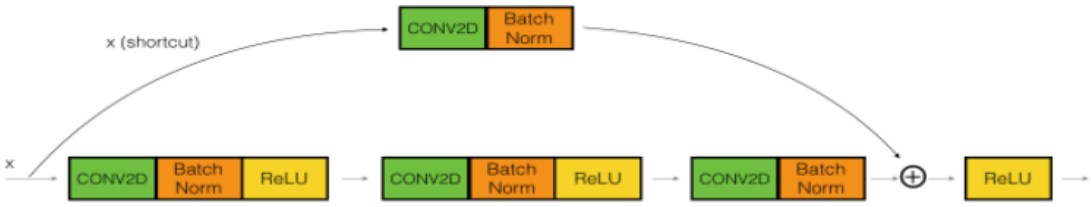


Figura 6-7. Convolution block.

La capa de convolución se utiliza para cambiar el tamaño de la entrada x para que tenga la misma dimensión

que la salida. Como ejemplo, si queremos reducir el tamaño en un factor de 2 podemos utilizar un filtro 1x1 con stride 2.

Estos son los dos bloques que utilizan las conexiones residuales, más tarde, cuando pasemos a describir la arquitectura de forma general nos referiremos a estos bloques por su nombre en inglés. Identity block y Convolution block, para el bloque identidad y el bloque convolución respectivamente.

6.2.3 Arquitectura

La arquitectura ResNet, uno de sus principales ventajes respecto a otras arquitecturas es que no presenta un esquema demasiado complejo, con todos los conceptos que hemos ido desarrollando a lo largo de la memoria se puede entender perfectamente.

La arquitectura presenta la siguiente estructura:

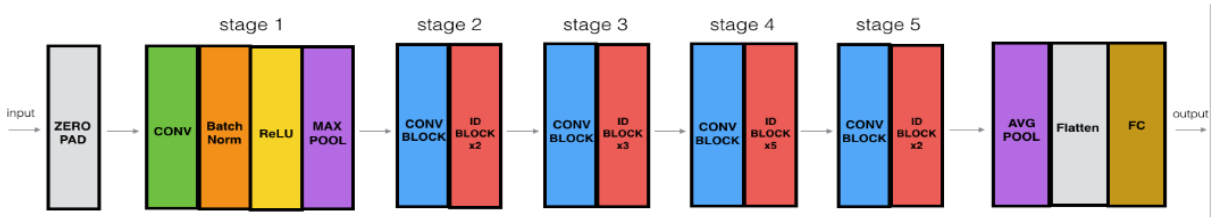


Figura 6-8. Arquitectura ResNet50

La arquitectura se divide en una serie de capas agrupadas en fases o stages. Pasaremos a definir la estructura de cada una de estas capas:

- Zero Pad: Primera capa, realiza un padding a la entrada 3*3. Se añaden 3 filas y 3 columnas adicionales a la matriz de entrada.
- Stage 1:
  - Capa de convolución: con 64 filtros de un tamaño (7,7) y strides de (2,2)
  - BathcNorm: aplicada al eje de los canales de la entrada
  - ReLu: capa para obtener la activación del mapa de características.
  - MaxPooling: usa una ventada de (3,3) con strides (2,2)
- Stage 2:
  - Convulational Block: cuenta con un número de filtros es de [64,64,256] con un tamaño cada uno de (3,3) y strides (1,1)
  - Identity block: Se utilizan dos bloques de identidad, el número de filtros es de [64,64,256], el tamaña de cada uno de estos filtros es (3,3)
- Stage 3:
  - Convulational Block: cuenta con un número de filtros es de [128,128,512] con un tamaño cada uno de (3,3) y strides (2,2)
  - Identity block: Se utilizan tres bloques de identidad, el número de filtros es de [128,128,512], el tamaña de cada uno de estos filtros es (3,3)
- Stage 4:
  - Convulational Block: cuenta con un número de filtros es de [256,256,1024] con un tamaño cada uno de (3,3) y strides (2,2)

  - Identity block: Se utilizan cinco bloques de identidad, el número de filtros es de [256,256,1024], el tamaña de cada uno de estos filtros es (3,3)
- Stage 5:
  - Convulational Block: cuenta con un número de filtros es de [512,512,2048] con un tamaño cada uno de (3,3) y strides (2,2)
  - Identity block: Se utilizan dos bloques de identidad, el número de filtros es de [512,512,2048] el tamaño de cada uno de estos filtros es de (3,3)
  - Average Pooling: utilizado extraer los parámetros más importantes obtenido en las capas de convolución, utiliza una ventana de (2,2) de tamaño
  - Flatten layer: Es una capa sin parámetros utilizada para convertir el mapa de características a una forma que pueda leer la siguiente que capa, que es una capa totalmente conectada.

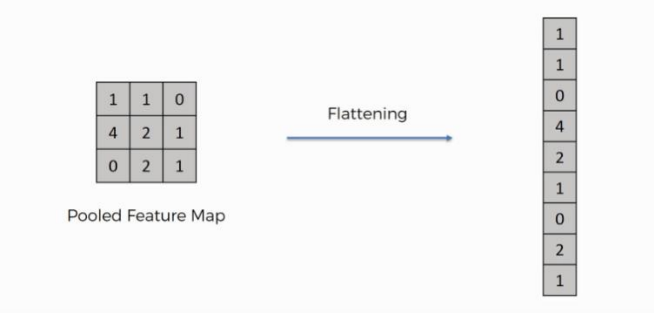


Figura 6-9. Flatten layer

- Fully Connected layer (Dense): capa final donde se realizará las predicciones del modelo, con una función de activación softmax para cada una de las clases de nuestro modelo.

Como se puede observar a medida que profundizamos en la red el número de filtros aumenta. Esto se debe a que en las capas iniciales serán las encargadas de capturas las características más sencillas y no necesitaran un número tan elevado de filtros para poder capturarlas, sin embargo, a medida que profundizamos en la red las capas se encargaran de capturar características más complejas de nuestra imagen y por ello necesitaran que el número de filtros aumenten.

6.2.4 Cambios en la arquitectura de la red

Hemos realizado una serie de cambios en la arquitectura de la red. Se han añadido algunas capas finales para poder aumentar la regularización e intentar corregir el overfitting. Realmente la arquitectura sin ninguna modificación funcionaba perfectamente, pero hemos querido experimentar como influye agregar algunas capas nuevas a la arquitectura de red. En este punto explicaremos las capas añadidas más adelante en la explicación de la aplicación se podrá ver explicada su implementación

Pasos realizados:

1º Eliminamos la capa final de predicción para poder introducir la nuestra, si se elimina la capa final de forma automática se elimina la capa AveragePooling. En este framework no es necesario implementar la capa Flatten, ya que el propio framework lo implementa por sí solo, esto lo hace en la capa AveragePooling que puede devolver una salida en forma flatten para que la capa totalmente conectada pueda utilizar los datos

2º Agregar la capa averagePooling anteriormente eliminada

3º Agregar una capa totalmente conectada con 256 unidades, función de activación relu y regularización L2

4º Agregar a la capa anterior Dropout, para intentar mejorar el overfitting. Dropout es una técnica de regulación que desactiva de manera aleatoria conexiones entre neuronas.

5º Crear una capa final totalmente conectada con función de activación Softmax y el número de neuronas igual al número de clases de nuestro proyecto que en este caso es 4.

7 TECNOLOGÍAS UTILIZADAS

7.1 Entorno de desarrollo

La aplicación ha sido desarrollo en dos entornos distintos. El entrenamiento de la red ha sido realizado en Google Colab y para desarrollar la aplicación de generación de diagnósticos fue realizado en Visual Studio Code.

Figura 7-1. Google colab logo.

Figura 7-2. Visual Studio Code logo.

7.1.1 ¿Por que Google Colab?

Como ya sabes el entrenamiento de redes neuronales conllevan un gran costo computacional, en mi caso no tenía de forma física un entorno que fuese capaz de realizar el entrenamiento de manera óptima de las redes en un tiempo razonable, por eso me decante por esta opción.

Google Colab nos ofrece un entorno de desarrollo en la nube y nos permiten utilizar de forma gratuita recursos. Si bien es cierto que se puede realizar un correcto entrenamiento de las redes más sencillas, pero para redes más complejas la forma gratuita que nos ofrecen puede ser escasa. Existe la posibilidad de mejorar el entorno pasando a la versión premium que si nos ofrecen todos los recursos necesarios para poder entrenar redes muy complejas. En mi caso he optado por la versión gratuita por los costes que ello conlleva.

Cabe destacar que todas las librerías necesarias, así como todos los frameworks vienen previamente instalados y no necesitamos hacernos cargos de este paso de instalación.

¿Cuáles son los recursos que se nos ofrecen?

Los usuarios gratuitos se le permite el acceso a tiempos de ejecución de GPU o TPU por gasta 12 horas.

Para la ejecución con GPU nos ofrecen:

-GPU: nvidia tesla k80

-CPU: Intel Xeon a 2.20 GHz.

-Ram: 13 gb de Ram

-VRAM: 12 gb DDR5

Para la ejecución con TPU nos ofrecen:

-TBU: TPU en la nube de 180 teraflops

-CPU: Intel Xeon a 2.30 GHz

-RAM: 13 gb RAM

Tenemos que mencionar que estos recursos no siempre están disponibles, aveces podemos recibir una tarjeta inferior a la que nos dicen que nos ofrecen o menos gb de RAM. Depende del estado de los servidores de Google. Nosotros hemos optado por la opción de utilizar una GPU ya que nos ofrecía mejor rendimiento que la TPU

Cabe destacar que Google colab permite integración con GitHub como almacenamiento en la nube vía Google Drive

7.1.2 ¿Por que visual code?

Visual Studio code es un editor de texto que nos permite trabajar de manera sencilla con las distintas librerías necesarias y Framework que necesita nuestro proyecto. Ofrece una terminal y una serie de ayuda que facilitan el proceso de instalación de los recursos necesarios.

Para nuestro caso hemos utilizado Visual Studio para la segunda parte de la aplicación donde no necesitábamos apenas recursos computacionales ya que la red ya ha sido entrenada en la primera parte de la aplicación. Por ello hemos utilizado Visual Studio, este también permite integración con GitHub

Existen otros editores o IDE interesantes que también podrían haber sido útiles en el desarrollo de esta aplicación como podría ser Pycharm.

7.2 Lenguajes De programación y Framework

7.2.1 PYTHON

El lenguaje de programación escogido para este proyecto ha sido Python, que es el lenguaje por excelencia en el desarrollo de aplicaciones de redes neuronales o tratamiento de datos. Permite el uso de bases de datos de forma sencilla y permite implementar librerías para poder trabajar con agentes externos como podría ser Google drive. Hemos escogido este lenguaje no solo por las ventajas que presenta si no porque tiene compatibilidad con las librerías y Framework necesario en nuestro proyecto. otro lenguaje que permite el entrenamiento de redes neuronales podría ser MatLab pero este no implementa las librerías y Framework necesarios para el desarrollo de esta aplicación.

7.2.2 TensorFlow

Respecto al framework utilizado en este proyecto es Tensorflow. Es uno de los Dramework más populares junto con Pytorch utilizado para el desarrollo de aplicación de redes neuronales.

¿Por qué Tensorflow?

Tensorflow ofrece números funcionalidades para el desarrollo de este tipo de aplicaciones de redes neuronales, que son:

- Ofrece la arquitectura de red neuronal más comunes y utilizadas para el diseño de estas aplicaciones, muchas de ellas ya vienen implementadas y solo con una línea de código podemos utilizarlas.
- Implementa las capas más comunes utilizadas en las redes neuronales como podría ser la capa de convolución de una CNN.
- Ofrece funciones necesarias para realizar el entrenamiento y test de nuestra red neuronal.
- Añade las funciones matemáticas más comunes necesarias para trabajar con las redes.

Figura 7-3. Logo TensorFLow.

7.2.3 Keras

Una de las principales dificultades que presenta tensorflow es la alta complejidad que presenta para ser utilizado. Por ello hemos utilizado la API Keras, que ha sido desarrollado en el lenguaje Python. Se trata de una biblioteca de código que se ejecuta sobre Tensorflow y ofrece un uso más sencillo e intuitivo del Framework. Ofrece la posibilidad de combinar capar, distintos optimizadores, esquemas de inicialización o funciones de activación.

Figura 7-4. Logo TensorFLow más Keras.

7.3 Base de datos y dataset

Una de las dificultades que puede suponer entrenar una red neuronal es encontrar unos datos de entrenamiento fiable, que nos permitan desarrollar un proyecto de este ámbito. Para nuestro proyecto hemos obtenido el dataset a través de Kaggle.

Kaggle es una página web donde se reúne donde se reúna una de las comunidades Data Science más grande del mundo. Ofrecen una gran diversidad de dataset de diversa índole de forma gratuita. Es una página bastante fiable ya que al tener una comunidad tan grande y activa todos estos dataset están en continua actualización.

El dataset seleccionado para este proyecto es:

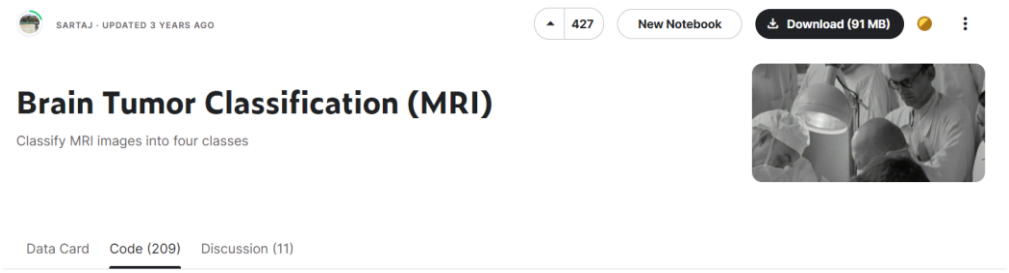


Figura 7-5. Dataset de la apliación.

Ofrece un dataset de 3.294 ejemplos de entrenamiento divididos de forma equilibrada entre las distintas 4 clases que presenta el proyecto. Los datos están divido entre carpetas de entrenamiento y testing. Dentro de estas carpetas estará dividido en 4 subcarpetas, cada una de ellas guardará los datos de una clase del entrenamiento. Aunque sea de gran ayuda ya tener divido los datos entrenamiento y testing nosotros no hemos seguido esta distribución. El único problema que presenta es que no ofrecen un fichero con los datos etiquetados y seremos nosotros los que tenemos que generar las etiquetas.

Respecto al lugar donde almacenamos este dataset hemos elegido guardar los datos en Google Drive ya que al utilizar Google Colab nos permitía conectarnos con este de forma sencilla.

8 APLICACIÓN

Para este proyecto de fin de grado se ha decidido realizar una aplicación médica basada en Deeplearning para la detección de cáncer cerebral usando imágenes de resonancia magnética. Nuestra aplicación se dividirá en dos partes que son:

1. En la primera parte se creará una aplicación para poder entrenar una red neuronal utilizando las diversas técnicas que se han ido explicando a lo largo de la memoria de este proyecto. Nos basaremos en la arquitectura ResNetV50 previamente explicada con algunas capas más agregadas.
2. La segunda parte corresponde con la creación de una aplicación para generar diagnósticos, utilizando la red previamente entrenada por nosotros, donde se generará un informe para el paciente examinado. Esta segunda parte se ha diseñado para dar un valor añadido y demostrar la utilidad de nuestra red.

8.1 Aplicación de entrenamiento de la red neuronal

El primer paso antes de comenzar a desarrollar la aplicación fue encontrar un conjunto de datos de entrenamiento idóneos para nuestra aplicación. En el apartado de tecnologías se explica cuál es el dataset utilizado y de donde se obtuvo. Esta aplicación se desarrolló en Google Colab y los datos de entrenamiento se guardaron en Google drive.

IMPORTS NECESARIOS

Estos son los imports necesarios para el funcionamiento de nuestra aplicación.

```
from google.colab import drive
drive.mount('/content/drive')
import numpy as np
import cv2
import tensorflow as tf
import os
from tqdm import tqdm
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from keras.applications.inception_v3 import preprocess_input
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D,
Dropout
from tensorflow.keras import regularizers
from keras import backend as K
```

Figura 8-1. Imports aplicación entrenamiento.

LECTURA DE DATOS

El primer paso en el desarrollo de nuestra aplicación fue implementar en el código la lectura de los datos de entrenamiento alojados en nuestro drive. Un problema que encontramos en este dataset es que no contaba con un fichero donde se estuviesen los datos etiquetados y tuvimos que generarlo nosotros. Para ello nos ayudamos de cómo estaba distribuido los datos. Existen 4 clases en nuestro proyecto que son: no-tumor, glioma tumor, meningioma tumor, pituitary tumor. Cada una de las clases estaba guardada en una carpeta con el nombre de su clase correspondiente hecho que nos ayudó a generar las etiquetas.

Adjuntamos el código correspondiente con la lectura de los datos.

```
"""
Definición de cada una de clases, existen 4 clases de distintas, tenemos
3 tipos de cáncer distintos
de cáncer y una clase que representa la ausencia de cáncer
"""

labels = ['no_tumor','glioma_tumor','meningioma_tumor','pituitary_tumor']

#Variables donde se guardaran las imagenes y sus labels correspondientes"

X = []
Y = []

#Definición del tamaño de la magen, en este caso el tamaño optimo para
nuestra arquitectura ResNet50 es de 224x224

image_size = 224

"""
Bucle for que iremos iterando para obtener cada una de las imágenes de
nuestro dataset, en este caso el dataset se encuentra guardado dentro de
una carpeta de mi drive en caso de querer
modificar la ubicación del dataset deberemos simplemente modificar el
path de la ubicación de la varible folderPath.
"""

for i in labels:

  """
  Ubicación de nuestro dataset se encuentra en mi drive y su ubicación
absoluta es /content/drive/Mydrive/tfg/dataset/Training/i donde 'i'
representa el nombre
  de la carpeta donde se guarda cada uno de los ejemplos de una misma
clase, el nombre de las carpetas tienen el mismo nombre que el nombre de
```

```
su clase
  """


  folderPath =
os.path.join('/content/drive/MyDrive/','tfg','dataset','Training',i)

  #recorremos cada uno de los elementos de la carpeta indicada en la
varibale folderPath

  for j in tqdm(os.listdir(folderPath)):
    "guardamos en la varibale una imagen en cada iteración y realizamos
el cambio de tamaño con un valor y añadimos dicho ejemplo a la varibale
X, así como su label correspondiente"
    img = cv2.imread(os.path.join(folderPath,j))
    img = cv2.resize(img,(image_size, image_size))
    X.append(img)
    Y.append(i)

#Mismo procedimiento que el bucle anterior, pero para el caso de las
imagenes de test

for i in labels:
    folderPath =
os.path.join('/content/drive/MyDrive/','tfg','dataset','Testing',i)
    for j in tqdm(os.listdir(folderPath)):
        img = cv2.imread(os.path.join(folderPath,j))
        img = cv2.resize(img,(image_size,image_size))
        X.append(img)
        Y.append(i)

#Conversión variables a array.

X = np.array(X)
Y = np.array(Y)
```

Figura 8-2. Lectura de datos aplicación

Aumento de datos

Una vez que comenzamos a diseñar la aplicación y realizamos el entrenamiento por primera se observó que el

número de datos de entrenamiento no era lo suficientemente grande, al tener un tamaño tan pequeño provocaba que nuestra red no fuese capaz de aprender características nuevas, provocando un overfitting bastante grande. Para solucionar esto desarrollamos una función para aumentar los datos. En dicha función se coge una imagen y se realiza una serie de transformaciones y se genera n nuevas imágenes a partir de estas. Las transformaciones son las siguientes: Rotación a favor o en contra de las agujas del reloj, Translación aleatoria en los ejes x e y, cambio de escala, reflejo horizontal, ajustes de brillo y contraste. El valor que toma en cada transformación es aleatorio dentro de unos valores máximos y mínimos que puede alcanzar para que dichas transformaciones no metan demasiado ruido o desplacen la imagen demasiado.

Adjuntamos el código correspondiente:

```
def aumentar_datos(X, Y, n):

    """
    Funcíon para aumentar el número de datos de entrenamiento realizando
transformaciones geometricos y ajusted de brillo y contraste.
    Devuelve un nuevo conjunto de datos implementando estas
transformaciones.

    """

    #Comprobación parametros

    assert isinstance(X, np.ndarray), "X debe ser un array de numpy"
    assert isinstance(Y, np.ndarray), "Y debe ser un array de numpy"
    assert len(X.shape) == 4, "X debe ser un tensor 4D con dimensiones
(batch, height, width, channels)"
    assert len(Y.shape) == 1, "Y debe ser un tensor 1D con dimensiones
(batch,)"
    assert len(X) == len(Y), "X e Y deben tener la misma longitud"
    assert isinstance(n, int) and n > 0, "n debe ser un entero positivo"

    #variables donde guardaremos nuestro nuevo conjunto de entrenamiento
aumentado

    X_aug = []
    Y_aug = []


    #recorremos todas las imagenes de entrada

    for i in range(len(X)):

        #Variables donde se guardaran cada imagen y sus etiquetas
correspondiente en cada iteración

        imagen = X[i]
        etiqueta = Y[i]

        #Bucle donde se apliran n transformaciones a cada imagen
```

```
        for j in range(n):

            #realizamos una rotación aleatoria entre 15 grados a favor de
las agujas del reloj o encontra a ellas

            angulo = np.random.randint(-15, 15)
            M = cv2.getRotationMatrix2D((imagen.shape[1]//2,
imagen.shape[0]//2), angulo, 1)
            imagen_rotada = cv2.warpAffine(imagen, M, (imagen.shape[1],
imagen.shape[0]))

            #Transalción aleatorias de 10 pixeles en los ejes x e y

            tx = np.random.randint(-10, 10)
            ty = np.random.randint(-10, 10)
            M = np.float32([[1, 0, tx], [0, 1, ty]])
            imagen_traslada = cv2.warpAffine(imagen_rotada, M,
(imagen.shape[1], imagen.shape[0]))

            #Cambio de escala entre 0.9 y 1.1

            escala = np.random.uniform(0.9, 1.1)
            M = cv2.getRotationMatrix2D((imagen.shape[1]//2,
imagen.shape[0]//2), 0, escala)
            imagen_escala = cv2.warpAffine(imagen_traslada, M,
(imagen.shape[1], imagen.shape[0]))

            #reflejo horizontal con probabilidad 0.5

            if np.random.random() < 0.5:
                imagen_reflejo = cv2.flip(imagen_escala, 1)
            else:
                imagen_reflejo = imagen_escala


            #Ajuste de brillo y contraste. Alpha representa el contraste
y beta el brillo que se va a modificar

            alpha = np.random.uniform(0.5, 2.0)
            beta = np.random.randint(-30, 30)
            imagen_bc = cv2.convertScaleAbs(imagen_reflejo, alpha=alpha,
beta=beta)

            #añadimos las modificaciones de imagen a nuestra variable
donde guardará todo las imagenes aumentadas

            X_aug.append(imagen_bc)
            Y_aug.append(etiqueta)
```

```
    return np.array(X_aug), np.array(Y_aug)
```

Figura 8-3. Función de aumento de datos.

Distribución de datos de entrenamiento y test

Tras generar nuestro nuevo conjunto de imágenes a partir del original tendremos que dividir los datos en ejemplos de entrenamiento y test con sus etiquetas correspondiente. Para ello usaremos el método *train_test_split*. La función divide todos los datos en ejemplos de entrenamiento y test de forma aleatoria. El parámetro *test_size* indica que porcentaje de todos los datos queremos que sean utilizado para el test, en nuestro caso hemos utilizado un valor del 15%, el siguiente parámetro es *random_state* es la semilla de generación aleatoria, en nuestro caso lo hemos dejado como un valor fijo para que reparta de forma aleatoria los datos, pero siempre de la misma manera para poder comprobar el funcionamiento de nuestra red. En la práctica este parámetro no se mantendrá fijo y en cada ejecución del entrenamiento tendremos una distribución aleatoria de los datos.

Adjunto el código correspondiente

```
"""
Reparto de las imagenes de entranamientos, test y sus labels
correspondiente. De todo nuestras imagenes el 85% serán utilizadas para
el entrenamiento y 15% para test
"""
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.15,
random_state=42)

labels = ['glioma_tumor','no_tumor','meningioma_tumor','pituitary_tumor']

#Conversión de nombre del label a un número entero para poder trabajar
con el

class_to_int =
{'no_tumor':0,'glioma_tumor':1,'meningioma_tumor':2,'pituitary_tumor':3}

#conversión del vector Y a un one-hot vector
Y_train_int = [class_to_int[labels] for labels in Y_train]
Y_train_one_hot = to_categorical(Y_train_int, num_classes=4)

Y_test_int = [class_to_int[labels] for labels in Y_test]
Y_test_one_hot = to_categorical(Y_test_int, num_classes=4)

#impresión número de ejemplo y shape de los vectores

print ("number of training examples = " + str(X_train.shape[0]))
```

```
print ("number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train_one_hot.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test_one_hot.shape))
```

Figura 8-4. Distribución de datos.

.

Creación arquitectura

Tras obtener los ejemplos de entrenamiento y test con sus etiquetas correspondientes el siguiente paso sería cargar la arquitectura y añadir las capas.

Pasos para crear la arquitectura:

1. Cargaremos la arquitectura ResNet50, además, aplicaremos transferlearning para cargar los pesos de esta arquitectura. Estos pesos han sido previamente entrenados en el conjunto de datos ImagenNet. Este conjunto de datos está formado por una gran cantidad de imágenes de diversos tipos que nuestra red utilizará para aprender las características más básicas y comunes que pueden compartir cualquier aplicación de reconocimiento de imágenes con nuestra aplicación médica. Por otro lado, hemos desactivado las capas superiores encargadas de realizar la predicción para así poder añadir nuestras propias capas al modelo.

```
base_model = ResNet50(weights='imagenet', include_top=False)
```

Figura 8-5. Creación modelo.

2. EL segundo paso será congelar todas las capas del modelo. Más tarde descongelaremos aquellas que queremos aplicar fine-tunning.

```
# Congelar todas las capas del modelo original
for layer in base_model.layers:
    layer.trainable = False
```

Figura 8-6. Congelación de capas.

3. El siguiente paso será añadir las capas nuevas a nuestro modelo, recordando las capas eran: AveraPooling->FullyConnectedLayer(256 unidades + L2 regularización + dropout y función de activación ReLu)->fullyConnected(función de activación Softmax con 4 salidas para cada una de nuestras clases)

```
num_classes = 4

# Agregar una capa de promediado global para reducir la
dimensionalidad de las características
x = base_model.output
x = GlobalAveragePooling2D()(x)
```

```
# Agregar una capa totalmente conectada con 256 unidades y una función
de activación ReLU, con regularización L2
x = Dense(256, activation='relu',
kernel_regularizer=regularizers.l2(0.01))(x)

# Agregar una capa de Dropout para regularizar y reducir el
overfitting
x = Dropout(0.5)(x)

# Agregar la capa de salida con el número de clases de su problema
predictions = Dense(num_classes, activation='softmax')(x)
```

Figura 8-7. Nuevas capas añadidas al modelo.

4. Una vez creada las nuevas capas pasaremos a definir el nuevo modelo, indicaremos que será la unión de la arquitectura ResNet más las nuevas capas creadas.

```
# Definir el modelo final
model = tf.keras.models.Model(inputs=base_model.input,
outputs=predictions)
```

Figura 8-8. Definición del modelo final.

5. Con el modelo definido paremos a compilarlo, para nuestro caso hemos utilizado como función de perdidas categorical_crossentropy. Optimizador: Adam. Para evaluar el modelo metrics: accuracy.

```
# Compilar el modelo con función de perdidas categorlical_crossentropy
y optimizador adam
model.compile(loss='categorical_crossentropy',
optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
metrics=['accuracy',tf.keras.metrics.FalsePositives(),
tf.keras.metrics.FalseNegatives()])
```

Figura 8-9. Compilación del modelo final.

6. Tras definir nuestro modelo, pasaremos a aplicar la técnica conocida como fine-tunning donde descongelaremos algunas capas del modelo original para entrenarlar a partir de los pesos previamente precargados. En nuestro descongelaremos en bloque final del stage 5, donde se aprenderán las características más importantes de nuestra aplicación.

```
# Descongelar capas finales para poder aplicar finetunning
for layer in base_model.layers:
    if 'conv5_block3' in layer.name:
        layer.trainable = True
```

Figura 8-10. Descongelar capas finales.

7. Como paso final deberemos volver a compilar de nuevo el modelo con las capas descongeladas. Esta nueva compilación se debe que tenemos que indicarle al modelo el learning rate que queremos que aplique a estas capas descongeladas. Para el caso de las capas descongeladas como ya fueron previamente entrenadas necesitamos un learning rate pequeño para no provocar que se actualicen demasiado los parámetros. Nuestra intención será realizar un pequeño ajuste de los parámetros de esta capa, en caso de utilizar un learning rate demasiado alto provocarían demasiado ajustes en los parámetros degradan el rendimiento final de la red.

```
#Compilación del modelo con las nuevas capas añadidas,
model.compile(33pt='categorical_crossentropy',
33ptimizar=tf.keras.optimizers.Adam(learning_rate=0.001),
metrics=['accuracy',tf.keras.metrics.FalsePositives(),
tf.keras.metrics.FalseNegatives()])
```

Figura 8-11. Compilación del modelo con las capas descongeladas.

Entrenamiento

El siguiente paso a la hora de crear nuestra aplicación será entrenarla. La nueva arquitectura y los pesos generados lo guardaremos en nuestro drive para poder obtenerlo en cualquier momento sin necesidad de entrenar la red de nuevo. El resultado de la ejecución será guardado en una variable que utilizaremos más tarde para imprimir los resultados del entrenamiento. En este caso también hemos introducido los ejemplos de test para que en cada epoch nos de información real sobre el resultado obtenido

```
#path donde se guardará los pesos entrenados y arquitectura utilizada
weights_path = "/content/drive/MyDrive/tfg/pesos/weights.h5"

#numero de epoch a reliazar
num_epochs = 100

#tamaño del bach
bach_tam=32

#Entrenamiento del modelo y validación del mismo
history = model.fit(X_train, Y_train_one_hot, validation_data= (X_test,
Y_test_one_hot) ,epochs = num_epochs, batch_size = bach_tam)

#Guardamos los pesos entrenados y la arquitectura en el path previamente
definido
model.save(weights_path)
```

Figura 8-12. Entrenamiento del modelo.

Todos los resultados obtenidos serán explicados en el apartado de "Resultados obtenidos" aquí nos centraremos en explicar el funcionamiento de nuestra aplicación.

Representación gráfica de los resultados.

Representaremos de forma gráfica los resultados obtenidos. Estos resultados se guardan en el diccionario history.

donde:

- Loss: representa valor de la función de pérdidas durante el entrenamiento
- Val_loss: valor de la función de pérdidas durante el test
- Accuracy: precisión obtenida durante el entrenamiento
- Val_accuracy: precisión obtenida durante el test.
- False_positivos = Falsos positivos en el entrenamiento.
- Val_ False_positivos = Falsos positivos en el test.
- False_negativos = Falsos negativos en el entrenamiento.
- Val_ False_negativos = Falsos positivos en el test.

Código en cuestión:

```
#Variables donde se cargan los valores de accuracy y val_acurracy
obtenidas en cada epoch de entrenamiento
acc = [0.] + history.history['accuracy']
val_acc = [0.] + history.history['val_accuracy']
#Variables donde se cargan los valores de loss y val_loss obtenidas en
cada epoch de entrenamiento
loss = history.history['loss']
val_loss = history.history['val_loss']
#variables donse cargan los valores de false_positives y
val_false_positives
false_positive = history.history['false_positives_1']
val_false_positive = history.history['val_false_positives_1']
#variables donse cargan los valores de false_negatives y
val_false_negatives
false_negative = history.history['false_negatives_1']
val_false_negative = history.history['val_false_negatives_1']

# Creamos cuatro gráficas separadas
fig, axs = plt.subplots(4, 1, figsize=(8, 12))

# Representación de accuracy y val_acuraccy
axs[0].plot(acc, label='Training Accuracy')
axs[0].plot(val_acc, label='Validation Accuracy')
axs[0].legend(loc='lower right')
axs[0].set_ylabel('Accuracy')
axs[0].set_ylim([min(plt.ylim()), 1.2])
axs[0].set_title('Training and Validation Accuracy')
```

```
# Representación de Loss y val_loss
axs[1].plot(loss, label='Training Loss')
axs[1].plot(val_loss, label='Validation Loss')
axs[1].legend(loc='upper right')
axs[1].set_ylabel('Cross Entropy')
axs[1].set_ylim([0, 1.0])
axs[1].set_title('Training and Validation Loss')

# Representación de false_positives y val_false_positives
axs[2].plot(false_positive, label='Training false positive')
axs[2].plot(val_false_positive, label='Validation false positive')
axs[2].legend(loc='upper right')
axs[2].set_ylabel('nº false positive')
axs[2].set_ylim([0, 2000.0])
axs[2].set_title('Training and Validation false positive')


# Representación de false_negatives y val_false_negatives
axs[3].plot(false_negative, label='Training false negative')
axs[3].plot(val_false_negative, label='Validation false negative')
axs[3].legend(loc='upper right')
axs[3].set_ylabel('nº false negative')
axs[3].set_ylim([0, 2000.0])
axs[3].set_title('Training and Validation false negative')
axs[3].set_xlabel('epoch')

plt.show()
```

Figura 8-13. Representación gráfica del modelo.

Test

Comprobación con los datos de test la red entrenada. Como hemos dicho anteriormente los resultados serán discutidos más tarde.

```
#Comprobación del funcionamiento del modelo usando las imágenes de test
preds = model.evaluate(X_test, Y_test_one_hot)
#Perdidas y accuracy alcanzado en el test
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

Figura 8-14. Test del modelo.

8.2 Aplicación de diagnostico

La segunda parte de la aplicación es la encargada de generar los diagnósticos a partir de la red previamente

entrenada. Esta parte de la aplicación ha sido diseñada en local y el único requisito para su funcionamiento es tener los pesos previamente entrenados guardados en nuestro equipo local. Pasaremos a explicar la aplicación

IMPORTS NECESARIOS

```
import tkinter as tk
from tkinter import filedialog
import tensorflow as tf
import cv2
import numpy as np
import os
from datetime import datetime
import random
import string
from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import A4
```

Figura 8-15. Import aplicación diagnóstico.

1º definimos el tamaño de la imagen, en este caso es 224 porque es el tamaño con el que trabaja nuestra red neuronal

```
image_size = 224
```

Figura 8-16. Tamaño de la imagen necesaria.

2º Creamos una ventana para poder seleccionar la imagen a la cual queremos realizar el diagnostico

```
    # Crear ventana para poder seleccionar la imagenen que realizaremos la predicción
    root = tk.Tk()
    root.withdraw()
```

Figura 8-17. Creación ventana de selección de imagen.

3º Obtenemos la ruta del archivo que hemos seleccionado

```
# Obtenemos la ruta del archivo
```

```
    ruta_archivo = filedialog.askopenfilename()
```

Figura 8-18. Obtención ruta archivo.

4º Leemos la imagen apartir de la ruta, cambiamos su tamaño al definido anteriormente y la convertimos en un array para poder trabajar con él ya que nuestra red necesita que el input sea un array.

```
    #lectura de la imagen apartir de la ruta anterior obtenida
    imagen = cv2.imread(os.path.join(ruta_archivo))
    imagen = cv2.resize(imagen,(image_size, image_size))
    #convertir imagen en array
    imagen_array = np.array(imagen)
    imagen_array = np.expand_dims(imagen_array, axis=0)
```

Figura 8-19. Lectura, escalado y conversión de la imagen de un array.

4º Indicamos los pesos entrenados anteriormente y lo cargamos en un nuevo modelo. Este archivo además de tener guardado los pesos, también guarda la estructura de la arquitectura.

```
    #obtener pesos de la red
    pre_trained_weights = 'weights.h5'
    #Carga de los pesos en el nuevo modelo así como su arquitectura
                                    pre_trained_model                =
tf.keras.models.load_model(pre_trained_weights)
```

Figura 8-20. Carga del modelo.

5º Realizamos la predicción de la imagen seleccionada anteriormente.

```
    #predicción de la imagen seleccionada
    prediction = pre_trained_model.predict(imagen_array)
```

Figura 8-21. Predicción.

6º Creamos un diccionario que asocia cada una de las distintas clases.

```
clase_resultado = {0:'no tiene tumor',1:'tiene tumor glioma',2:'tiene tumor meningioma',3:'tiene tumor pituitary'}
```

Figura 8-22. Diccionario con clases del modelo.

7º Obtenemos la hora actual a la cual fue generado el diagnóstico, creamos una cadena que contiene números y letras que será el identificador del diagnóstico. Lo imprimimos por terminal, más tarde estos valores serán imprimidos a un pdf que será el diagnostico final que podrá enviado al paciente en cuestión.

```
    #obtenemos la fecha cuando se ha genrado el diagnostico
    fecha_actual = datetime.now().strftime(' A la fecha %d-%m-%Y, A la Hora: %H:%M:%S')
    length_of_string = 5

    # Generar una cadena de caracteres aleatoria
    numero_diagnostico = ''.join(random.choice(string.ascii_letters + string.digits) for _ in range(length_of_string))

    #imprimirmos los resultados obtenidos de la predicción
    print("Número del diagnostico: "+numero_diagnostico)
    print("Diagnostico Generado" + fecha_actual)
```

Figura 8-23. Impresión valores en la terminal.

8º Obtenemos cual ha sido la clase resultado de la predicción a partir del índice que nos devuelve el modelo y lo asociamos al anterior diccionario creado. El modelo devuelve para clase con su índice correspondiente la probabilidad que esta sea la clase final.

```
print("El resultado del diagnostico es: ", clase_resultado.get(np.argmax(prediction)), "Con una probabilidad de acierto del:", "%.7f" % np.amax(prediction),"%")
```

Figura 8-24. Impresión de la predicción final.

Una vez obtenido estos resultados que han sido imprimidos por la terminal, nuestro siguiente objetivo será trasladar dicha información a un pdf que será el enviado a nuestros pacientes. A parte de la información anterior se agregará nueva para intentar recrear un diagnóstico real. Estos valores pueden ser modificados a libertad de usuario de la aplicación para que se ajusten de la mejor manera a la entidad que genera los diagnósticos.

9º Creamos un objeto tipos canvas que nos permite trazar la información y la imagen del diagnóstico a un archivo tipo pdf.

```
# Crear un objeto Canvas con el nombre del archivo PDF que quieres generar
    nombre_archivo = "Diagnostico_"+numero_diagnostico+".pdf"
    documento = canvas.Canvas(nombre_archivo, pagesize=A4)
    _, h = A4
```

Figura 8-25. Creación objeto canvas.

10º Agregamos toda la información al diagnóstico, como hemos dicho anteriormente esta puede ser modificada a gusto del consumidor.

```
    # Agregar una línea para imprimir en el pdf los resultados obtenidos
    text = documento.beginText(50, h - 50)
    text.setFont("Times-Roman", 12)
    text.textLine("Número del diagnostico: "+numero_diagnostico)
    text.textLine("Centro de salud: Hospital de Valme")
    text.textLine("Diagnostico Generado" + fecha_actual)
        text.textLine("El resultado del diagnostico es: " +clase_resultado.get(np.argmax(prediction)))
    text.textLine("probabilidad de acierto es: " + prob_acierto + "%")
```

```
    text.textLine("imagen del diagnostico:")

    #guardamos en el documento las líneas escritas

    documento.drawText(text)
```

Figura 8-26.Impresion en el PDF de la descripción y resultado de la predicción.

11º Añadir la imagen a la que le hemos hecho el diagnóstico y guardamos el archivo

```
    #obtenemos la dirección de la imagen de forma absoluta

    ruta_imagen_path = os.fspath(os.path.abspath(ruta_archivo))

    #imprimimos la imagen en el documento PDF

    documento.drawImage(ruta_imagen_path, 0, 100, 500, 500)

    #gaurdamos el documento PDF

    documento.save()
```

Figura 8-27. Impresión de la imagen de la predicción.

El resultado de la ejecución es el siguiente:

Primero se abre una interfaz para seleccionar la imagen

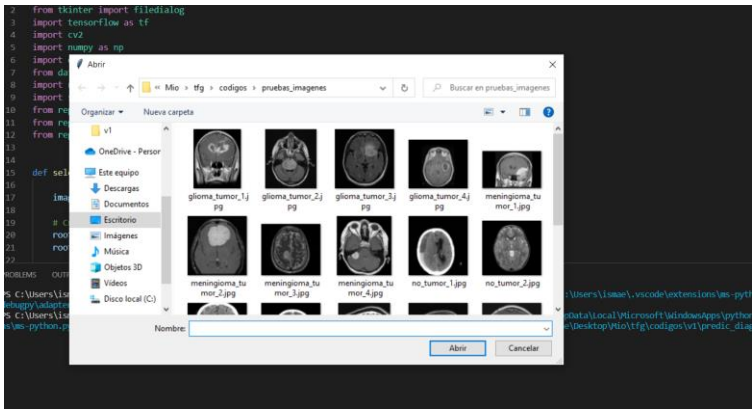


Figura 8-28. Interfaz selección imagen.

Tras esto realiza la predicción y generaría la información que ha sido explicada anteriormente.

Por la terminal este es el resultado:

```
1/1 [==============================] - 1s 904ms/step
Número del diagnostico: uKBb2
Diagnostico Generado A la fecha 29-04-2023, A la Hora: 20:46:17
El resultado del diagnostico es:  tiene tumor meningioma Con una probabilidad de acierto del: 0.9840746 %
PS C:\Users\ismae\Desktop\Mio\tfg\codigos\v1>
```

Figura 8-29. Resultado de la predicción terminal.

Y este es el pdf generado:

Número del diagnostico: uKBb2
Centro de salud: Hospital de Valme
Diagnostico Generado A la fecha 29-04-2023, A la Hora: 20:46:17
El resultado del diagnostico es: tiene tumor meningioma
probabilidad de acierto es: 0.9840746%
imagen del diagnostico:

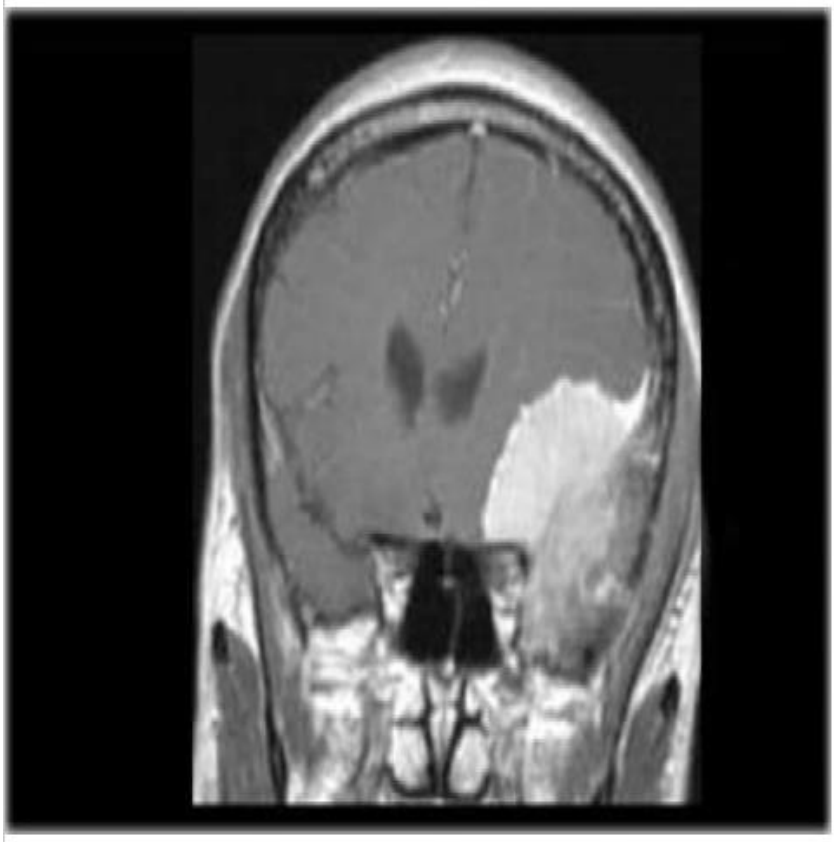


Figura 8-30. Diagnóstico paciente.

9 DISCUSIÓN VALORES

Para este proyecto de fin de grado se ha decidido realizar una aplicación médica basada en deeplearning para la detección de cáncer cerebral usando imágenes de resonancia magnética. Nuestra aplicación se dividirá en dos partes que son:

Finalmente habiendo explicado los conceptos necesarios para entender qué es el Deep learning y como este funciona centrándonos en las redes convolucionales nos encontramos en la posición de poder comprender los resultados obtenidos de la aplicación. Explicaremos y demostraremos los resultados modificando los distintos hiperparámetros de la red que son: learning rate, tamaño del bacth, número de epoch, porcentaje de distribución de datos, valor aumento de datos. Por otro lado, también comprobaremos cual es el algoritmo de optimización y función de pérdidas que más se ajustan a nuestra aplicación

Antes de comenzar una pregunta que puede surgir es ¿Realmente es necesario entrenar la red si está ya ha sido entrenada para detectar imágenes de otros ámbitos? La respuesta es sí, ya que, aunque nuestra red haya sido previamente entrenada y cargado los pesos utilizando la técnica de transferlearning en una aplicación de detección de imágenes, esta no está entrenada ante este tipo de datos, aunque sean imágenes. Realizaremos una ejecución y observaremos el resultado obtenido.

```
Loss = 2.240041732788086
Test Accuracy = 0.17222599685192108
```

Figura 9-1. Resultado de la red sin entrenamiento.

Como se puede ver el resultado obtenido es el esperado, 17% de acierto con una alta tasa de perdidas. Esto responde a la pregunta anterior. Ahora sí comenzamos con la discusión de los resultados.

9.1 Algoritmo de optimización

Una parte importante a la hora de diseñar una aplicación es ver que algoritmo de optimización utilizar, en nuestro caso vamos a analizar los siguientes 3: Adam, RMSprop y sgd con momento. Estos algoritmos son 3 de los más utilizados para las redes convolucionales.

Para poder comprobar que algoritmo de optimización es el que más se ajusta a nuestra aplicación, deberemos de entrenar de la misma forma nuestra red manteniendo el valor de los distintos hyperparametros y función de perdidas, los valores que se utilizarán en cada ejecución serán:

- Función de perdidas: categorical crossentropy
- Número de epoch: 20
- Tamaño del batch: 32
- Training y test distribución: 85% de los datos para entrenamiento y 15% test
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

La nomenclatura utilizada será la siguiente

- Acurracy: precisión alcanzada en el entrenamiento
- Loss = valor de la función de perdidas entrenamiento
- Val_acuracy = precisión en el test
- Val_loss= valor función de perdidas test
- False_positive = número de falsos positivos en el entrenamiento
- False_negativo= número de falsos negativos en el entrenamiento
- Val_False_positive = número de falsos positivos en el test
- Val_False_negativo= número de falsos negativos en el test

ADAM

Resultado obtenido:

N.º epoc	1	2	3	4	5	10	15	20
Loss	1.7854	0.4654	0.3082	0.2106	0.1653	0.1084	0.0939	0.0839
Acuracy	0.8103	0.9030	0.9279	0.9506	0.9661	0.9813	0.954	0.9871
False_positves	1192	652	510	323	240	142	100	92
False_negatives	1848	957	666	467	310	167	110	120
Val_loss	0.6649	0.4156	0.2859	0.2523	0.6256	0.2135	0.1662	0.1559
Val_acuracy	0.8747	0.9013	0.9312	0.8645	0.9074	0.9523	0.9543	0.9880
Val_false_positives	146	127	92	82	188	116	60	46
Val_false_negatives	205	166	107	100	206	139	70	50

Tabla 9–1. Resultados obtenidos del algoritmo ADAM

Valor final obtenido en el test:

- Precisión: 0.988
- Valor función pérdidas: 0.1559
- N.º falsos positivos: 46
- N.º falsos negativos: 50

Representación gráfica de la evolución del entrenamiento:

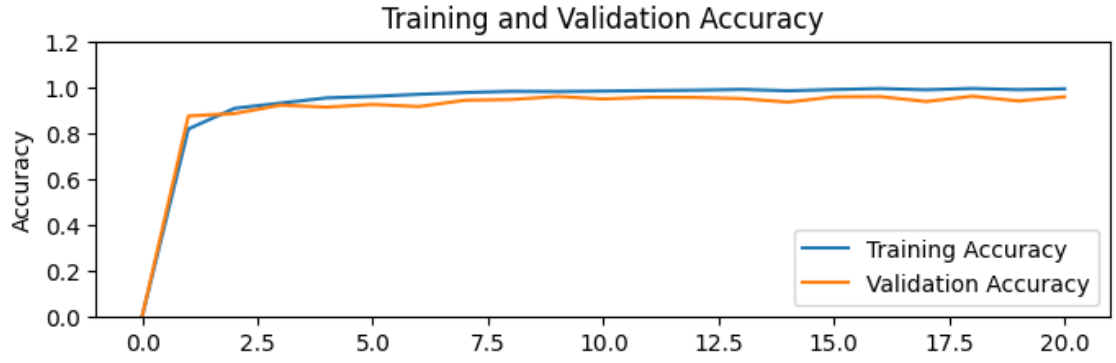


Figura 9-2. Precisión obtenida en el entrenamiento y test usando ADAM.

Figura 9-3.. Perdidas obtenidas en el entrenamiento y test usando ADAM.

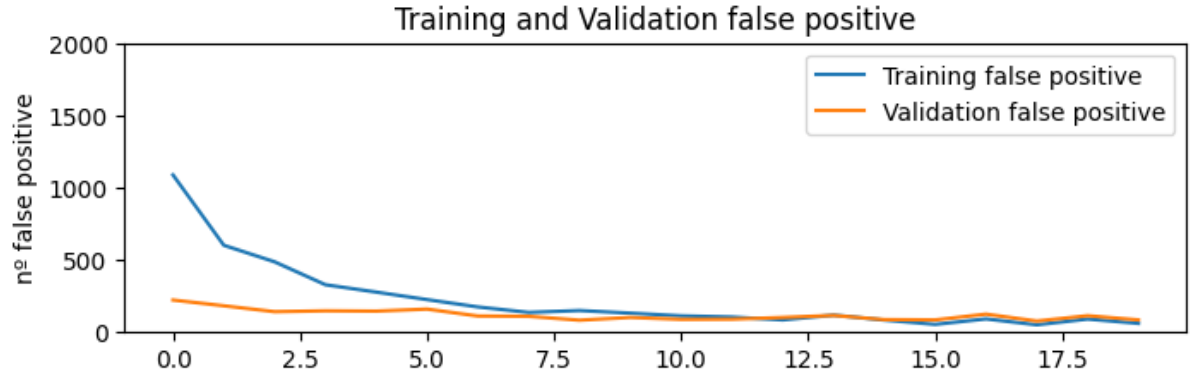


Figura 9-4. Número de falso positivo obtenida en el entrenamiento y test usando ADAM.

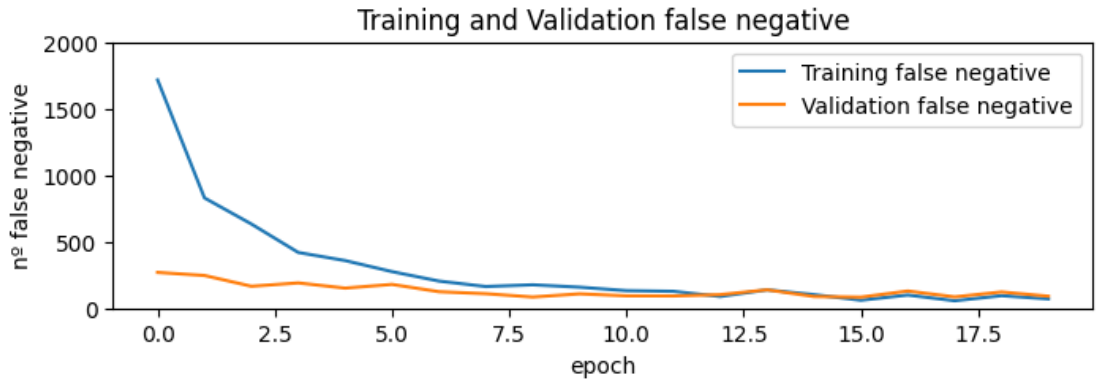


Figura 9-5. Número de falso negativo obtenida en el entrenamiento y test usando ADAM

.

RMSTOP

Resultado obtenido:

N.º epoc	1	2	3	4	5	10	15	20
Loss	1. 4837	0.3989	0.2781	0.02023	1729	0.0940	0.0449	0.0548
Acuracy	0.7900	0.8896	0.9250	0.9488	0.9581	0.9814	0.9916	0.9845
False_positves	1370	789	538	386	316	145	65	60
False_negatives	2026	1033	678	456	380	165	73	69
Val_loss	1.1211	0.3910	0.2756	1.6770	0.2179	0.2365	0.2156	0.216
Val_acuracy	0.7706	0.8897	0.9333	0.6909	0.9558	0.9251	0.9354	0.9436
Val_false_positives	307	146	90	433	65	108	105	98
Val_false_negatives	349	175	106	474	69	113	110	103

Tabla 9–2. Resultados del algoritmo RMSTOP

Valor final obtenido en el test:

- Precisión: 0.9436
- Valor función pérdidas: 0.216
- N.º falsos positivos: 98
- N.º falsos negativos: 103

Representación gráfica de la evolución del entrenamiento:

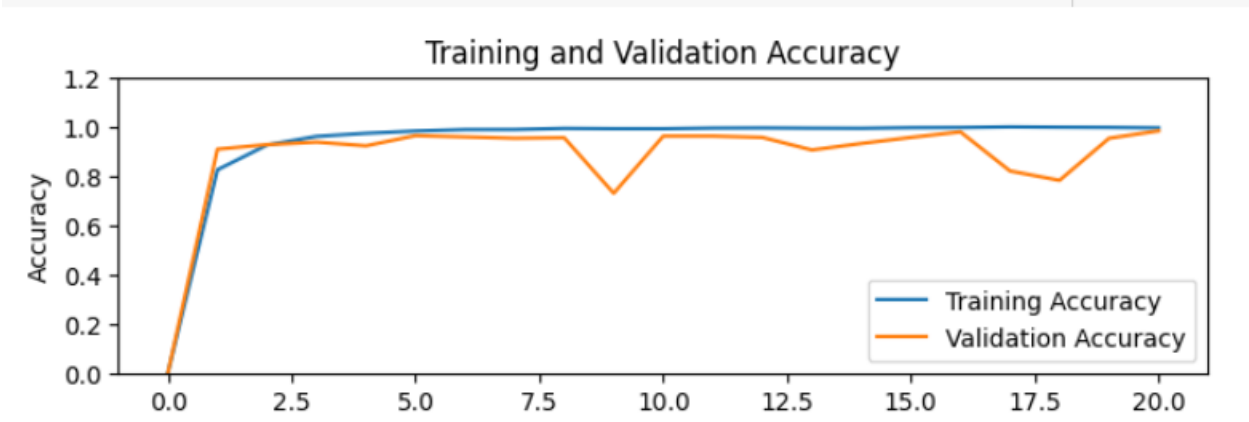


Figura 9-6. Precisión obtenida en el entrenamiento y test usando RMStop.

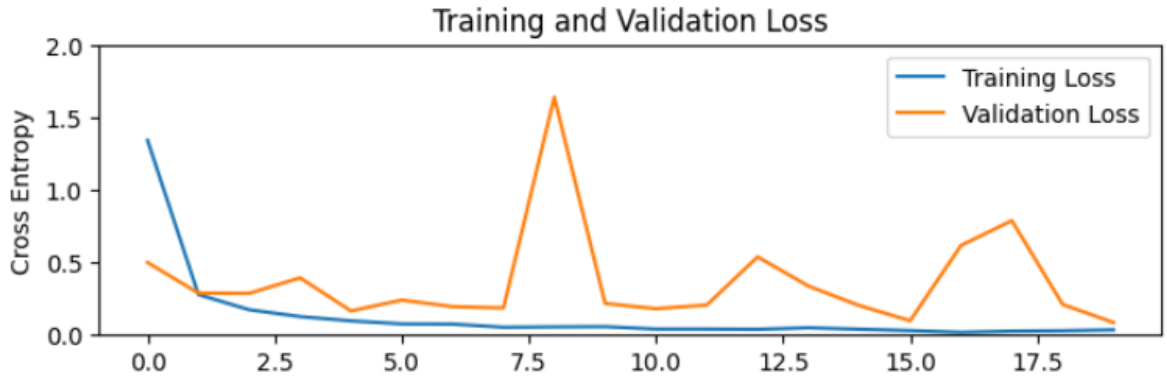


Figura 9-7. Valor de perdidas obtenidas en el entrenamiento y test usando RMStop.

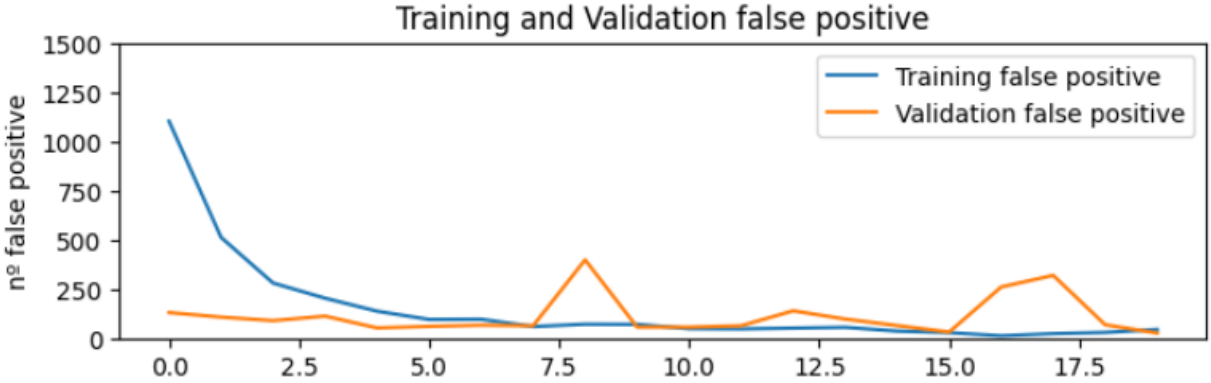


Figura 9-8. Número de falsos positivos obtenidos en el entrenamiento y test usando RMStop.

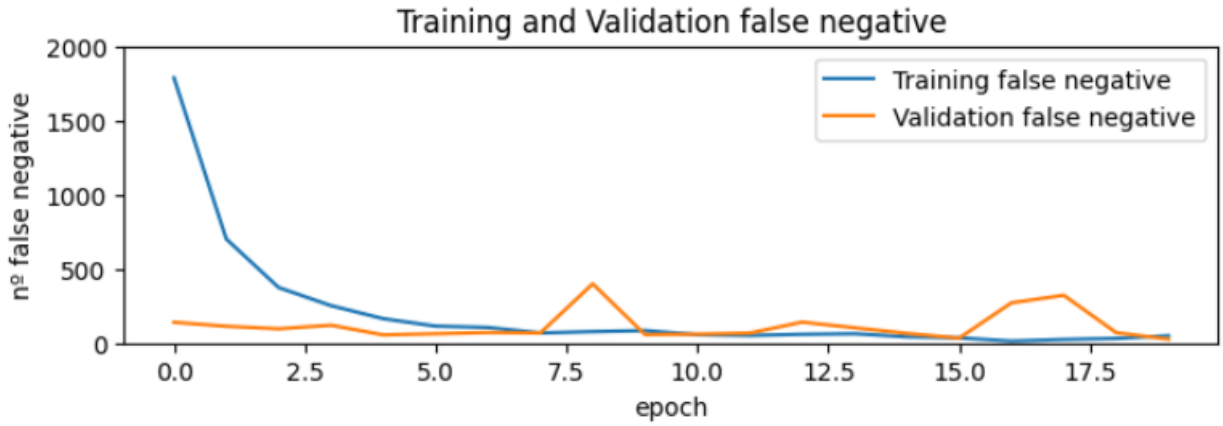


Figura 9-9. Número de falsos negativos obtenidos en el entrenamiento y test usando RMStop.

SGD CON MOMENTUM

Resultado obtenido:

N.º epoc	1	2	3	4	5	10	15	20
Loss	4.996	4.2492	3.7597	3.3544	2.9896	1.7400	1.0467	0.6219
Acuracy	0.7384	0.8766	0.9165	0.9397	0.9605	0.9941	0.9901	0.998
False_positves	1401	810	571	436	290	46	45	9
False_negatives	2749	1204	792	573	363	53	47	10
Val_loss	4.5495	4.2450	4.5131	5.7953	2.9617	1.7903	1.445	0.7064
Val_acuracy	0.8339	0.8039	0.6392	0.4847	0.9406	0.9465	0.9587	0.9626
Val_false_positives	179	266	510	756	120	106	85	56
Val_false_negatives	292	306	547	760	154	111	83	55

Tabla 9–3. Resultado del algoritmo SGD con momento

Valor final obtenido en el test:

- Precisión: 0.9626
- Valor función pérdidas: 0.706
- N.º falsos positivos: 83
- N.º falsos negativos: 55

Representación gráfica de la evolución del entrenamiento:

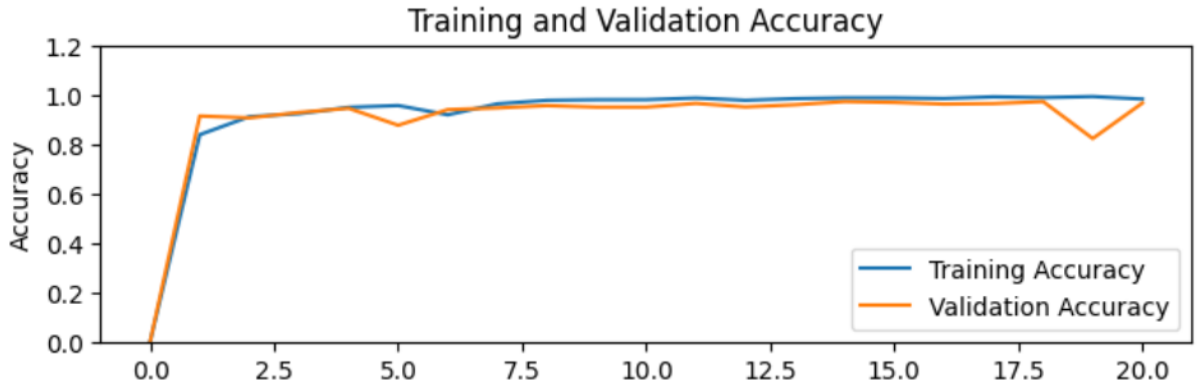


Figura 9-10. Precisión obtenida en el entrenamiento y test usando SGD con momento.

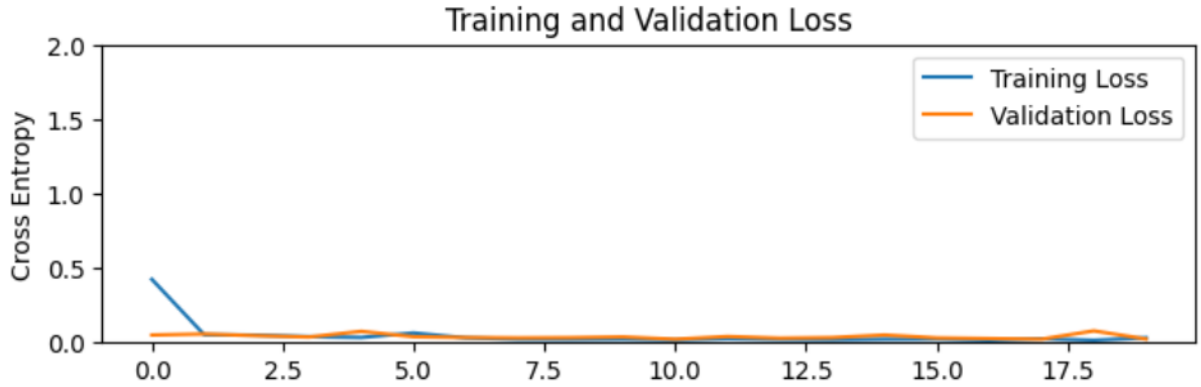


Figura 9-11. Valor de perdidas obtenida en el entrenamiento y test usando SGD con momento.

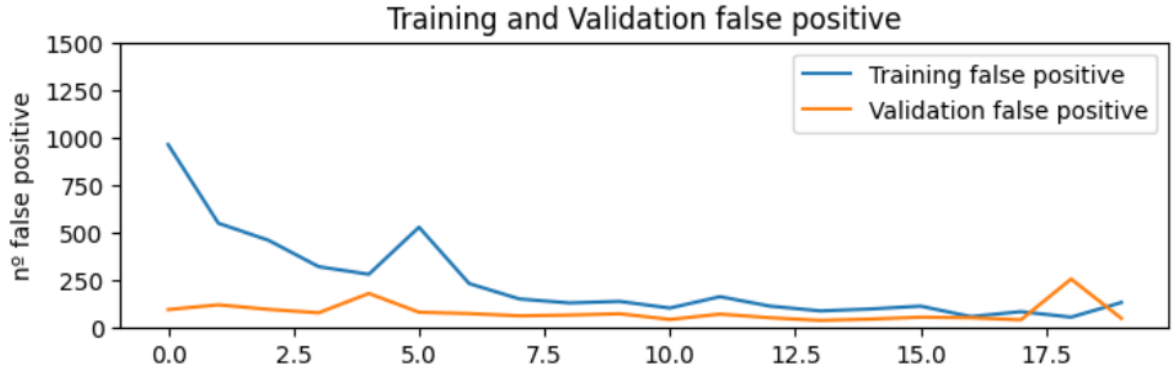


Figura 9-12. Número de falsos positivos obtenida en el entrenamiento y test usando SGD con momento

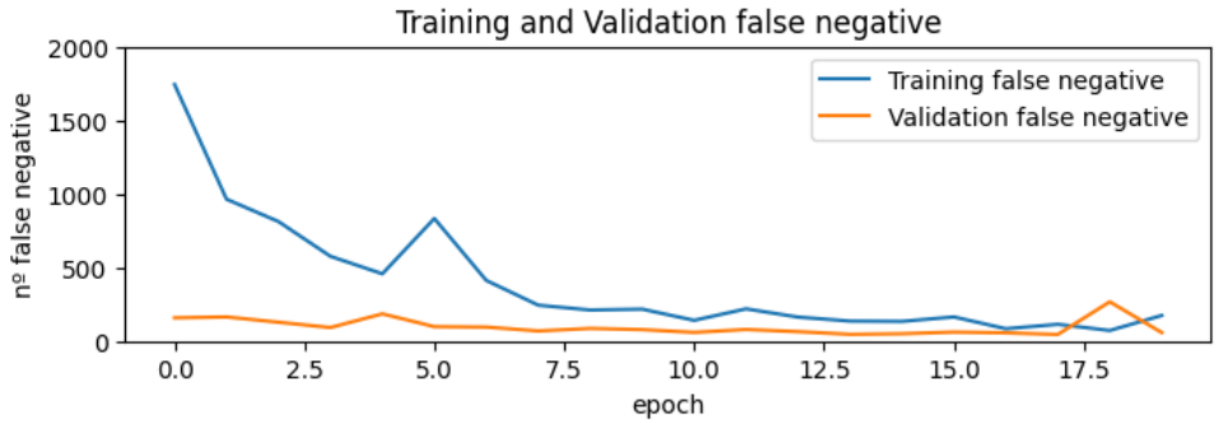


Figura 9-13. Número de falsos positivos obtenida en el entrenamiento y test usando SGD con momento.

RESULTADO FINAL

Como se puede observar el optimizador que mejor desempeño tiene es Adam. Es el que mejor relación tanto de precisión, valor de función de pérdidas y número de falsos/positivos negativos tiene.

9.2 FUNCIÓN DE PERDIDAS

Uno de los pasos más importante a la hora de desarrollar una aplicación es elegir la función de perdidas. Esta nos indicará lo buena o mala que están siendo las predicciones y será un punto crítico a la hora de optimizar nuestro modelo. Una mala elección de la función de perdidas puede hacer que nuestro modelo no se comporte de la manera esperada.

Pasaremos a ver cuál es la mejor función de pérdidas que se ajusta a nuestro modelo, se evaluarán las 2 siguientes: entropía cruzada (Cross entrophy) y error cuadrático (Means Squared Error) medio.

Para ello en cada simulación se utilizarán los siguientes hyperparámetros y optimizador:

- Optimizador: ADAM
- Número de epoch: 20
- Tamaño del batch: 32
- Training y test distribución: 85% de los datos para entrenamiento y 15% test
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

Cross entrophy

Resultado obtenido:

N.º epoc	1	2	3	4	5	10	15	20
Loss	1.7854	0.1654	0.3082	0.2106	0.1653	0.1084	0.0939	0.0839
Acuracy	0.8103	0.9030	0.9279	0.9506	0.9661	0.9813	0.954	0.9871
False_positves	1192	652	510	323	240	142	100	92
False_negatives	1848	957	666	467	310	167	110	120
Val_loss	0.6649	0.4156	0.2859	0.2523	0.6256	0.2135	0.1662	0.1559
Val_acuracy	0.8747	0.9013	0.9312	0.8645	0.9074	0.9523	0.9543	0.9880
Val_false_positives	146	127	92	82	188	116	60	46
Val_false_negatives	205	166	107	100	206	139	70	50

Tabla 9–4. Resultados con función de perdidas Cross entrophy

Valor final obtenido en el test:

- Precisión: 0.988
- Valor función pérdidas: 0.1559
- N.º falsos positivos: 46
- N.º falsos negativos: 50

Representación gráfica de la evolución del entrenamiento:

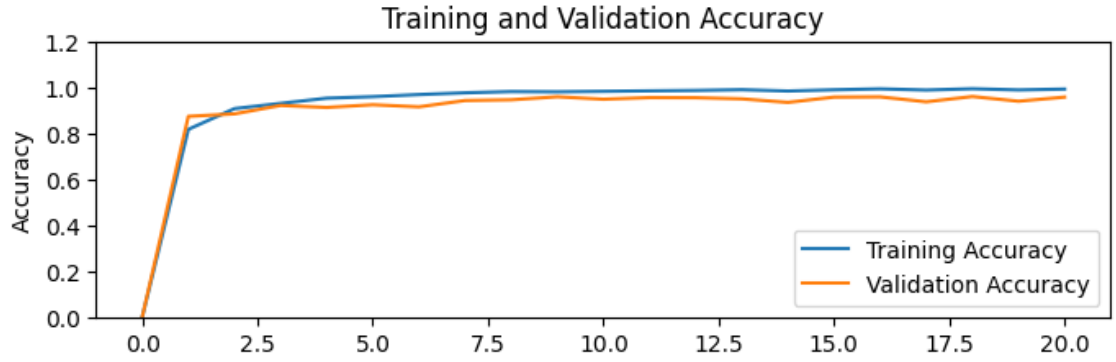


Figura 9-14. Precisión obtenida en el entrenamiento y test usando Cross entrophy

.

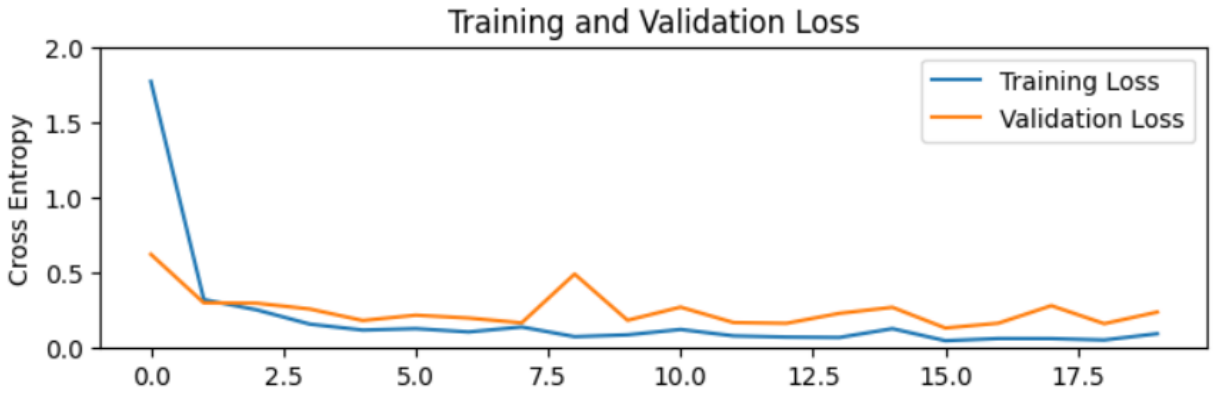


Figura 9-15. Perdidas obtenidas en el entrenamiento y test usando Cross entrophy.

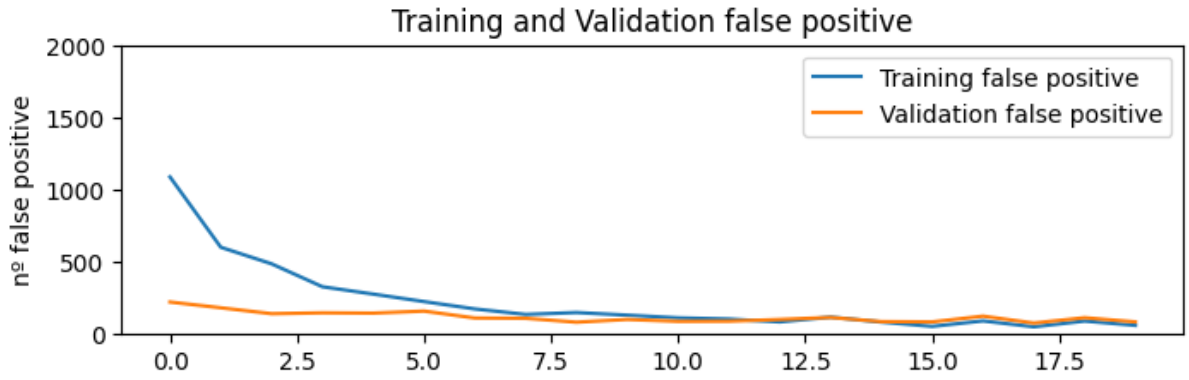


Figura 9-16. Número de falso positivo obtenida en el entrenamiento y test usando Cross entrophy.

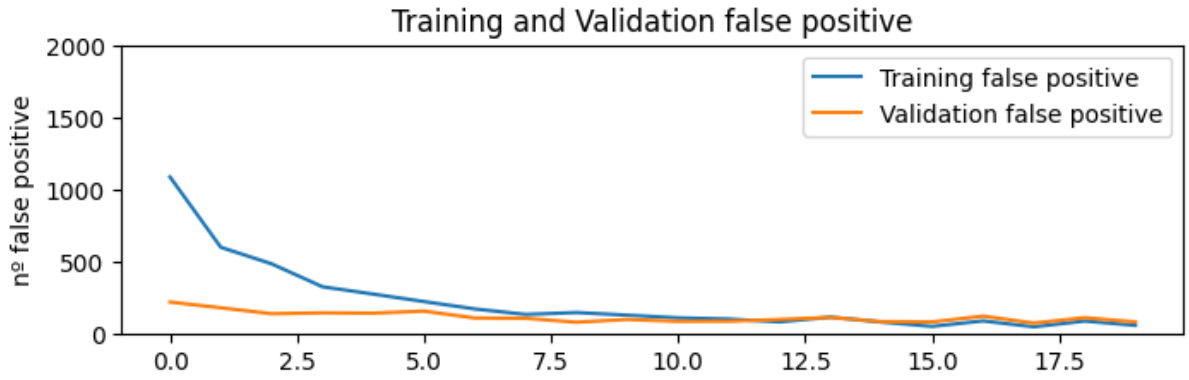


Figura 9-17. Número de falso negativo obtenida en el entrenamiento y test usando Cross entrophy

.

Mean squared error

Resultado obtenido:

N.º epoc	1	2	3	4	5	10	15	20
Loss	0.4611	0.0617	0.0494	0.0452	0.0362	0.0212	0.0146	0.0113
Acuracy	0.7603	0.8820	0.8107	0.8278	0.8441	0.0212	0.0146	0.0113
False_positves	1531	731	555	456	352	155	98	57
False_negatives	2319	1231	915	737	570	263	143	98
Val_loss	0.0711	0.0806	0.579	0.0467	0.0463	0.0364	0.0269	0.0266
Val_acuracy	0.8577	0.8373	0.8775	0.9176	0.9258	0.9368	0.9439	0.9568
Val_false_positives	155	197	120	99	85	57	53	50
Val_false_negatives	235	286	238	140	128	85	65	58

Tabla 9–5. Resultado con función de perdidas en squad error

Valor final obtenido en el test:

- Precisión: 0.9568
- Valor función pérdidas: 0.0266
- N.º falsos positivos: 50
- N.º falsos negativos: 58

Representación gráfica de la evolución del entrenamiento:

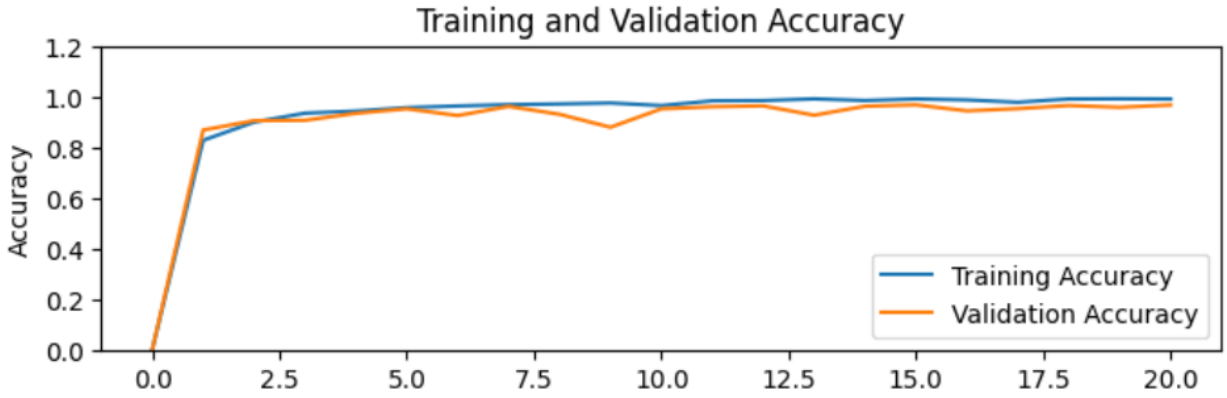


Figura 9-18. Precisión obtenida en el entrenamiento y test usando mean squad error.

Figura 9-19. valor de perdidas obtenido en el entrenamiento y test usando mean squad error.

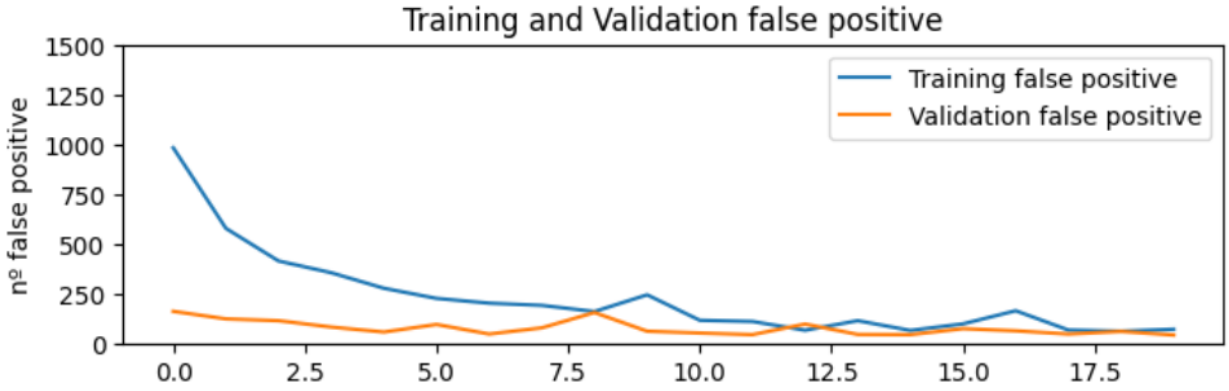


Figura 9-20. Número de falsos positivos obtenidos en el entrenamiento y test usando mean squad error.

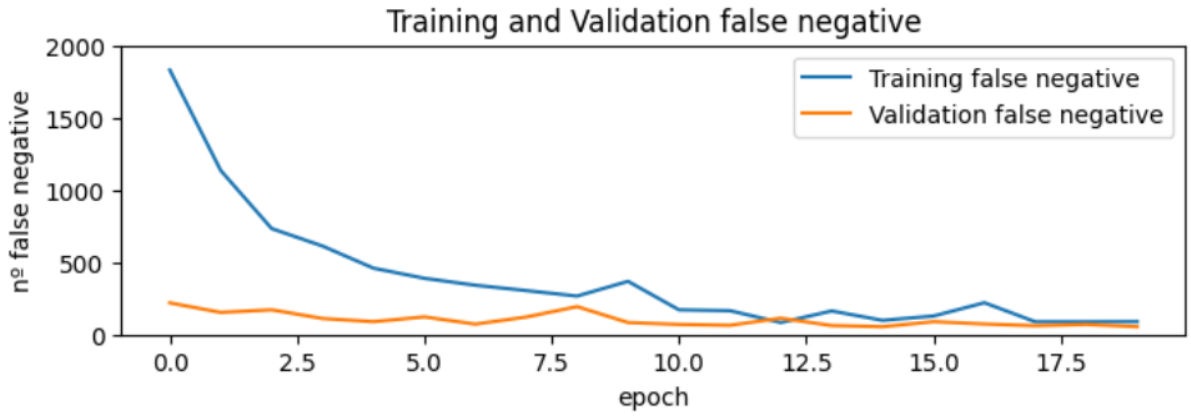


Figura 9-21. Número de falsos negativos obtenidos en el entrenamiento y test usando mean squad error.

RESULTADO FINAL

Ambas funciones tienen un desempeño bastante buen y podrían ser útiles para el proyecto, pero la elección final va a ser la función categorical cross entrophy. Tiene una precisión prácticamente igual y prácticamente tienes los mismos falsos/positivos negativos, sin embargo, lo principal que nos decanta elegir esta función ante la otra es que la función mean Squared error esta más orientada a problemas de regresión lineal

9.3 Hiperparámetros

Por último, deberemos elegir qué valor de los hiperparámetros elegir, discutiremos que valor será el óptimo para el learning rate y para batch size.

LEARNING RATE

Para averiguar este hiperparámetro estos serán los valores utilizados en la simulación

- Optimizador: ADAM
- Función de perdidas: Categorical Cross Entrophy
- Número de epoch: 20
- Tamaño del batch: 32
- Training y test distribución: 85% de los datos para entrenamiento y 15% test
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

Learning-rate	0.1	0.01	0.001	0.0001	0.00001
Loss	1.3691	0.7052	0.0544	0.0653	2.010
Acuracy	0.2869	0.7405	0.9933	0.9912	0.9929
False_positves	0	2065	51	65	51
False_negatives	8323	2236	65	79	72
Val_loss	1.3872	0.3300	0.1616	0.1900	2.0660
Val_acuracy	0.2672	0.9401	0.9598	0.9585	0.9489
Val_false_positives	0	64	51	56	71
Val_false_negatives	1469	110	61	64	78

Tabla 9–6. Cálculo del valor óptimo para learning rate

RESULTADO FINAL

Como se puede observar los mejores valores del learning rate son: 0.001 y 0.0001. Para nuestro caso nos quedaremos con el valor más grande ya que permite realizar un poco más de actualización de los parámetros que el valor consecutivo.

BATCH SIZE

Para averiguar este hiperparámetro estos serán los valores utilizados en la simulación

- Optimizador: ADAM
- Función de perdidas: Categorical Cross Entrophy
- Número de epoch: 20
- Learning rate: 0.001
- Training y test distribución: 85% de los datos para entrenamiento y 15% test
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

El tamaño del batch se deberá escoger en múltiplos de 2.

Batch size	16	32	64	128
Loss	0.1126	0.0544	0.1444	0.0177
Acuracy	0.9839	0.9933	0.9785	0.9992
False_positves	121	51	149	6
False_negatives	149	65	218	7
Val_loss	0.1473	0.1616	0.1774	0.3879
Val_acuracy	0.9748	0.9598	0.9564	0.8856
Val_false_positives	35	51	62	160
Val_false_negatives	39	61	65	175

Tabla 9–7. Cálculo valor óptimo batch size

RESULTADO FINAL

Como se puede observar tanto el valor de 32 como 64 nos ofrecen un funcionamiento similar pero el tamaño 32 es un poco mejor y por eso va a ser el seleccionado para el proyecto.

AUMENTO DE DATOS

Otro punto importante en la aplicación es averiguar cuanto debemos aumentar los datos para que nos ayuden a mejorar el rendimiento de la aplicación sin que estos

Para averiguar la cantidad de datos que debemos de aumentar utilizaremos estos valores para el entrenamiento:

- Optimizador: ADAM
- Función de perdidas: Categorical Cross Entrophy
- Número de epoch: 20
- Learning rate: 0.001
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

Aumento n veces	3	4	5	6
Loss	0.0839	0.0628	0.0544	
Acuracy	0.9871	0.9914	0.9942	
False_positves	92	88	75	
False_negatives	120	98	89	
Val_loss	0.1559	0.1705	0.1397	
Val_acuracy	0.9880	0.9729	0.9767	
Val_false_positives	46	49	55	
Val_false_negatives	50	52	58	

Tabla 9–8. Cálculo valor óptimo aumento de datos

Donde N.º de aumento hace referencia al número de que se aumentara, es decir, tam_original_data_set*Aumento

RESULTADO FINAL

Solo se ha podido aumentar hasta 5 veces el número de datos de entrenamiento a partir de 6 supera los recursos máximos de Google colab. Los valores recomendados para el aumento de datos suelen ser entre 3 y 10 por eso estos valores seleccionados. Como el resultado de 3 es suficientemente bueno respecto a los demás y es el que menos coste computacional requiere, este será el valor escogido

DISTRIBUCIÓN DE DATOS

Otro punto importante a la hora de diseñar una aplicación es decidir qué porcentaje de datos se utilizarán para training y cuantos para test.

Para averiguar este el porcentaje

- Optimizador: ADAM
- Función de perdidas: Categorical Cross Entrophy
- Número de epoch: 20
- Learning rate: 0.001
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

Training/test percent	80/20%	85/15%	90/10%
Loss	0.1125	0.0679	0.0821
Acuracy	0.9820	0.9903	0.9877
False_positves	128	67	99
False_negatives	152	90	119
Val_loss	0.2345	0.2118	0.2564
Val_acuracy	0.9648	0.9571	0.948
Val_false_positives	66	60	47
Val_false_negatives	73	66	58

Tabla 9–9. Distribución de datos óptimo

Resultado Final

El resultado es muy parecido en todas las distribuciones, nuevamente modificar un poco estos valores no supone un gran cambio en los resultados de la aplicación. Nos quedaremos con el valor de 85/15%.

9.4 Resultado final

Tras discutir cual es la mejor elección de optimizador, función de pérdidas y que valor de hiperparámetros se ajusta mejor a nuestro modelo vamos a realizar una ejecución final con todo esto, pero esta vez será un entrenamiento bastante largo para saber hasta dónde puede llegar nuestra red.

Los valores serán:

- Optimizador: ADAM
- Función de perdidas: Categorical Cross Entrophy
- Número de epoch: 100
- Learning rate: 0.001
- Aumento de 3 veces el tamanño del data set
- Training y test distribución: 85% de los datos para entrenamiento y 15% test
- Añadiremos las capas anteriormente definidas y las entrenaremos
- Aplicaremos fine tunning a partir del último bloque del stage 5 de la red, el resto de las capas congelas.

Nº epoc	1	10	25	50	70	80	90	100
Loss	1.7673	0.1451	0.0335	0.0474	0.0339	0.0467	0.0277	0.053
Acuracy	0.8128	0.9769	0.9974	0.9938	0.9960	0.9941	0.9956	0.9939
False_positves	1150	169	22	47	26	43	36	46
False_negatives	1887	207	26	57	37	53	39	56
Val_loss	0.7833	0.2863	0.2872	0.3051	0.1270	0.1211	0.0876	0.1456
Val_acuracy	0.8550	0.9462	0.9489	0.9510	0.9796	0.9715	0.9850	0.9775
Val_false_positives	195	75	69	35	29	29	22	30
Val_false_negatives	225	87	78	41	31	31	28	34

Tabla 9–10. Resultado en cada en los epochs de entrenamiento

Representación gráfica:

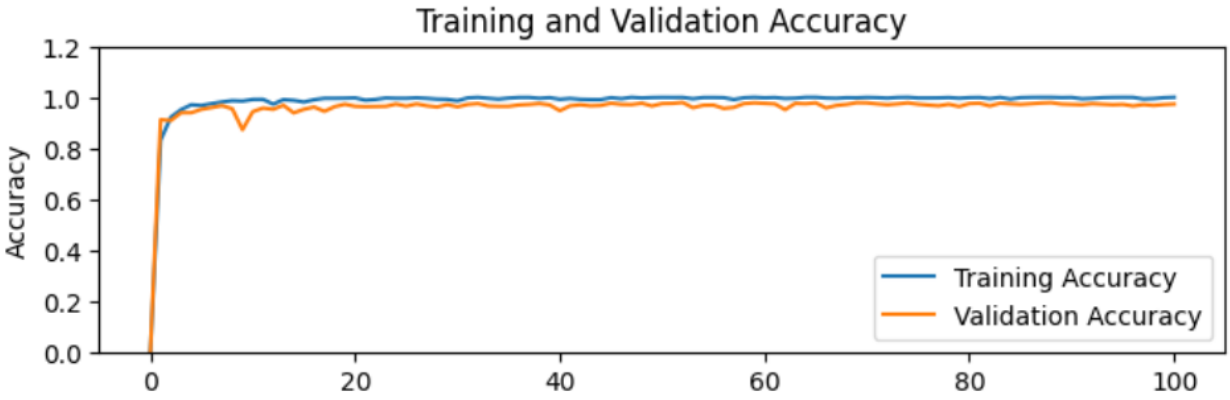


Figura 9-22. Representación final obtenido de precisión.

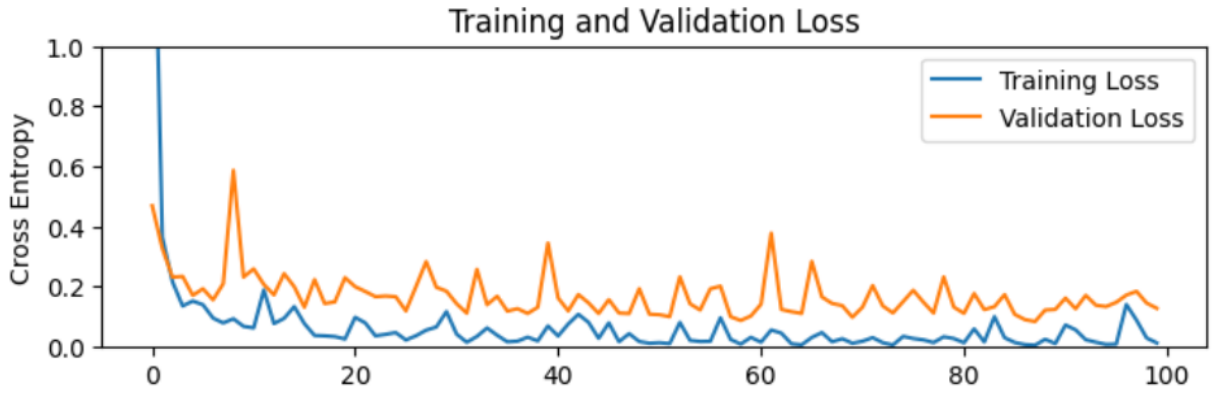


Figura 9-23. Representación final obtenido de perdidas.

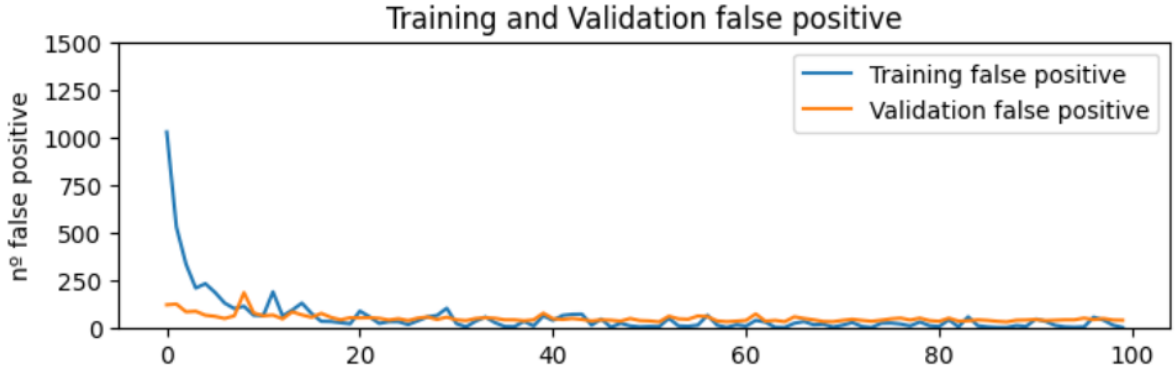


Figura 9-24. Representación final del número de falsos positivos obtenidos.

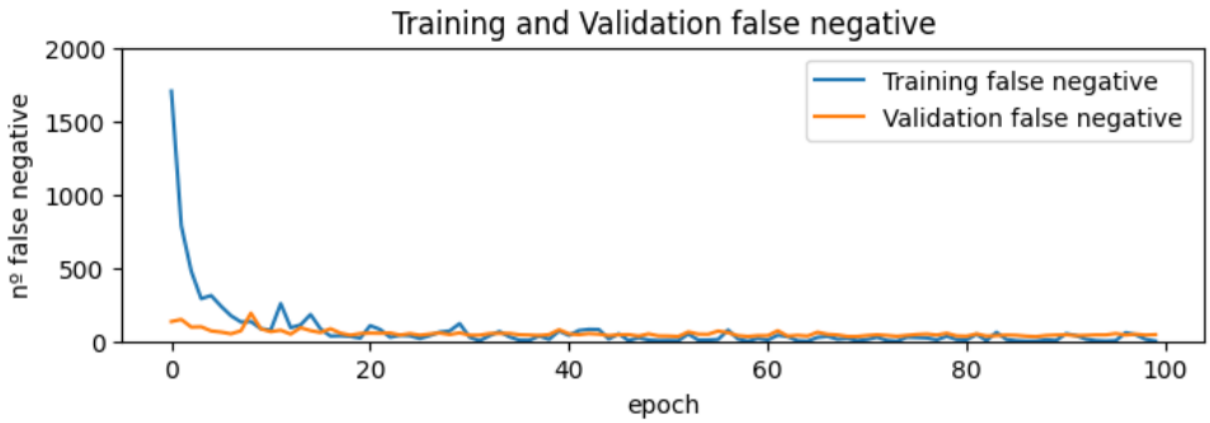


Figura 9-25. Representación final del número de falsos negativos obtenidos.

Como se puede observar a partir del epoch 20 la red no produce más mejoras y los valores empiezan a oscilar. No merece la pena entrenar mucho más la red siendo 20 epoch los necesarios para alcanzar el máximo rendimiento de nuestra red.

Los resultados finales obtenidos han sido:

Resultado final	
Loss	**0.053**
Acurracy	**0.9939**
False_positives	46
False_negatives	**56**
Val_loss	**0.1456**
Val_accurracy	**0.9775**
Val_false_positives	30
Val_false_negatives	**64**

Tabla 9–11. Resultado final de la aplicación

10 FUTURAS MEJORAS

Como en todos los proyectos existen mejoras y cambios que se pueden realizar para mejorar el funcionamiento del mismo, nuestro proyecto no es diferente y pasaremos a describir cual pueden ser las posibles mejoras que se podrían implementar:

1. Cambiar el entorno de desarrollo por otro con más recursos, nuestro entorno es Google colab utilizado de forma gratuita. Una posible mejora sería utilizar la versión premium que es de pago, otra posible solución, pero es demasiado cara y no aporta lo suficiente en relación al coste económico que implica es comprar un hardware de alto nivel orientado al procesamiento de imágenes e información.
2. Utiliza una arquitectura de red más profunda. Un cambio a una mejor arquitectura de red puede mejorar el funcionamiento de nuestra aplicación, pero esto implica que tenemos que tener un entorno de desarrollo mejor que el actual.
3. Implementar segmentación de imagen, esto es una técnica para detectar en que zona se encuentra el tumor, de esta forma, además de indicar que tumor tiene mostraría la zona donde se encuentra.
4. Cambiar el conjunto de datos de entrenamiento por uno que provenga de un hospital o centro de salud donde nuestra aplicación tiene intención de operar. Nuestro conjunto de entrenamiento es bastante bueno, pero al utilizar datos que provenga del sitio final de donde se va a utilizar nuestra aplicación podría ayudarnos a ver como nuestra aplicación funciona en un entorno real.
5. Introducir una interfaz de usuario para la aplicación de generación de diagnósticos para que sea más intuitiva. También podríamos crear una aplicación para móviles o tablets que permitan a través de una foto de la resonancia magnética generar un diagnostico
6. Añadir más información al diagnóstico del peciento como podría ser: pruebas anteriormente realizadas, posibles tratamientos que debería de realizar el paciente o en qué estado se encuentra el cáncer.

Estas son algunas de las mejoras que se podrían aplicar a nuestro proyecto, también podría utilizar esta misma arquitectura para otro tipo de proyectos de detección de imagen, pero se debería de ajustar los hiperparámetros que se han utilizado, así como mejorar algunas funciones para que se ajusta mejor a la aplicación final que se desee mejorar

11 CONCLUSIONES

COmo se ha podido observar durante la realización del proyecto, una aplicación en Deep learning no es algo trivial. Se necesita una gran cantidad de conocimientos previos para poder llevar a cabo un proyecto de estas características, además, cada aplicación puede llegar a ser distinta por lo que no todas las arquitectura y maneras de trabajar funcionan siempre. Si bien es cierto, que siempre se pueden tomar ideas de otras aplicaciones parecidas, pero para conseguir un buen funcionamiento será necesario crear un proyecto más personalizado.

En mi caso me ha sido de gran ayuda poder realizar los cursos de OpenAI, por eso también he decido orientar el tfg con una breve introducción al Deep learing, para que cual quiera que no sea un experto en el tema pueda entender de forma general todo el proyecto.

Respecto a los resultados obtenidos puedo decir que estoy bastante orgulloso de lo que se ha logrado. Se ha conseguid una precisión cercana al 98%, lo que es bastante alto paro una aplicación de este ámbito. Aunque hayamos tenido problemas con la cantidad de recursos necesarios para poder desarrollar el proyecto, se ha logrado un gran resultado.

Para finalizar me gustaría hablar sobre las implicaciones que puede tener una aplicación de este tipo. Es cierto que mucha gente ponga en duda el uso de la inteligencia artificial y que esta va a causar una gran pérdida de trabajo, pero gracias a este tipo de aplicaciones podremos aligerar el proceso de diagnosticación de los pacientes. Muchas veces no existen una infraestructura lo suficientemente grande como para poder atender a todo el mundo siendo necesario utilizar aplicaciones como esta para poder resolver el gran problema de las colas de espera en el sistema sanitario.

REFERENCIAS

[1] World Cancer Report Cancer research for cancer prevention. Availeble*:* https://publications.iarc.fr/586

[2] Open AI <Redes neuronales y aprendizaje profundo>. available: https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning

[3] Open AI <Improving Deep Neuronal Networks> Hyperparameter Tuning, Regularization and Optimization> . Availeble: https://www.coursera.org/learn/deep-neural-network?specialization=deep-learning

[4] Open AI <Structuing Machine Learning Projects>. Available https://www.coursera.org/learn/machine-learning-projects?specialization=deep-learning

[5] Open AI <Convulational Neural Networks>. Available https://www.coursera.org/learn/convolutional-neural-networks?specialization=deep-learning

[6] Open AI <modelos secuencias>. Available https://www.coursera.org/learn/nlp-sequence-models?specialization=deep-learning

[7]Jordi Torres <Deep learning Introducción practica con Keras> Available: https://books.google.es/books?hl=es&lr=&id=ju1mDwAAQBAJ&oi=fnd&pg=PA78&dq=que+es+el+deep+learning&ots=k_VcLD54UT&sig=jkg8tLwxVDfhmuV_Gq0FUcNGF4A#v=onepage&q=que%20es%20el%20deep%20learning&f=false

[8] Michael Nielsen <Neural Networks and Deep Learning >,Michael Nielsen. Available: https://www.ise.ncsu.edu/fuzzy-neural/wp-content/uploads/sites/9/2022/08/neuralnetworksanddeeplearning.pdf

[9] . Charu C. Aggarwal <Training Deep Neural Networks. Available>: https://link.springer.com/chapter/10.1007/978-3-319-94463-0_3

[10] Daniel Svozil , Vladimir Kvasnicka, Jiri Pospichal. <Introduction to multi-layer feed-forward neural networks>. Available: https://www.sciencedirect.com/science/article/pii/S0169743997000610

[11] Neural Network and Deep Learning. Charu C Aggarwal. https://link.springer.com/book/10.1007/978-3-319-94463-0

[12] Arohan Ajit, Koustav Acharya, Abhishek Samanta. <A review Of convulational neuronal networks>. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9077735

[13] Google COlab <todo lo que necesitas saber>. Available: https://geekflare.com/es/google-colab/

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun <Deep residual Learning for imagen Recognation> Available: https://arxiv.org/pdf/1512.03385.pdf

[15] Zhen Hu, et al. <Handling Vanishing gradient problem using artificial derivate>. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9336631

[16] . George Philipp, Dawn Song, Jaime G. Carbonell <The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions>. Available: https://arxiv.org/pdf/1712.05577.pdf

[17] Hamza Rafiq Almadhoun and Samy S. Abu Naser <Detection of Brain Tumor Using Deep Learning>. Available: https://philpapers.org/rec/ALMDOB

[18] Mohamed A. Naser , M. Jamal Deen <Brain tumor segmentation and grading of lower-grade glioma using deep learning in MRI images>.
Available:https://www.sciencedirect.com/science/article/pii/S0010482520301335

[19] Hao dong, et al <Automatic Brain Tumor Detection and Segmentation Using U-Net Based Fully Convolutional Networks>. Available: https://arxiv.org/abs/1705.03820

[20] Nima Tajbakhsh, et al <Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?>. Available: https://arxiv.org/abs/1706.00712

[21] Tina Jacob.<Vanishing Gradien Problem, Explained>. Available: https://www.kdnuggets.com/2022/02/vanishing-gradient-problem.html#:~:text=When%20there%20are%20more%20layers,this%20the%20vanishing%20gradient%20problem.

[22] Ian Goodfellow, et al. <Deep learning>. Available: https://books.google.es/books?hl=es&lr=&id=omivDQAAQBAJ&oi=fnd&pg=PR5&dq=deep+learning&ots=MNU2emtBNY&sig=x25fte4LNVWkHpm3aCn1N7SKSBo#v=onepage&q=deep%20learning&f=false

[23] Jiahui Yu, Linjiue Yang, et al. <Slimmable Neural Network>. Available: https://arxiv.org/abs/1812.08928

[24] Fuzhen Zhuang , et al. <A comprehensive Survey on Transfer Learning>. Available: https://ieeexplore.ieee.org/abstract/document/9134370

[25] Connor Shorten <A survey on image Data Augmentation for Deep Learning>. Available : https://journalofbigdata.springeropen.com/counter/pdf/10.1186/s40537-019-0197-0.pdf

[26] Douglas M. Hawkins <The Problem of Overfitting>. Available: https://pubs.acs.org/doi/full/10.1021/ci0342472

[27] ImagenNet <Database of ImageNet>. Available: https://www.image-net.org/

[28] TensorFlow <libraries>. Available: https://www.tensorflow.org/resources/libraries-extensions?hl=es-419

[29] Keras <Keras>. Available: https://keras.io/

[30] Numpy <Numpy Documentation>. Available: https://numpy.org/doc/stable/reference/

[31] Scikit-learn <sklearn library documentation>. Available: https://scikit-learn.org/stable/