UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Estudio de utilización de Redes Neuronales Convolucionales en Tensorflow para Segmentación de Imagen Médica

Autor:

D. Roberto García Higuera

Tutor:

Dr. D. Mario Martínez Zarzuela

Valladolid, 13 de julio de 2020

TÍTULO:	**Estudio de utilización de Redes Neuronales Convolucionales en Tensorflow para Segmentación de Imagen Médica**
AUTOR:	**D. Roberto García Higuera**
TUTOR:	**Dr. D. Mario Martínez Zarzuela**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Dra. Dª. Miriam Antón Rodríguez**
SECRETARIO:	**Dr. D. Mario Martínez Zarzuela**
VOCAL:	**Dr. D. David González Ortega**
SUPLENTE 1:	**Dr. D. Carlos Gómez Peña**
SUPLENTE 2:	**Dr. D. Francisco J. Díaz Pernas**

FECHA:	**13 de julio de 2020**
CALIFICACIÓN:	

Agradecimientos

En primer lugar me gustaría agradecer al Grupo de Telemática e Imagen la oportunidad de poder haber realizado este trabajo con ellos y en especial a Dr. D. Mario Martínez Zarzuela por guiarme y apoyarme hasta el final de esta etapa en mi vida.

A todos mis compañeros de viaje a lo largo de esta carrera y a mis amigos los cuales han sido un pilar fundamental a lo largo de todos estos años donde me han apoyado en los momentos difíciles y en las decisiones que he tomado.

Por último a mi familia, a mis padres y mi hermano por todo el apoyo, motivación y ayuda que me han proporcionado durante todos estos años y especialmente durante la realización de este trabajo.

Resumen

El objetivo de este Trabajo Fin de Grado consiste en el desarrollo de un estudio de utilización de redes neuronales convolucionales vía software para la segmentación de imágenes biomédicas procedentes de una base de datos (dataset) de resonancias magnéticas con el fin de facilitar el diagnóstico médico.

A través de los diferentes puntos de vista a la hora de enfocar la problemática de la segmentación de las imágenes elegiremos la mejor opción para abordar el estudio.

Estudiaremos la teoría de las redes neuronales, donde nos centraremos en las operaciones que se realizarán en las redes convolucionales. Tras esto, veremos un modelo real para nuestro problema donde se realizarán las distintas modificaciones, con el fin de implementar de una forma funcional un modelo de red

Todo este estudio se basa en el uso de la Inteligencia Artificial, sustentada en el concepto de Deep Learning, entendiendo los conceptos básicos de dicha teoría, y desarrollándoles especialmente a través de TensorFlow.

Expondremos los fundamentos teóricos y matemáticos en los que basamos nuestro diseño de redes neuronales, estudiaremos la mejor manera de trabajar nuestra base datos con nuestro sistema de Deep Learning, el procesamiento más correcto de los datos y a continuación expondremos las decisiones tomadas para el diseño de las diferentes redes usadas, realizando diferentes pruebas para ajustar los parámetros para conseguir los mejores resultados.

Finalmente trabajaremos con las redes diseñadas, analizaremos los resultados matemática y gráficamente, valorando si nuestros modelos son funcionales y operativos, fin con el que desarrollamos este estudio de Trabajo de Fin de Grado.

Palabras claves

Inteligencia Artificial, Deep Learning, Redes neuronales convolucionales, resonancias magnéticas, BraTS, U-net, Tensorflow, Tfrecords.

INDICE

INDICE DE FIGURAS:

INDICE DE TABLAS:

BLOQUE 1: INTRODUCCIÓN

1. INTRODUCCIÓN

1.1. LA MOTIVACIÓN

La capacidad evolutiva de las tecnologías ha permitido que a día a día de hoy el *Deep Learning* sea una asignatura para cualquier persona que trabaje con volúmenes de datos.

Mediante el uso de esta tecnología se puede llegar a desarrollar aplicaciones que clasifiquen esos datos o incluso redes que pueden predecir las tendencias a corto plazo.

Todo esto no sería posible sin la capacidad matemática que tienen nuestros ordenadores en el presente, nos permiten realizar modelos de una forma eficaz, ya que en años atrás nos hubiéramos podido plantear este Trabajo Final de Grado.

Este Trabajo Final de Grado se ha desarrollado y gestado en el Grupo de Telemática e Imagen (GTI) de la Universidad de Valladolid donde sus años de experiencia con el procesado de imágenes han marcado las pautas para la correcta implementación de una aplicación desarrollada con las Redes Neuronales Convolucionales para el ámbito biomédico, en un lenguaje de programación conocido como es *Python* y con la herramienta de desarrollo *TensorFlow*.

1.2. LOS OBJETIVOS

A continuación se detallan los requisitos necesarios para la realización de este Trabajo Final de grado, los cuales son:

- El principal objetivo es conseguir la realización completa de una aplicación para segmentación de imágenes biomédicas, que proceden de un *dataset* de resonancias magnéticas (MRI) que contienen cortes con diferentes tejidos de gliomas.
- Utilización de *TensorFlow* como librería principal para el desarrollo de la aplicación de aprendizaje profundo.
- Transformar de una forma correcta el dataset proporcionado para la tarea.
- Generar ficheros de *tfrecords* para los distintos sets (entrenamiento, validación y test) que se utilizaran a lo largo de la aplicación.
- Implementar de una forma funcional un modelo de red para la segmentación de imágenes.
- Analizar distintos modelos de red para la segmentación de imágenes.
- Poder evaluar el modelo o los modelos de predicción durante el test, buscando el mejor resultado entre todas las posibles pruebas detallando si los resultados son favorables o no.
- Documentar el proceso completo de la creación de la aplicación basada en *TensorFlow* con el objetivo de que se pueda adaptar el programa a otros problemas similares o para el uso de otras bases de datos.

1.3. FASES DEL PROYECTO

Las principales fases en la que se va a dividir este Trabajo Final de Grado se explicaran de una forma breve y concisa:

- **Familiarización con el entorno de desarrollo** Antes de comenzar a realizar este Trabajo Final de Grado es conveniente realizar algún tipo de contacto con *Python* y *TensorFlow* ya que la forma de programar se basa principalmente en los grafos y tensores, un concepto bastante extendido para aquellos que usan *TensorFlow*.
- **Búsqueda de una base de datos:** Es necesario encontrar para la aplicación un *dataset* que contenga imágenes biomédicas siendo más precisos, nuestra búsqueda se centrará en un dataset que contenga imágenes de resonancias magnéticas de la cabeza de los pacientes. Este tipo de *dataset* es complicado de encontrar debido a que necesitamos que cada corte que hay en dicho *dataset* debe estar segmentado de la forma más precisa por expertos.
- **Instalación del software para el desarrollo del trabajo:** En este caso para poder realizar cualquier tipo de aproximación al problema es necesario realizar la instalación del entorno de trabajo como la instalación de todas aquellas librerías y herramientas que podamos necesitar a lo largo del trabajo.
- ***Deep Learning* y *TensorFlow* principales conceptos:** Para poder desarrollar la aplicación basado en un sistema *Deep Learning*, lo primero de todo es ser capaces de entender los conceptos básicos de teoría y a partir de este punto hay que estudiar el cómo se puede implementar dichos conceptos, especialmente con *TensorFlow* el cual dispone a día de hoy de una gran cantidad de herramientas.
- **Estudio bibliográfico de trabajos basados en BraTS:** Para poder realizar nuestro sistema de *Deep Learning* el cual pueda trabajar con las imágenes del dataset BraTS, es necesario conocer los principales estudios sobre la base de datos y el *Deep Learning.*
- **Procesamiento de la base de datos:** Una parte importante del problema tras haber realizado las anteriores fases, es el cómo vamos a proporcionar los datos a nuestro sistema de *Deep Learning*, para ello realizaremos un preprocesado debido a su gran volumen.
- **Diseño de las redes neuronales:** Tras tener preprocesados los datos en un formato ideal, a continuación se plantean las decisiones tomadas para el diseño de las diferentes redes usadas.
- **Pruebas realizadas:** Durante el diseño de las diferentes redes que desarrollemos para nuestro sistema, se llevaran a cabo un número de pruebas para poder ajustar los diferentes parámetros, lo que buscamos es saber cuál puede llegar a ser el mejor resultado de los diseños propuestos.

1.4. HARDWARE Y SOFTWARE

En este punto vamos a ver los principales componentes de nuestro ordenador con el que hemos trabajado y las diferentes herramientas de software que hemos utilizado.

1.4.1. HARDWARE

Este proyecto se ha desarrollado principalmente en un ordenador portátil ASUS ZenBook Pro UX550VD con las siguientes especificaciones (ASUS ZenBook Pro UX550VD | Portátiles | ASUS España, 2020):

- Un procesador Intel® Core™ i7-7700HQ el cual tiene las siguientes especificaciones (Procesador Intel® Core™ i7-7700HQ, 2020):
    - Cantidad de núcleos: 4
    - Frecuencia básica del procesador: 2.8 GHz
    - Caché: 6 MB Intel® Smart Cache
    - Cantidad de subprocesos: 8
    - Frecuencia turbo máxima: 3.80 GHz
- Memoria de 8 GB en placa DDR4 2400MHz SDRAM
- Almacenamiento 256 GB SATA3 SSD
- Tarjeta Gráfica NVIDIA® GeForce® 1050 Ti con las siguientes especificaciones (Tarjeta gráfica GeForce GTX 1050, 2020):
    - NVIDIA CUDA®Cores: 768
    - Base Clock: 1290 MHz
    - Boost Clock: 1392 MHz
    - Velocidad de Memoria: 7 Gbps
    - Configuración de Memoria Estándar: 4 GB GDDR5
    - Memoria de Banda Ancha: 112 GB/sec

Tanto el ordenador como sus componentes realizan un trabajo excepcional para el nivel de un notebook ya que no es su mejor uso, pero aun así realiza el entrenamiento de una forma correcta, a mayores se ha tenido que añadir una base refrigeradora Woxter Notebook Cooling Pad 2200 Aluminium la cual tiene un ventilador de 22 cm para reducir la temperatura del notebook, debido a que en ciertas condiciones puede llegar a apagarse el portátil por culpa de la temperatura (Aluminium, 2020).

En el mercado hay portátiles con una memoria mayor y una tarjeta gráfica mejor, pero la condición ideal para desarrollar esta aplicación sería en un ordenador de sobremesa, un servidor dedicado para ello o alguna de múltiples nubes que permiten levantar máquinas virtuales para este tipo de trabajos.

1.4.2. SOFTWARE

El software que utilizaremos en el desarrollo de este Trabajo Final de Grado es el que se muestra a continuación.

Sistema Operativo

La primera elección antes de seguir cualquier paso en esta aplicación fue la del sistema operativo, las dos opciones son bastante conocidas por un lado tenemos Linux el cual es un sistema operativo de tipo *Unix Posix* multiplataforma y software libre (*Linux* y GNU - Proyecto GNU - Free Software Foundation, 2020).

Por otro lado tenemos Windows 10 sistema operativo desarrollado por Microsoft cuyo software no es de código libre y sí que es multiplataforma (Características de *Windows* 10: Qué hay en *Windows* 10 | Microsoft, 2020).

La elección de *Windows* 10 para el desarrollo de la aplicación es por la simplicidad a la hora de instalar el software y no tener problemas con los diferentes drivers de la tarjeta gráfica, además es necesario instalar previamente la herramienta CUDA proporcionada por NVIDIA ya que utilizaremos la GPU por encima de la CPU (CUDA Zone, 2020).

En *Windows* al igual que en *Linux* es necesario generar mediante un programa los diferentes contenedores para poder trabajar con múltiples entornos de programación, es decir, poder tener varias versiones de Python en la misma máquina con la que trabajamos.

Lenguajes de programación

Python es un lenguaje de programación poderoso y fácil de aprender debido a que cuenta con estructuras de datos eficientes y de alto nivel además de una programación secuencial u orientada a objetos. Es bastante intuitivo, tiene interoperabilidad con diferentes bibliotecas y lenguajes de programación como pueden ser C o C++ (1. Introducción — Tutorial de Python 3.6.3 documentation, 2020).

Este lenguaje es multiplataforma se puede utilizar en los siguientes sistemas operativos: los principales como *Mac Os, Unix, MS-DOS y Windows 10* que además en mayo de 2019 se incluyó una actualización en la cual se dispone de una preinstalación para dicho lenguaje además de incorporar las principales bibliotecas y herramientas que se usan a día de hoy.

Creado por van Rossum en 1991 en Holanda y a día de hoy gestionada por la Python Software Foundation. Cuya licencia del lenguaje es PSFL o Python Software Foundation License la cual permite que sea un software libre (General Python FAQ — Python 3.8.2rc2 documentation, 2020).

Desde la experiencia propia este lenguaje de programación tiene una de las mejores características que es su sintaxis nos permite tener el código perfectamente sangrado para una lectura más limpia y rápida.

La versión en la que se ha desarrollado este proyecto es la 2.7

Software de desarrollo para Deep Learning

TensorFlow

TensorFlow, biblioteca de código abierto desarrollada por Google, su principal función es el aprendizaje autónomo a través de sus sistemas que pueden construir y entrenar diferentes redes neuronales que detectan y descifran los diferentes patrones y correlaciones consiguiendo aprendizajes y razonamientos análogos a los humanos (TensorFlow, 2020).

En 2011 el grupo Google Brain desarrollo DistBelief como un sistema de aprendizaje automático, principalmente usaba sus redes neuronales para dicho sistema. La utilización de esta biblioteca se expandió rápidamente a través de diferentes compañías de Alphabet, la cual es una empresa que desarrolla productos y servicios relacionados con Internet, software, dispositivos electrónicos.

Esto permitió que el 9 de Nobiembre de 2015 se creará la segunda generación de un sistema de aprendizaje autónomo, además liberado como software de código abierto. TensorFlow es capaz de correr en múltiples *CPUs, GPUs y TPUs*, además de ser un software multiplataforma ya que está disponible para Linux de 64 bits, *macOs* y para las principales plataformas móviles Android e iOS.

Además a lo largo del año 2019 se anunció la versión alfa de *Tensorflow 2.0*, cuya principal diferencia con las versiones anteriores es su simplicidad y su facilidad de uso ya que utiliza diferentes APIs basadas en otro software de aprendizaje autónomo que es *Keras* (TensorFlow, 2020).

La versión en la que se ha desarrollado este proyecto es la 1.8

Entorno de trabajo

Anaconda

Anaconda es una distribución para los lenguajes *Python* y *R* además esta distribución es libre y abierta, su principal uso es para el aprendizaje autónomo o *Deep Learning* y para todo aquello que trate volúmenes de datos. Esto puede incluir el procesamiento de estos volúmenes los cuales sueles ser bastante pesados, permite el análisis predictivo y computo científico (Anaconda Python/R Distribution - Free Download, 2020).

Aunque su principal función es la simplificación del despliegue en múltiples plataformas y la administración de los paquetes software. Estos paquetes pueden administrarse en diferentes versiones y en diferentes entornos de trabajo, todo ello se realiza mediante la gestión de paquetes conda, el cual es bastante fácil de instalar,

correr y actualizar las diferentes versiones de nuestros paquetes en los diferentes entornos que tengamos.

Entre sus principales paquetes encontramos las principales librerías, APIs que se usaran a lo largo del proyecto como pueden ser *TensorFlow, Scikit, Numpy, etc...*

Esta distribución permite el uso en las principales plataformas *Windows, Linux y MacOs*.

Jupyter Lab

El proyecto es un spin-off de IPython, creado por Fernando Pérez en 2014. *Jupyter* soporta la ejecución de diferentes entornos de trabajo para un abanico amplio de lenguajes de programación. Proyecto Jupyter ha desarrollado diferentes productos para este tipo de tareas (Project Jupyter, 2020).

El primero de ellos es *Jupyter Notebook*, es una aplicación basada en web para programar en un entorno de trabajo con diferentes lenguajes de programación y librerías. Los "Notebook" contienen una lista de celdas de entrada y salida ordenadas, es decir cada celda contiene las líneas de código, esto a priori no parece mucho pero permite la ejecución de partes de código sin necesidad de ejecutar todo el código completo de la aplicación que estamos realizando.

Por otro lado también ha desarrollado JupyterHub el cual es un multiserver para el uso de los notebooks, para múltiples usuarios alojados en una máquina.

Por último JupyterLab es una versión más moderna que mejora la interfaz de los Jupyter Notebooks, además es con esta versión con la que se ha desarrollado el proyecto (Project Jupyter , 2020).

BLOQUE 2: DEEP LEARNIG

2. DEEP LEARNING

2.1. INTRODUCCIÓN

A lo largo de este bloque veremos como el *Deep Learning* es mucho más que una técnica para desarrollar este trabajo final de grado sino que además iremos desglosando cuidadosamente desde la base teórica hasta las diferentes opciones que tomaremos en el diseño de nuestra aplicación.

Principalmente nos centraremos en una segmentación de imágenes para este trabajo junto con el uso de redes neuronales, pero para ello tendremos que ir explicando poco a poco lo que es la Inteligencia Artificial y todas sus subcategorías como son el *Machine Learning* y el *Deep Learning* como podemos ver en la Figura 1 de (Chollet, 2017)

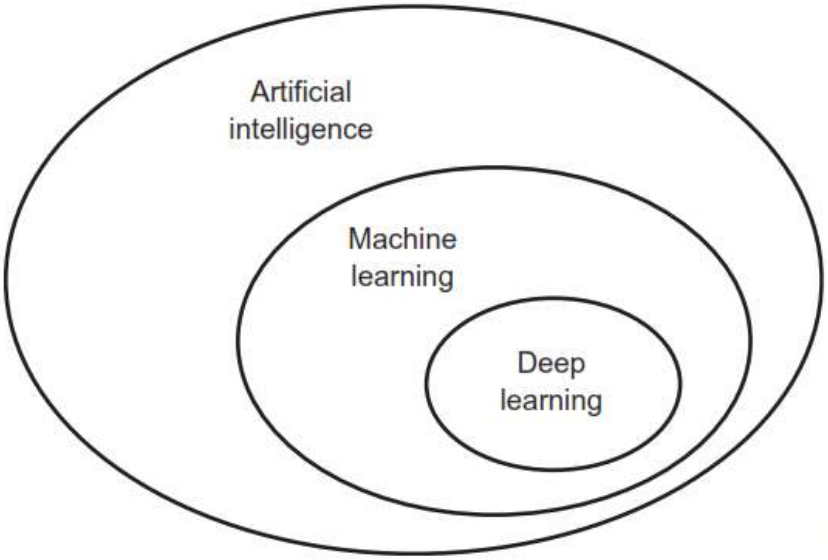

Figura 1. Inteligencia Artificial, Machine Learning, Deep Learning

2.2. HISTORIA DEL DEEP LEARNING

A lo largo de la década de los 1950 un grupo de pioneros del campo de la informática comenzaron a cuestionarse si podían hacer que las máquinas "pensaran", esta pregunta a día de hoy que aún se sigue resolviendo en diferentes ámbitos. Una definición concisa sería la siguiente: *el esfuerzo por automatizar las tareas intelectuales normalmente realizadas por los seres humanos*. La inteligencia abarca mucho más allá de los campos de aprendizaje como son el *Deep Learning* y el *Machine Learning*, abarca también enfoques que no implican ningún tipo de aprendizaje. Por ejemplo los primeros programas de ajedrez incluían reglas de código hechas por los propios programadores lo cual no era aprendizaje automático (Chollet, 2017).

Los expertos creyeron durante bastante tiempo que la inteligencia artificial podía alcanzar el nivel humano haciendo reglas explicitas en sus programas para manipular el conocimiento. Esto se conoce como *IA Simbólica* y fue el principal paradigma desde los años cincuenta hasta finales de la década de los ochenta. Este tipo de inteligencia fue totalmente adecuado para resolver problemas lógicos bien definidos como el ejemplo anterior del ajedrez (Chollet, 2017).

Para nuestro caso como puede ser la clasificación de imágenes, el reconocimiento de patrones en frases, la traducción automática de un lenguaje a otro o el reconocimiento del habla de una persona nacería un nuevo enfoque en lugar de esa inteligencia artificial simbólica, el cual conocemos como aprendizaje automático.

El aprendizaje automático, surge a partir de una cuestión: ¿Es capaz una máquina ir más allá de "lo que sabemos cómo ordenarle que realicé" y que aprenda por sí mismo a realizar una o varias tareas?

Este cambio del paradigma permite cuestionar esas reglas elaboradas por los programadores a mano sino que sea una máquina la que pueda aprender de forma automática estas reglas mirando simplemente los datos. Entonces en la programación clásica los humanos introducían las reglas y los datos para ser procesados de acuerdo a estas reglas y recibían respuestas. Pero con el aprendizaje automático los humanos simplemente introducimos los datos así como las respuestas que esperamos de ellos obteniendo las reglas. Estas reglas pueden ser luego aplicadas a nuevos datos para producir las preguntas originales.

Figura 2. Inteligencia Artificial Simbólica vs Aprendizaje Automático.

En la Figura 2 de (Chollet, 2017) vemos ese cambio producido por la forma de pensar y el cambio de paradigma, entonces un sistema basado en aprendizaje automático se entrena en vez de programarlo explícitamente como ocurría en un pasado.

A lo largo de los años 90 y principio del siglo XXI el *Machine Learning* empezó a crecer en popularidad logrando convertirse rápidamente en el subcampo de la inteligencia artificial más popular y eficiente. Este es debido por la mejoría del hardware en las máquinas al igual que había bases de datos grandes de las que se disponían para trabajar. Además numerosas empresas empiezan a transformarse cambiando su modelo de negocio hacia el manejo de los datos donde añadieron múltiples técnicas de *Machine Learning* para la venta de sus productos o la forma de analizar sus procesos y sus servicios consiguiendo ventajas competitivas con el resto de la competencia.

Durante esta época como hemos comentado aparte de que las empresas usaran los datos, se empezó a utilizar dicha tecnología para la minera de datos, el uso de un software adaptable y diferentes aplicaciones web basadas en aprendizaje de texto y de idiomas (Chollet, 2017).

Finalmente en esta década uno de los padres del Deep Learning "Geoffrey Hinton" realizo múltiples investigaciones en los algoritmos del Machine Learning pero no fue hasta que acuñó el término de Deep Learning tras introducir los algoritmos de retro propagación o *backpropagation*. A día de hoy el Deep Learning sirve para que todo tipo de máquinas puedan realizar computación visual, realizar traducciones a los diferentes idiomas.

2.3. TEORIA FUNDAMENTAL DEL APRENDIZAJE PROFUNDO

Previamente antes de realizar cualquier tipo de aprendizaje ya sea Machine Learning o Deep Learning es necesario que tipo de necesidades tenemos para desarrollar nuestro algoritmo. Lo primero de todo sería necesario saber qué características de los datos extraemos.

Sabiendo las características más valiosas de nuestros datos, hay que entender que hay principalmente dos tipos de aprendizaje para la realización de nuestro entrenamiento:

- **Aprendizaje No Supervisado (*Unsupervised Learning*):** Para este tipo de aprendizaje deseamos conocer la estructura inherente de nuestros datos sin utilizar ningún tipo de etiquetas proporcionadas de forma explícita donde la máquina solo tiene los datos y a partir de ciertas técnicas sacar sus propios resultados.
- **Aprendizaje Supervisado (*Supervised Learning*):** Para este tipo de aprendizaje es necesario el uso de etiquetas (*labels* o *ground truth*) donde a la máquina se le proporciona un set de datos etiquetados para que sepa sacar conclusiones y clasificar a partir de dichas etiquetas.

En nuestra aplicación usaremos un aprendizaje supervisado debido a que tenemos una máscara donde cada pixel es una etiqueta.

El principal objetivo de este tipo de aprendizaje es que sea capaz de clasificar de una forma correcta además de poder realizar otro tipo de tareas como podría ser la detección en una foto de los diferentes tipos de animales que hay en ella.

La aplicación más sencilla para empezar es un **clasificador binario lineal**, el clasificador se encarga de distinguir entre dos categorías por eso el nombre de binario y como utiliza funciones lineales para las entradas de ahí la parte de lineal.

La manera en la que el clasificador lineal binario trabaja es muy simple, se calcula una función lineal de las entradas y se determina si el valor es mayor o menor que un umbral. En la siguiente ecuación 1 podemos ver la función lineal:

$$w_1 x_1 + w_2 x_2 + \cdots + w_D x_D + b = w^T x + b \qquad (1)$$

Si generalizamos a partir de dicha ecuación obtenemos la siguiente función lineal, muy fácil de entender:

$$z = w^T x + b \tag{2}$$

En esta función lineal nuestra variable W es un vector de **pesos** y b es un valor escalar **sesgo** o **bias**. Ambas variables son entrenadas para conseguir una predicción en nuestro caso Z y la última variable será nuestros datos proporcionados a la entrada X (Grosse, 2018).

Tras realizar nuestra función obtendremos una salida Z que será un vector de escalares denominados **logits**, donde cada elemento de dicho vector corresponderá a una predicción Y que podrá ser clasificada a partir del umbral r de la siguiente forma.

$$y = \begin{cases} 1 \; si \; z \geq r \\ 0 \; si \; z < r \end{cases} \tag{3}$$

Este clasificador tiene una versión para clasificar dos clases o más, vamos a ver ahora el funcionamiento del **clasificador lineal**. Lo primero sería utilizar etiquetas contiguas, si vamos a trabajar con N clases, nuestros valores para las etiquetas serían desde 0 hasta N-1. Hay otro método para poder representarlas que puede llegar a ser mucho más conveniente en algunos escenarios, esta técnica se conoce como ***one-hot encoding*** en las siguientes ecuaciones vamos a ver como se produce el cambio de valores enteros a binarios y el cambio de dimensión, (Grosse, 2018).

$$t = [2,0,3,1] \; donde \; N = 4 \tag{4}$$

Procedemos al cambio de dichos valores enteros a binarios:

$$\begin{cases} 0 = [0,0] \\ 1 = [0,1] \\ 2 = [1,0] \\ 3 = [1,1] \end{cases} \tag{5}$$

Y finalmente las etiquetas serán de la siguiente forma:

$$t' = [[1,0], [0,0], [1,1], [0,1]] \tag{6}$$

El principal inconveniente de esta conversión es que la dimensión del vector de etiquetas cambia a ser bidimensional o matricial.

En nuestra aplicación al tener una imagen que se conoce como ***ground truth*** o máscara donde cada pixel es un entero etiquetado, si realizamos el ***one hot encoding*** pasamos de tener una matriz a un cubo compuesto por tantas matrices como N clases tengamos.

Para poder seguir explicando funciones y teoría que luego usaremos en nuestra aplicación es necesario que continúe con esta modalidad de etiquetas.

Supongamos que hemos obtenido los resultados Y que hemos visto en la salida de la segunda ecuación, dichos valores se tendrán que convertir a unas probabilidades que estarán entre 0 y 1 para ello usaremos la una función 7 llamada ***softmax***, la cual es:

$$y_k = softmax(z_1, \dots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} \tag{7}$$

La entrada a esta función serán los ***logits*** producidos por nuestro clasificador lineal y las salidas serán valores no negativos que sumarán 1, así que se podrá interpretar como una distribución de probabilidades sobre un número de K clases. Entonces en dicho vector de probabilidades a la salida se tomara como predicción el valor de probabilidad más alto y se comparará con el vector de etiquetas y el resultado será una salida que se podrá parecer a los datos de entrada, si está comparativa no es acertada nuestra tarea llevada a cabo será nefasta (Grosse, 2018).

Para poder comparar estos resultados, necesitamos otra función llamada entropía cruzada la cual se usa comúnmente para cuantificar la diferencia entre dos distribuciones de probabilidad, ecuación 8.

$$L_{Cross-entropy}(y, t) = -\sum_{k=1}^{K} t_k \, log \, y_k = -t^T (log \, y) \tag{8}$$

Donde los elementos t_k corresponden a la distribución de probabilidades verdadera es decir nuestras etiquetas y los elementos y_k corresponden a la distribución de probabilidades conseguida en la función previa ***softmax***.

Si ponemos todo estas cosas de forma conjunta y ordenada obtenemos la **regresión logística multiclase** donde cada función actúa de forma conjunta con la anterior y la siguiente donde en algunos casos se combina todo en una única función llamada ***softmax-cross-entropy*** (Grosse, 2018)**.**

En nuestra aplicación en vez usar ***softmax-cross-entropy*** utilizaremos la función ***sparse-softmax-cross-entropy-with-logits*.** La función se debe usar para la clasificación multinomial mutuamente excluyente.

La principal diferencia que hay con la anterior versión está en la codificación de las etiquetas, las etiquetas se especifican como números enteros, no con una codificación ***one-hot*** esto hace que los consumos de memoria sean menores (Murat, 2018).

En ambos casos lo que perseguimos es que la pérdida o ***loss*** en las predicciones sea mínima, para ello utilizaremos un algoritmo de **optimización.** Ya que usamos la entropía cruzada para conseguir la medida de similitud entre la distribución verdadera y la predicha, dicha distancia entre valores será el proceso de **optimización** ya que la distancia más corta marcará la elección a la clase correcta por contra la distancia más larga corresponderá a la clase incorrecta. La función típica para usar en una regresión lineal, ecuación 9 se realiza de la siguiente manera:

$$J(w,b) = \frac{1}{N}\sum_{n=1}^{N} L_{Cross-entropy}(y,t) \tag{9}$$

Está ecuación calcula la media de todas las entropías cruzadas de todo el ejercicio o entrenamiento. Así que es necesario encontrar los pesos y sesgos adecuados para minimizar dicho error, (Murat, 2018).

El método más sencillo de explicar para conseguir los valores necesarios de los pesos para que minimicen el error es el **Descenso de Gradiente** o ***Gradient Descent*****.**

$$b^{(k+1)} = b^{(k)} - \alpha \frac{\partial}{\partial b^{(k)}} J(w,b) \tag{10}$$

$$w^{(k+1)} = w^{(k)} - \alpha \frac{\partial}{\partial w^{(k)}} J(w,b) \tag{11}$$

Donde k = ***epoch*** o **iteración actual** y α = ***Learning rate*** o **tasa de aprendizaje**

Profundizando, este método de optimización lo que realiza es una derivada de la iteración actual para cambiar la dirección y el sentido intentando obtener el mínimo, tras la ejecución tanto los sesgos como los pesos de la aplicación cambian.

En la siguiente figura podemos ver el cómo se actualizan dichos pesos.

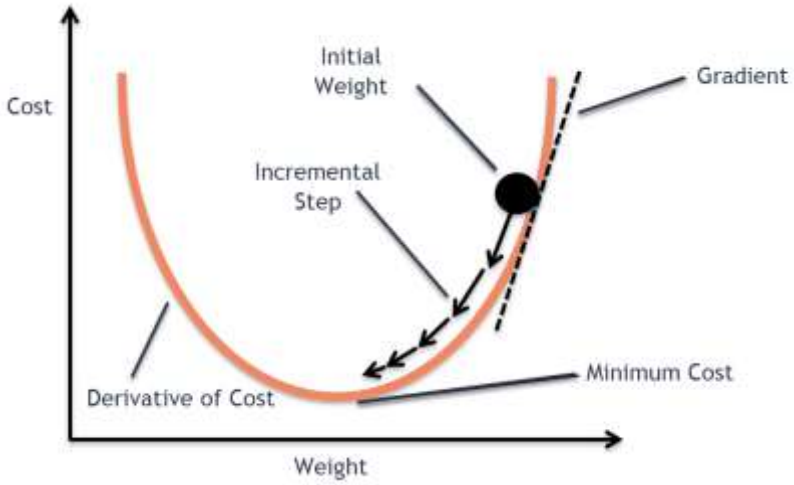

Figura 3 Gradient Descent

Previamente consideraremos la función de pérdidas con un único coeficiente de pesos para poder explicar la Figura 3 donde podemos ver el proceso de optimización mediante ***Gradient Descent***.

Como un montañista que quiere bajar una montaña hacía un valle, la montaña será la función de perdidas, el montañista los pesos y el valle el coste mínimo. Cada paso que realice el montañista por dicha inclinación de la pendiente será nuestro ***Gradient Descent*** y la longitud de la pierna será la tasa de aprendizaje.

Este método de optimización no se utiliza, se utiliza otro método que se basa en la misma idea llamado ***Stochastic Gradient Descent*.** La ventaja que tiene es que al pasar las muestras de entrenamiento a través de la red y los pesos de cada capa se actualizan con el gradiente calculado. Los parámetros de la red se irán actualizando cada iteración ya que en el ***Gradient Descent*** no ocurre de esta manera, no existe concepto de ***epoch*** o de ***batch***.

Además en sus principales ventajas es la convergencia en los *datasets* más rápido que el GD debido a que permite el escalado de los parámetros de la red, siendo más eficiente que su predecesor.

A pesar de ello, el algoritmo utilizado en nuestra aplicación será ***Adam Optimizer***, su nombre completo es ***Adaptive Moment Estimation***.

Presentado por Diederik Kinga y Jimmy Ba en 2017, ***Adam*** es diferente al clásico SGD ya que mantiene la idea de una única tasa de aprendizaje para la actualización de los pesos en vez de usar diferentes tasas de aprendizaje para cada uno de los parámetros de la red (Kingma & Ba, 2015).

Además combina el uso de dos extensiones del SGD:

- ***Adaptive Gradient Algortih (AdaGrad):*** el algoritmo AdaGrad (Duchi, J. et al., 2011, citado en Kingma & Ba, 2015) mantiene una tasa de aprendizaje para cada parámetro esto hace que mejore el rendimiento en ciertos campos con gradientes dispersos como pueden ser la visión artificial y el procesamiento de lenguajes naturales (Kingma & Ba, 2015).
- ***Root Mean Square Propagation(RMSProp)****:* el algoritmo RMSProp (Tieleman & Hinton,2012, citado en Kingma & Ba, 2015) mantiene una tasa de aprendizaje de cada parámetro como en el caso anterior pero se adapta gracias al promedio de las magnitudes de los gradientes para los pesos. Esto significa que vale para resolver problemas estacionarios no lineales y lineales como puede ser el caso del ruido (Kingma & Ba, 2015).

Así que Adam en vez de adaptar los parámetros de la tasa de aprendizaje en función del primer momento medio como ocurre en el caso de ***Root Mean Square Propagation (RMSProp)*** también utiliza la media de los segundos momentos de los gradientes (Kingma & Ba, 2015).

El algoritmo calcula una media exponencial móvil del gradiente y del gradiente cuadrado, donde los parámetros β_1y β_2 controlan las tasas con las que decrecen dichas medias. Es por eso que los valores recomendados sean cercanos a 1.0 ya que pueden dar como resultado un buen sesgo de estimaciones de momento cero. Por último hay un valor ε el cual suele ser un número muy pequeño 10^{-8} ya que se utiliza para prevenir cualquier tipo de división por cero en las implementaciones (Kingma & Ba, 2015).

En el documento de los autores hay una figura que compara los diferentes costes de un entrenamiento en una red neuronal multicapa con ***dropout*** usando la base de datos MNIST la cual es una de las principales bases de datos compuestas por diferentes

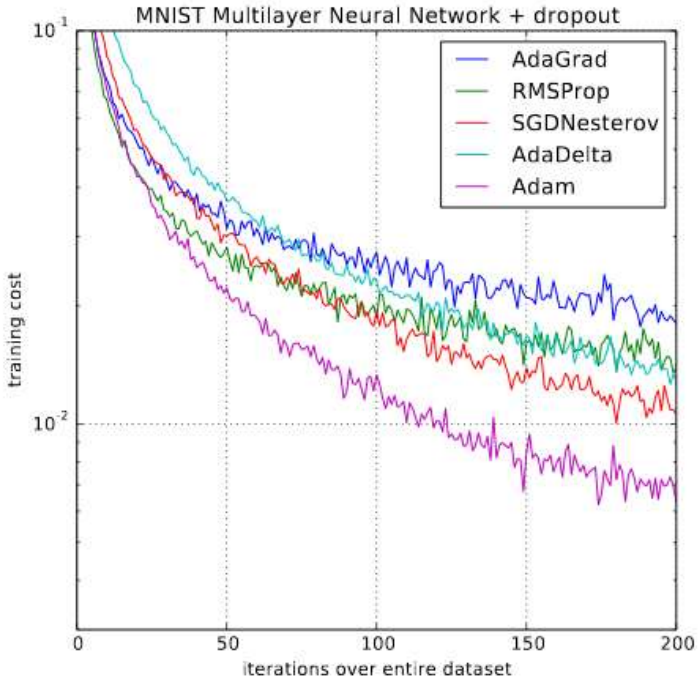

Figura 4: Entrenamiento de una red neuronal multicapa sobre las imágenes MNIST usando dropout, donde se puede observar la comparativa de Adam con el resto de optimizadores

números y donde hay un gran número de elementos, esta base de datos se utiliza como tutorial para cualquier persona que quiera adentrarse en el mundo del *Deep Learning*.

En la Figura 4, de (Kingma & Ba, 2015), se muestra como Adam es capaz de conseguir el coste más bajo durante el entrenamiento. Es por eso que implementaremos en este Trabajo Final de Grado la utilización del optimizador ***Adam*.**

A lo largo de las pruebas realizadas en nuestra aplicación, usaremos los siguientes valores para las constantes siguiendo en cierta medida las recomendaciones dadas por los autores en el documento:

- La tasa de aprendizaje o ***Learning rate*** será 0,0001 aun así será el valor que con el que más jugaremos en las diferentes pruebas realizadas.
- La variable β_1 será 0.9.
- La variable β_2 será 0.999.
- La variable ε será 10^{-8}.

Después de dicha operación con el optimizador ***Adam*** se procede a realizar la minimización de las pérdidas, se utiliza la función ***minimize*** que se encarga de calcular dichos gradientes y aplicarlos a lo pesos y sesgos mediante la propagación hacía atrás.

BLOQUE 3:

REDES NEURONALES CONVOLUCIONALES

3. REDES NEURONALES CONVOLUCIONALES

3.1. DESCRIPCIÓN DETALLADA

A lo largo de este bloque, veremos cómo plantear el problema que se nos pide de nuestra aplicación mediante la teoría correspondiente a las Redes Neuronales Convolucionales más conocidas como **CNN** y las múltiples y diversas formas de tratar nuestras imágenes para acércanos en lo mejor de lo posible a lo que se nos requiere.

Como nuestro problema a abordar es de segmentación de imágenes vamos a continuar con los conceptos básicos para ver los diferentes tipos de segmentación que hay y procederemos con la parte teórica de las capas que componen una red de este tipo.

Por último veremos qué tipos de Redes Neuronales Convolucionales hay y explicaremos como es la red que usaremos con una aproximación teórica.

3.2. CLASIFICACIÓN, LOCALIZACIÓN Y SEGMENTACIÓN

En el mundo del *Deep Learning* hay múltiples aplicaciones que nos pueden facilitar el día a día, algunas de ellas pueden utilizar redes para la traducción entre diferentes idiomas y buscar coincidencias las cuales no se pueden traducir de forma directa, por ejemplo uno de los traductores que utiliza inteligencia artificial es DeepL Figura 5 de (DeepL, 2017).

Figura 5 Traductor desarrollado por Linguee mediante Redes Neuronales Convolucionales

Otra aplicación podría ser la que mostro Nvidia en una conferencia de prensa del 2017 donde conseguían componer una pieza musical a partir del aprendizaje de diferentes instrumentos utilizados por una orquesta y el aprendizaje mediante la actuación de diferentes piezas musicales, que permiten a dicha aplicación el poder generar de forma autónoma nuevas piezas musicales creadas por un ordenador (GTC 2017: I Am AI Opening in Keynote, Nvidia).

Centrándonos más en el título de esta sección tenemos una aplicación bastante cuestionada por diferentes países, esta es el uso del reconocimiento facial por parte del gobierno chino donde se consigue saber la identidad de una persona a partir de múltiples fotos tomadas del individuo en cuestión en cualquier punto del país.

La aplicación que tenemos que desarrollar en este Trabajo Final de Grado es la segmentación de imágenes biomédica. Antes de nada veremos que diferentes técnicas hay para el uso de las imágenes para los distintos campos.

3.2.1. CLASIFICACIÓN Y LOCALIZACIÓN

La clasificación y localización de las imágenes consiste en asignar una etiqueta de una categoría a las imágenes analizadas, pero además de poder predecir la categoría, también nos puede interesar la ubicación del objeto en la imagen.

En términos matemáticos se puede conseguir dibujar un cuadrado alrededor delimitando dicho objeto en la imagen como podemos ver en la siguiente Figura 6, de (Parmar, 2018).

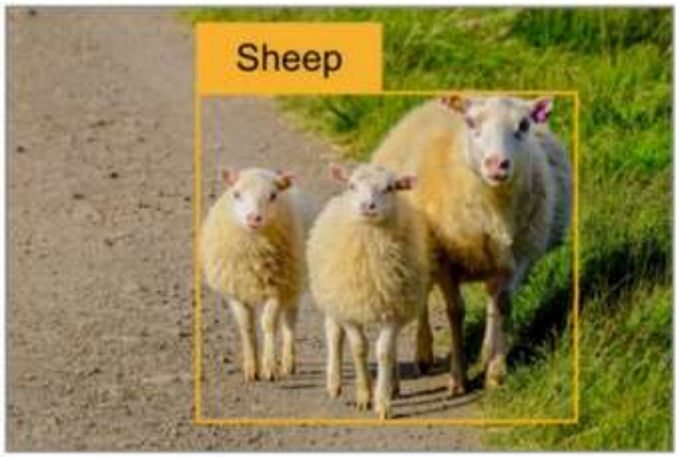

Figura 6: Ejemplo de clasificación y localización.

Es necesario alimentar la entrada con las imágenes que deseamos clasificar y utilizar redes Convolucionales, lo más normal como puede ser el caso de la base de datos MNIST donde solo podemos clasificar y en el caso de CIFAR-10 donde realizamos la clasificación y además localizamos.

La red más utilizada es con capas Convolucionales y capas que realizan el ***maxpooling*** y finalmente conectada a una capa ***fully connect*** donde la salida es el número de clases. Además podemos utilizar otra capa para localizar el centro del objeto con coordenadas x e y, y dibujar la caja que perfila el objeto destacado. Por el contrario tenemos dos funciones de coste una para las clases y otra para los encuadres. El esquema básico sería de la siguiente forma (Parmar, 2018).

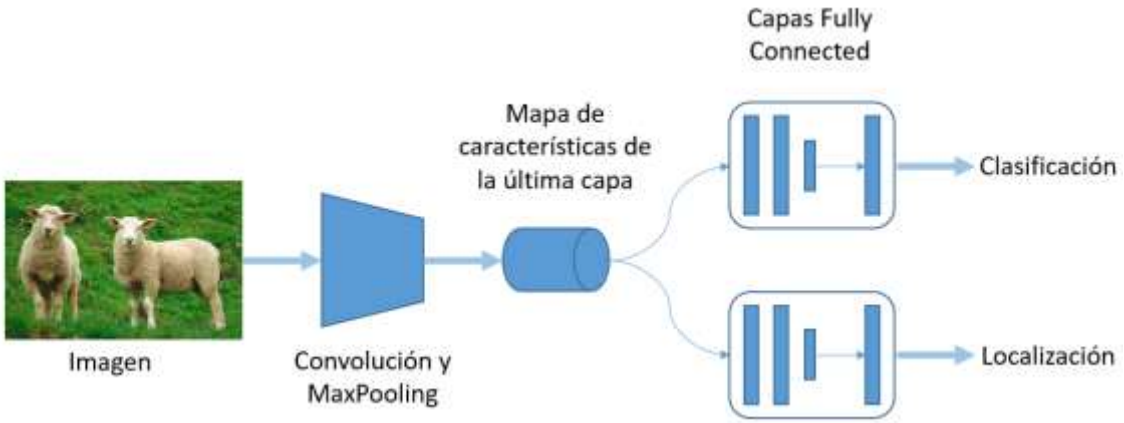

Figura 7: Esquema de la técnica de Clasificación y Localización

3.2.2. DETECCIÓN

Esta técnica es una forma de la aplicación de la clasificación y la localización de las imágenes pero no se aplica de la misma forma, aquí la idea radica en que la detección de objetos empieza por tener un set de datos etiquetados con diferentes clases.

El problema a tratar es la detección de un número variable de objetos en la misma fotografía no solo localizamos dicho objeto sino que además se le consigue etiquetar como podemos ver en la Figura 8. Esto se consigue mediante la repetición de la misma foto para que vaya pintando y prediciendo los diferentes objetos (Parmar, 2018).

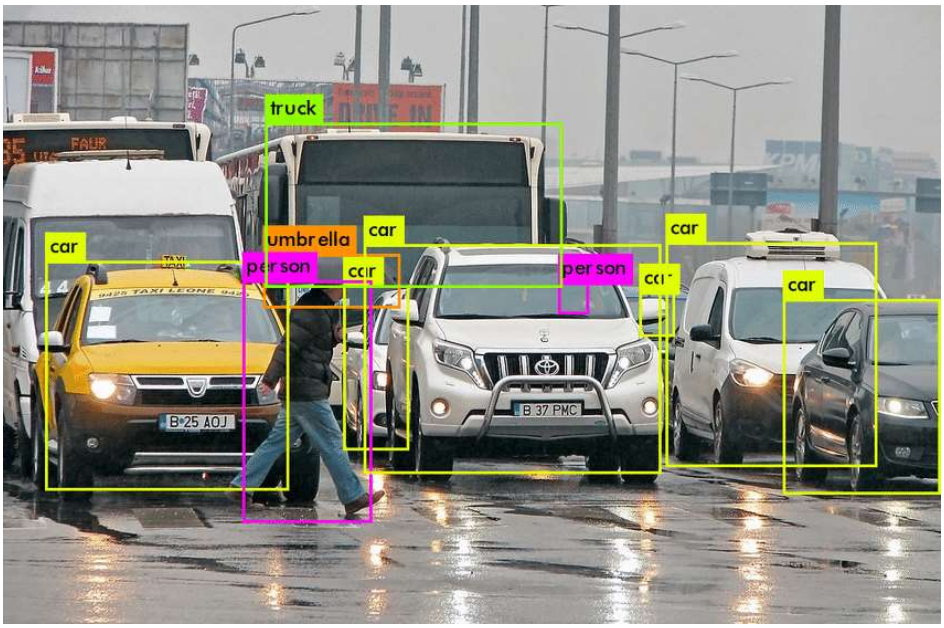

Figura 8: Diferentes objetos de clases distintas localizados mediante la detección en la foto

Otra diferencia principal con la simple clasificación de objetos, es que la salida de la aplicación en el caso de la detección de objetos es variable, cada foto tiene su número de objetos detectados, no siempre va a ser el mismo entre dos fotos de la base de datos.

Los principales usos a día de hoy en la detección de objetos varían mucho a lo largo de los distintos ámbitos, por ejemplo Tesla la marca de coches eléctricos tiene un software incluido en el coche que es capaz de hacer detección de objetos lo que facilita el autopilot, permitiendo que el coche gire, acelere y adelante de forma autónoma dentro del carril (Tesla Piloto Automático, 2020). Otra empresa que ha desarrollado una herramienta de detección de objetos ha sido Adobe Photoshop la cual es un selector de objetos integrados en dicho programa que permite marcar objetos de forma automática e instantánea, siendo de gran ayuda para editores y fotógrafos.

3.2.3. SEGMENTACIÓN

A parte de los tres problemas anteriores en el campo de la visión por ordenador hay que añadir la segmentación semántica, la cual se encarga de analizar cada pixel de forma individual y tomar una decisión categórica, es decir clasificaríamos cada pixel de la imagen en un abanico de posibles categorías (Parmar, 2018).

Una forma de aproximar el problema en cuestión es el uso de una ventana que se va moviendo a lo largo de la foto o conocido como ***sliding window***, es una región rectangular marcada con un ancho y un largo que va cortando la imagen en pequeñas imágenes del mismo tamaño y se analizan cada una de ellas. Cada parte o ***crop*** es la forma con la que se alimentan ciertas redes Convolucionales donde la salida de ellas proporciona la clasificación del pixel por comparación de las categorías, aun así es una técnica resolutiva pero no eficiente, más adelante veremos el uso de cierto tipos de red que llegan a ser más eficientes que realizar la técnica de ventana (Parmar, 2018).

A mayores disponemos de la segmentación de instancias, esta técnica emplea la segmentación semántica y a mayores la técnica de detección de objetos, no solo nos interesa marcar cada pixel que hay en la imagen sino también tratar de entender el que hay en una imagen.

Como podemos observar en la Figura 9 de (Agarwal, 2019) las diferentes posibilidades para abordar nuestro problema se resumen a las cuatro formas de tratar la visión por ordenador, donde podemos llegar a la conclusión de que el problema que va a tratar nuestra aplicación es de segmentación semántica, esto se debe a que vamos a analizar pixeles y no vamos a buscar objetos en la imagen, sino clasificar cada pixel de la imagen para poder predecir una máscara con los resultados obtenidos del modelo de red que podamos llegar a usar.

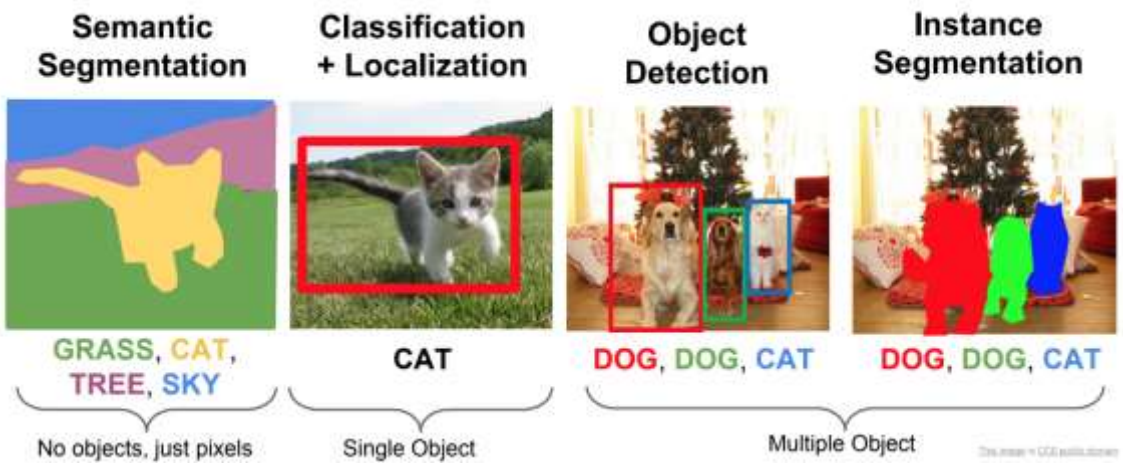

Figura 9: Ejemplo de las distintas técnicas de Deep Learning aplicadas a imágenes o fotografías

3.3. REDES NEURONALES

Tras ver cómo podemos abordar nuestro problema en cuestión vamos a abandonar los modelos lineales y nos vamos a centrar en las redes neuronales, explicaremos en qué consisten estas y procederemos a ver cuáles son las características de las redes que se utilizan para tratar imágenes.

3.3.1. REDES NEURONALES PROFUNDAS (CNN)

Vamos a introducir nuestro nodo de procesamiento que va a ser similar a una neurona:

$$a = \emptyset\left(\sum_j w_j x_j + b\right) \tag{12}$$

Donde x_j son las entradas al nodo, w_j son los pesos y por ultimo b es el bias, $\emptyset$ es la función de activación no lineal y por tanto a es la activación de los nodos.

Así que una red neuronal es simplemente la combinación de una gran cantidad de estas unidades conocidas como nodos o neuronas. Cada una de ellas realiza una simple y acotada función. Las redes más simples son las llamadas ***multilayer perceptron (MLP)***, como podemos observar en la Figura 10 de (Grosse, 2018).

Los nodo están unidos en un conjunto de capas, y cada capa contiene un número idéntico de nodo. Todo nodo en una capa está conectada a todo nodo de la siguiente capa mediante conexiones que tienen distintos **pesos**, así que podemos decir que la red es ***Fully Connected*** o completamente conectada.

Siempre la primera capa es la capa de entrada por la que los nodos toman los valores de las características de entrada. La última capa normalmente la salida suele tener

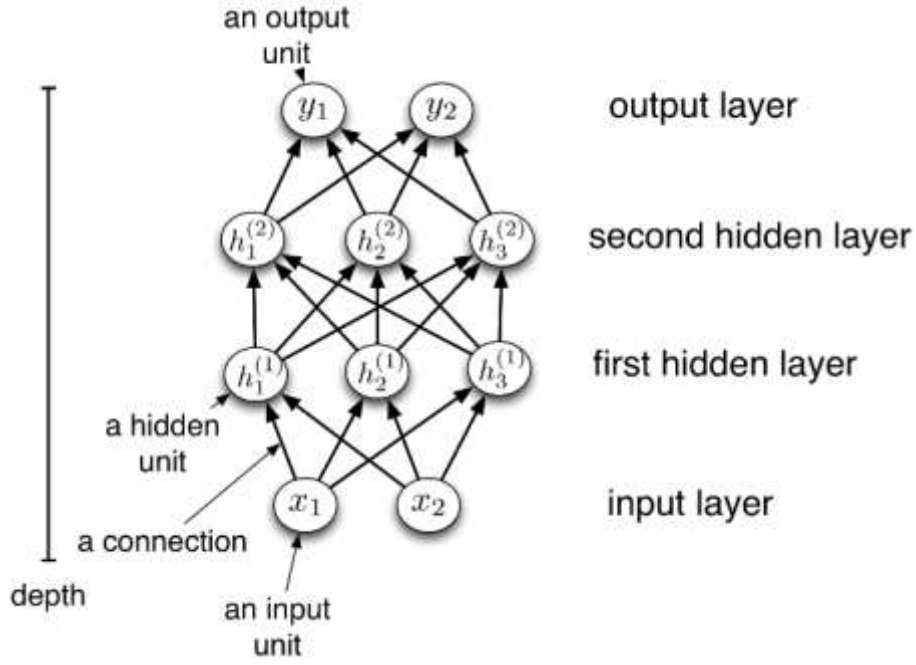

Figura 10: Red multicapa Perceptron con dos capas escondidas

un nodo para cada valor de salida de la red, por ejemplo en un clasificador binario, habría 2 nodos, en caso de ser multiclase habría K salidas.

Entre la capa de entrada y la capa de salida están las capas ocultas o escondidas, se llaman así debido a que no se sabe de antemano que es lo que las neuronas tienen que calcular y es lo que se descubren durante el aprendizaje (Grosse, 2018).

Las redes tienen un número de capas el cual se conoce como **profundidad** de la red y el número de los nodos en una capa se conoce como la **anchura** de la capa (Goodfellow,I. et al., 2016).

Los correspondientes nodos de la red consiguen transformar los datos o las características de la capa de entrada mediante operaciones no lineales para crear ese umbral de decisión de la entrada de la red.

Entonces la salida de la red debe de ser parecida a los datos que hemos introducido, es decir se espera que los valores de la capa de salida de la red deben ser parecidos o cercanos a los valores de la etiqueta determinados en los datos de la entrada.

Como podemos observar solamente podemos saber cómo se obtienen los resultados a partir del propio algoritmo de aprendizaje y no de los ejemplos de entrenamiento que son introducidos al inicio de la red, además como no se muestran los resultados intermedios de los mapas de características de las capas intermedias es por eso que se llaman ***hidden*.**

Respecto a por qué se llaman redes neuronales, es debido a que las neuronas que componen las redes tienen sus entradas, sus pesos y sus salidas un equivalente en la vida humana donde las dendritas serían las entradas a la neurona, el soma o la sinapsis serían los pesos y por último el axón seria la equivalencia a la salida natural de la neurona, como podemos ver en la Figura 11.

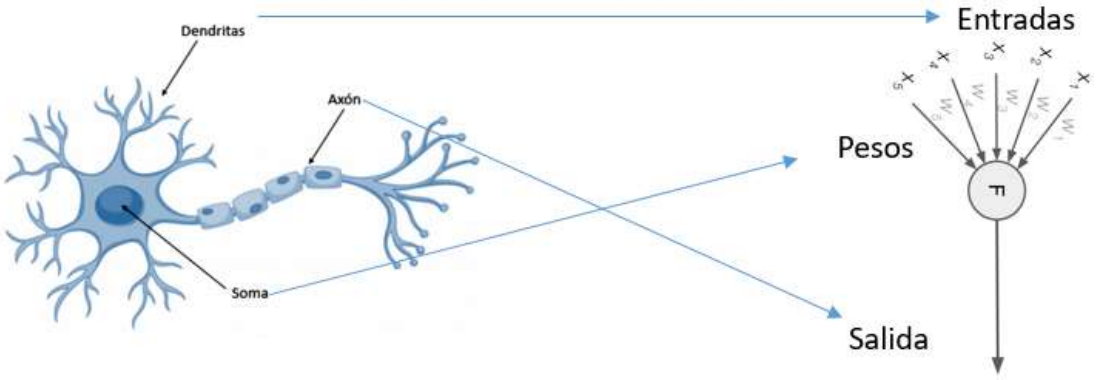

Figura 11 Comparación entre una neurona humana y una neurona artificial

Es por eso que las redes neuronales son una combinación de varios nodos y se llaman así por la similitud con las de nuestro cuerpo humano.

La neurona trabaja con una gran cantidad de valores de entrada como si fuera uno solo es decir un vector de datos, para la entrada a la neurona se calcula la **función de entrada** a partir del vector de entrada, normalmente se multiplican dichos valores por los pesos asignados a la neurona (asignados de forma aleatoria la primera vez) pero también hay otro tipo de funciones lineales que se pueden llegar a utilizar como la suma de las entradas con los pesos, la multiplicación de las entradas con los pesos o el máximo de las entradas con los pesos (Goodfellow,I. et al., 2016).

Independientemente de la función lineal que usemos, a dicho resultado se le aplicará una **función de activación** en este caso dicha función no es lineal. Esta función de activación al igual que en el ser humano la unidad o neurona puede estar activa (excitada) o inactiva (no excitada), es por ello que las neuronas "artificiales" tienen diferentes estados de activación algunas simplemente pueden tener dos estados de activación pero otras pueden tener varios estados de activación. A priori es difícil saber cuál de los diferentes tipos de funciones de activación va a ser la recomendada para trabajar con ella y obtener los mejores resultados, simplemente vamos a comentar las que utilizaremos en nuestra aplicación para la diferentes pruebas.

- **ReLU (*Rectified Linear Unit*):** La función unidad lineal rectificada ha llegado a ser muy popular en los últimos ya que su función 13 es la siguiente:

$$f(x) = max\,(0, x) \tag{13}$$

La activación es un simple umbral ubicado en cero como podemos observar en la figura 12. Debido a una de sus principales características se encuentra a que encuentra fácilmente la convergencia del gradiente descendiente estocástico lo que significa una mayar velocidad. Comparando con otras posibles funciones de activación, la función ReLU no involucra unas operaciones complejas como pueden ser las exponenciales ya que ReLU puede ser implementada por un simple umbral de una matriz de activaciones con valor cero. Esta función de activación será la que principalmente usaremos a lo largo de las pruebas realizadas (Goodfellow,I. et al., 2016).

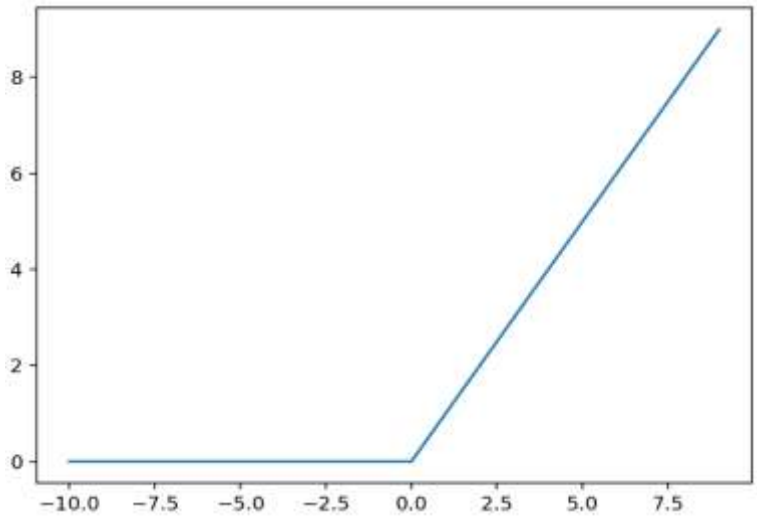

Figura 12 Función de activación ReLU

Además en un modelo lineal como hemos podido ver en el bloque anterior sería la siguiente ecuación 14:

$$z = f(w^T x + b) \tag{14}$$

- ***Sigmoid Function***: La función sigmoide o ***logistic sigmoid*** cuya función de activación es la siguiente:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{15}$$

Está función tomará un valor real y lo compara en un rango entre 0 y 1. Los valores negativos mayores se convierten en 0 y los valores positivos mayores se convierten en 1, el principal problema que tiene es para los valores intermedios o cercanos a cero puede ser más sensible, además esto hace que los algoritmos de aprendizaje donde se puedan llegar a usar gradientes sea más complicado que con el uso de la función ReLU (Goodfellow,I. et al., 2016).

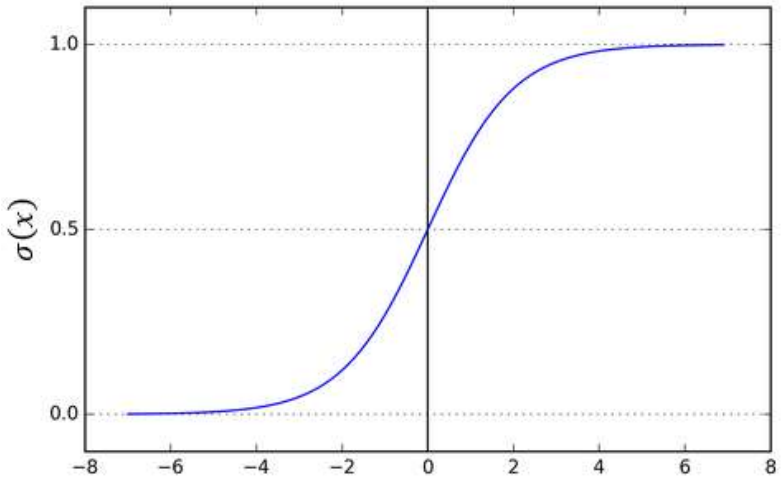

Figura 13 Función de activación Sigmoid

- ***TanH Function*:** Función tangente hiperbólica cuya función de activación es la siguiente:

$$g(z) = tanh(z) \tag{16}$$

$$tanh(z) = 2\sigma(2z) - 1 \tag{17}$$

Esta función en cierta medida está relacionada con la función anterior, la función sigmoide, ya que en algunos casos puede llegar a rendir mejor que la función sigmoide, esto es porque se parece demasiado a la función identidad ya que su derivada en cero es $\frac{1}{2}$ asemejando en algunos casos el entrenamiento de un modelo de red neuronal al del entrenamiento de un modelo lineal de dos clases haciéndolo mucho más sencillo (Goodfellow,I. et al., 2016).

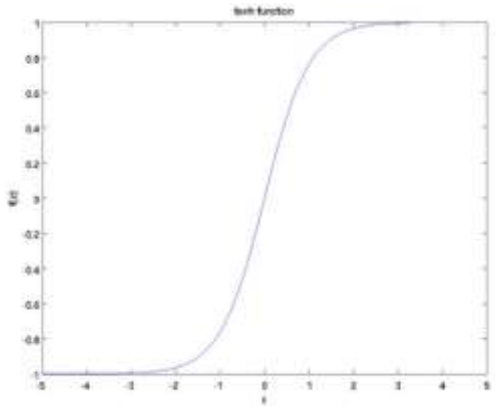

Figura 14 Función de activación tanh

Como podemos ver en el ejemplo de una red ***feedforward*** muy sencilla la cual puede ser vista en dos formas diferentes, esta tiene dos unidades en una **capa oculta.** En la Figura 15 (a) de (Goodfellow,I. et al., 2016) cada unidad tiene un nodo en el grafo, esta forma es muy explícita y para el caso en el que tengamos capas con demasiadas unidades puede llegar a tener un problema de espacio, en cambio en la otra forma se puede observar en la Figura 15 (b) de (Goodfellow,I. et al., 2016) como tenemos un único nodo en el grafo para las dos unidades que contienen las capas.

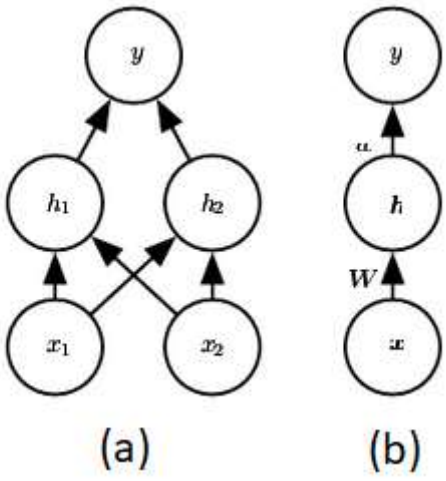

Figura 15 Red neuronal Feedforward, modelo completo (a) y modelo vectorial (b)

Por otro lado las redes neuronales realizan la propagación hacía atrás o ***back-propagation*****,** esta propagación hacia atrás es hacía las capas anteriores donde a partir de los resultados y los costes, cada unidad es capaz de ajustar los parámetros idóneos para obtener mejores resultados ya que esto permite la minimización del error o del coste, de esta forma la red se retroalimenta variando los valores de sus parámetros internos hasta llegar a conseguir los mejores resultados para dicha red.

3.3.2. REGULARIZACIÓN PARA DEEP LEARNING

Uno de los principales problemas en Machine o Deep Learning es como hacer que el algoritmo que puede funcionar bien con los datos de entrenamiento, pueda mejorar de una forma óptima el error "general" de nuestra aplicación.

La mayor parte de las estrategias de regularización están basadas en estimadores (Tensorflow, 2020) en nuestro caso sí que haremos uso de ellos para la distribución de los datos de entrenamiento a lo largo del bucle del entrenamiento, lo que nos permite:

- Cargar los datos de una forma controlada y aleatoria.
- Manejo de excepciones
- Poder crear varios *checkpoints* a lo largo del proceso para en caso de caída de la aplicación poder cargar desde el último punto o el mejor punto de la red

Además de usar los estimadores, utilizaremos otra técnica de regularización conocida como ***Early Stopping*** incluida en la técnica conocida como ***Reduce Learning Rate on Plateau (RoP)*** esto evita en la mayor parte el ***Overfitting*** de la red, y el uso de capas de **Dropout.**

- ***Early Stopping*****:** Cuando entrenamos modelos con mucha profundidad o muchas capas normalmente suelen tender al *overfitting*, aun así observamos que el error de entrenamiento disminuye constantemente con el tiempo pero en la parte que se encarga de la validación podemos observar que tiende a crecer o de forma abrupta o con una pendiente muy muy pequeña, como podemos observar en las Figuras 16 (a) y (b).

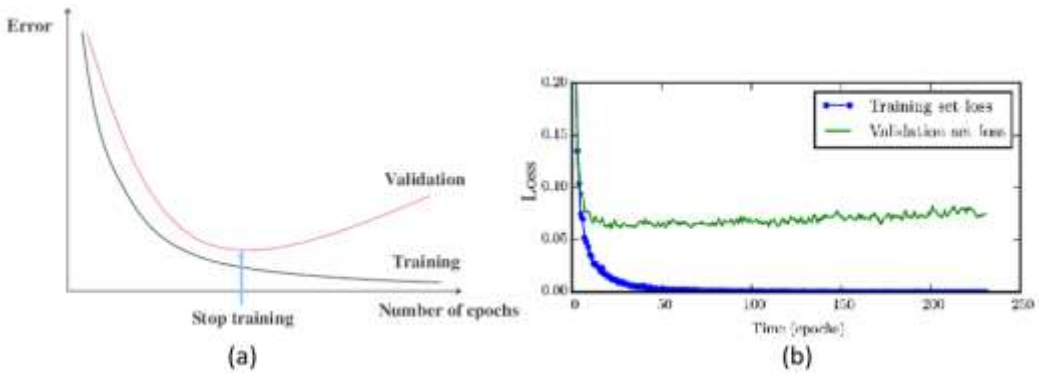

Figura 16 Overfitting con una pendiente abrupta (a), Overfitting con una lígera pendiente casi constante (b)

Esto implica que el mejor modelo con el mejor valor de la función de pérdida para el set de validación, es aquel donde hemos obtenido el punto más bajo para el error dentro del set de validación, para ello podemos realizar un *checkpoint* de los parámetros de nuestra red en ese momento y poder

retomar siempre la red a ese punto para probar con un set diferente de datos, o cambiar el algoritmo para minimizar dicho error, o simplemente cambiar alguno de los parámetros prefijados antes de comenzar el entrenamiento (Goodfellow,I. et al., 2016).

Esta estrategia conocida como ***Early Stopping*** es probablemente la más usada para la regularización en Deep Learning además de ser una de las más populares.

Para implementar la estrategia es necesario saber cuándo parar después de obtener el mejor resultado durante la validación, ya que se puede dar el caso en el que el tiempo sea una de nuestras principales preocupaciones, no por poner un número grande de *epochs* vamos a conseguir mejores resultados, por ello vamos a implementar el método de optimización del ***Early Stopping*** mediante el desarrollo de un algoritmo que se ejecuta durante la validación llamado ***Reduce Learning Rate on Plateau*****.**

Al igual que con la anterior técnica de regularización, esta técnica o algoritmo es bastante intuitivo y popular, manteniendo durante las iteraciones un *Learning Rate* alto cambiamos y reducimos cuando llegamos a un punto llamado ***plateau*** es decir una zona plana debido a que las métricas no mejoran e incluso empeoran y por eso reducimos su valor, partimos de una variable llamada **paciencia** donde si durante la validación no ha habido mejora se aumenta dicho contador en 1, así hasta que la paciencia se pierde y da lugar al cambio del ***Learning Rate***, además una vez realizado dicho cambio se puede llegar al ***early-stop*** poniendo un valor pequeño para el ***Learning Rate*** o tener otro contador de **paciencia** global donde si se supera dicho contador se para la ejecución del programa. Esta segunda forma es como esta implementado en este trabajo final de grado.

- ***Dropout*****:** El ***dropout*** es una de las técnicas más efectivas que se utilizan en la regularización de las redes neuronales, el ***dropout*** se aplica a la capa o capas de nuestra red, la cual consiste en poner a cero o apagar un cierto número de características de las características de salida de la capa durante el entrenamiento. El ***dropout*** tiene un valor configurable el cual es la fracción total entre 0 y 1 que queremos poner a cero, normalmente se suelen poner entre 0,2 y 0,5 pero en ciertos casos es admisible desde 0,1 hasta 0,8.

  Las unidades o neuronas que se apagan durante las interacciones del entrenamiento son elegidas de forma aleatoria teniendo en cuenta el número total de unidades. Esto da lugar a que tengamos el resultado de una aproximación de varias redes independientes (supongamos que cada uno de los resultados es visto como una red independiente) lo que hace que sea un aprendizaje redundante (Chollet, 2017).

3.4. REDES NEURONALES CONVOLUCIONALES (CNN)

Redes Convolucionales o Redes Neuronales Convolucionales son un tipo de redes especializadas para datos procesados que es conocida por su topología en malla. El nombre proviene de la operación matemática llamada **Convolución** una operación que es lineal y se realiza en al menos alguna de sus capas que componen la red (Goodfellow,I. et al., 2016).

Antes de meternos como es la estructura de una red convolucional explicaremos que es una Convolución.

En la forma más general una Convolución es una operación sobre dos funciones de un argumento real. En la terminología del Deep Learning es más una operación que estima basándose en una función de pesos como podemos ver en la siguiente ecuación 17:

$$s(t) = \int x(a)w(t-a)da \tag{17}$$

La operación Convolución se representa típicamente con un asterisco ($*$) y la ecuación 18 cambia a ser de la siguiente forma:

$$s(t) = (x * \mathrm{w})(t) \tag{18}$$

$$s(t) = x(t) * \mathrm{w}(t) \tag{19}$$

En la parte técnica de las redes Convolucionales, el primer argumento de la ecuación (la función x) de la Convolución normalmente se refiere a la entrada o ***input*,** y el segundo argumento de la ecuación (la función w) ***kernel*** en otros casos también llamado ***patch*.** Por último la salida o el resultado de la Convolución se le conocen como el ***feauture map*** o mapa de características.

Normalmente al estar trabajando con datos en el ordenador, las expresiones vistas anteriormente tienen que ser discretizadas, es decir es necesario la conversión a señales muestreadas en el tiempo dando lugar a la siguiente ecuación 20

$$s(t) = (x * \mathrm{w})(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{20}$$

En las aplicaciones de *Deep Learning* o *Machine Learning*, la entrada de los modelos para suelen ser vectores multidimensionales de datos y el ***kernel*** es normalmente un vector multidimensional de parámetros que son adaptados al algoritmo de aprendizaje.

Nos referiremos a estos vectores multidimensionales como **tensores**, ya que cada elemento de la entrada y del kernel deben ser almacenados explícitamente separados, debido a que estas funciones son cero en cualquier parte, excepto en el conjunto de puntos finitos para los valores que estamos almacenando.

Esto da como resultado en la práctica que podemos implementar la suma de números infinitos como una suma sobre un número finito de elementos del vector. Finalmente usaremos normalmente la operación Convolución para más de una dimensión, sino que vamos usar para dos dimensiones.

Siguiendo el ejemplo de las ecuaciones anteriores, supongamos que nuestra **I** es una **imagen** bidimensional tomada para la entrada, así que usaremos un ***kernel*** **K** bidimensional también. La expresión para la Convolución sería la siguiente:

$$S(i,j) = (I * \mathrm{K})(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{21}$$

Como la Convolución es conmutativa, significa que se puede rescribir de la siguiente manera:

$$S(i,j) = (K * \mathrm{I})(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{22}$$

La propiedad conmutativa de la Convolución surge porque hemos intercambiado lo relativo al ***kernel*** por la entrada, pero realizar este cambio no es demasiado importante debido a que no suele ser una propiedad de la implementación de la red neuronal así que la mayor parte de librerías que se pueden usar en machine Learning implementan una función relacionada que se conoce como la **correlación cruzada**, la cual es la misma que la Convolución pero sin cambiar el ***kernel*** dando lugar a la siguiente ecuación 23:

$$S(i,j) = (K * \mathrm{I})(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{23}$$

La convolución discreta se puede ver también como una multiplicación de matrices, pero la matriz tiene varias entradas acotadas para ser iguales a otras entradas, esto se debe a que el ***kernel*** suele ser más pequeño que la imagen de entrada y además la parte en la que se cambia el ***kernel*** no suele ser relevante para las capas de las redes neuronales (Goodfellow,I. et al., 2016).

Como podemos ver en la Figura 17 tenemos una convolución de 3x3, es decir, el ***kernel*** es una matriz de dimensión 3x3, con un ***stride*** de 1. Este elemento llamado ***stride*** se define como el tamaño de paso del ***kernel*** al deslizarse por la imagen de entrada. Un ***stide*** con valor 1 significa que deslizamiento del ***kernel*** a través de la imagen es pixel a pixel. Un ***stride*** con valor 2 significa que el deslizamiento del ***kernel*** es moviendo 2 pixeles cada paso que se realiza durante la reducción de la imagen.

Por otro lado hay otro elemento en dichas capas convolucionales que es el ***padding*** o padeo**,** se encarga de manejar los bordes de la imagen durante las convoluciones, lo que mantendrá la dimensión de salida igual a la dimensión de la imagen de entrada. Este valor puede ser configurado de dos formas tal y como permite la configuración en

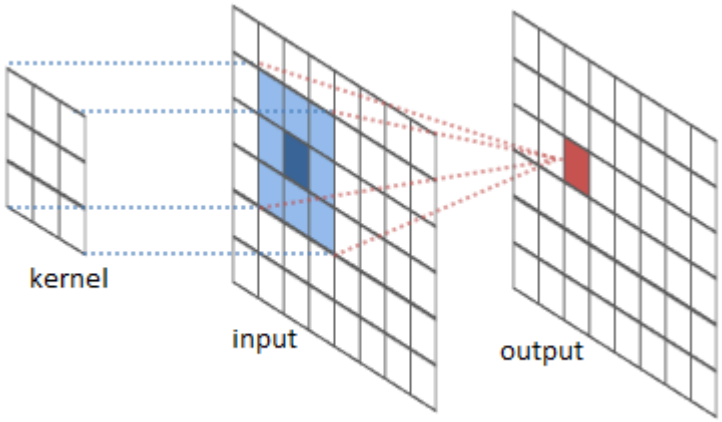

Figura 17 Operación convolución

Tensorflow:

- ***Same*****:** En los límites de la matriz de entrada, supongamos el elemento (1,1) para que la convolución se pueda llevar a cabo se añaden una fila y una columna de ceros a cada lado.
- ***Valid*****:** En este caso no se añadirán los pixeles con valor cero alrededor de los bordes para facilitar la convolución sino que estamos dispuesto a perder cierta información y no tenemos en cuenta dichas filas y columnas, es decir se quedan los datos como estaban en un principio.

3.5. CAPAS DE POOLING

Continuando con la siguiente capa de la red convolucional, introduciremos la capa de ***pooling*****,** este tipo de capas resumen o mejor dicho comprimen el **mapa de**

características obtenido a la salida de la anterior capa, normalmente suele ser una capa convolucional, donde simplemente se realiza el cálculo de una función sobre las pequeñas regiones de la imagen. Lo más común es que la función sea cogiendo el máximo siendo esta operación conocida como el nombre de ***max-pooling***.

Suponiendo que hay un mapa de características, cada unidad de la salida del mapa calcula el máximo sobre alguna región (llamada grupo de pooling) del mapa de entrada normalmente, la región suele ser de 3x3 o 2x2. Para reducir su representación, no se consideran todos los desplazamientos, sino que los espaciamos como en el caso de la capa convolucional por un ***stride*** a lo largo de cada dimensión que tengamos.

Esto da lugar a que la representación se reduce por un factor de aproximadamente S el cual es el ***stride*** a lo largo de cada dimensión, el valor más utilizado para el ***stride*** es 2. De hecho usaremos en las capas de ***max-pooling*** un valor 2 de ***stride*** para nuestra aplicación.

En las opciones del pooling también tenemos la operación ***average-pooling*** la cual en vez de quedarse con el máximo de los datos de la región seleccionada, realiza un media de los datos o valores. Podemos ver las diferencias de ambas operaciones en la Figura 18 de (Grosse, 2018).

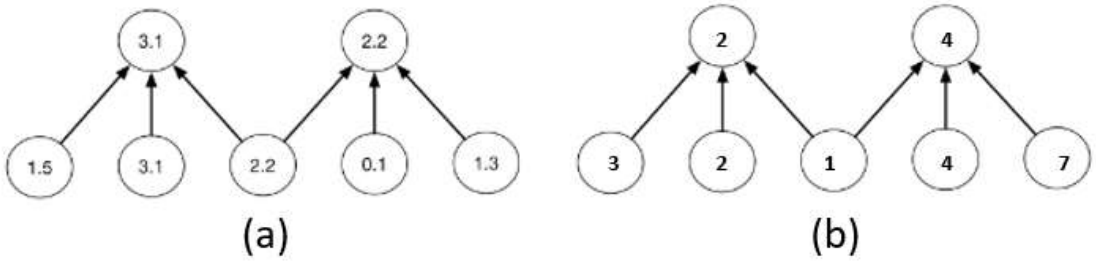

Figura 18 (a) Operacion de Max-Pooling (b) Operación de Average-Pooling

Normalmente cuando realizamos una red convolucional, la capa convolucional y la capa de pooling suelen ir una a continuación de la otra, es por eso, que hay que diseñarlas con los valores acordes, ya que hay que definir la ventana de ***kernel***, la distancia de deslizamiento o ***stride*** y el número de **mapas de características** que habrá a la salida de la capa convolucional, esta salida será la entrada para la capa de pooling donde lo primero de todo es decidir cuál de las dos operaciones realizaremos y como en la capa anterior hay que definir la ventana de ***kernel*** y el ***stride***.

CAPAS DE CONVOLUCIÓN TRANSPUESTA (DECONVOLUCIÓN):

Para diferentes aplicaciones y diferentes redes, solemos utilizar transformaciones que van en el sentido contrario a una capa convolucional, normalmente cuando hacemos el ***up-sampling*** donde partiendo de una salida de un mapa de características con

dimensiones muy pequeñas queremos volver a las dimensiones utilizadas en los datos introducidos para una aplicación de segmentación de imágenes.

La convolución transpuesta o mejor conocida como deconvolución es una operación que no es la que usamos en el procesado de señales o de imágenes, pero por facilidad a la hora de hablar de esta operación usaremos dicho nombre.

También es posible implementar una deconvolución con una convolución directa, ya que en parte se siguen utilizando las mismas variables, ***kernel, strides*** *y* ***padding*****.** Como podemos ver en la siguiente figura el ejemplo visual de la operación de la deconvolución:

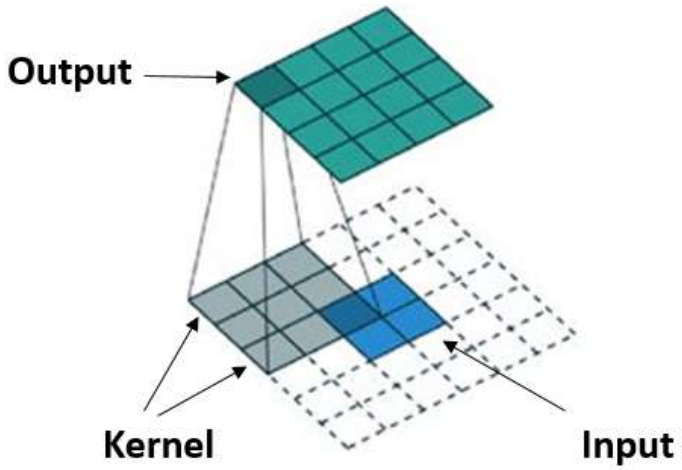

Figura 19 Operación Convolución transpuesta

Con ello conseguimos realizar un **up-sampling** de una imagen muy pequeña en una imagen cada vez mayor, lo cual para nuestra aplicación es lo que realmente necesitaremos.

Nuestro modelo de red será un conjunto de los anteriores conceptos vistos, ya que utilizaremos una versión modificada de un tipo de red neuronal convolucional.

3.6. U-NET, RED NEURONAL CONVOLUCIONAL PARA SEGMENTACIÓN DE IMÁGENES BIOMÉDICAS:

Según el documento (Ronneberger et al., 2015) sus autores conocían que el uso de las redes neuronales convolucionales eran para tareas de clasificación, donde la salida de una imagen estaba etiquetada a una única etiqueta de clase. Sin embargo, en muchas tareas de la computación visual, sobre todo en los campos donde se procesan imágenes biomédicas, la salida deseada debería incluir localización y clasificación de lo que supuestamente se asigna a cada pixel de las imágenes. Además, miles de imágenes de entrenamiento son normalmente usadas detrás de las tareas biomédicas.

Vamos a modificar y extender el concepto de arquitectura para las redes convolucionales de tal forma que pueda llegar a trabajar con muy pocas imágenes de entrenamiento y pueda llegar a producir segmentaciones más precisas.

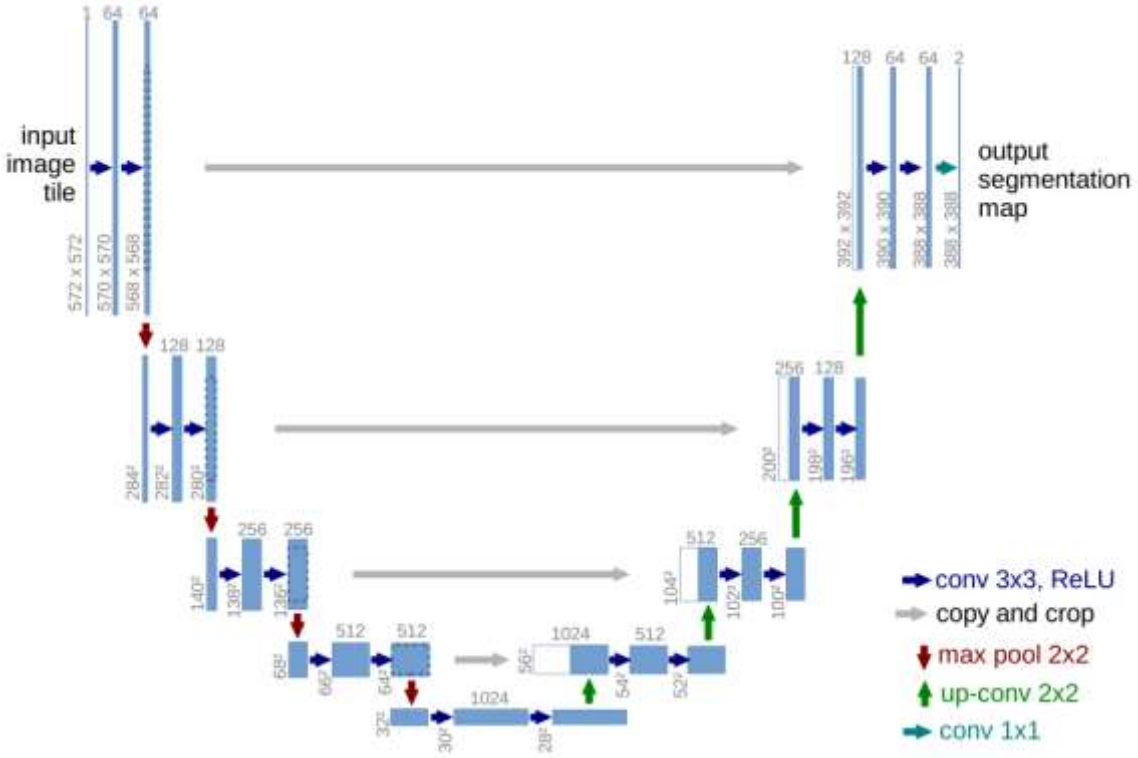

Figura 20 Arquitectura básica de una Unet

La principal idea es complementar una red habitual con capas sucesivas, donde los operadores de *pooling* son sustituidos por operadores de ***upsampling***. Por lo tanto, estas capas pueden aumentar la resolución de la salida, con la finalidad de localizar las características de mayor resolución del camino que seguimos hasta abajo, para que luego se puedan llegar combinar con el camino ascendente de salida.

Una modificación importante de la arquitectura respecto a otras es que en la parte de ***upsampling*** tenemos también un gran número de canales de las características, los cuales permiten a la red propagar la información del contexto a capas de mayor resolución, (no son resoluciones tan pequeñas como las que alcanzamos en el mínimo de la red). Una de las consecuencias es que el camino que reduce tiene que ser en gran parte simétrico, lo que produce la forma de U dando lugar al nombre de esta arquitectura (Ronneberger et al., 2015).

La red no utiliza capas ***Fully Connected*** y solamente utiliza la parte de cada convolución, por ejemplo el mapa de características que tiene pixeles en él.

Siguiendo además la Figura 21 (Ronneberger et al., 2015) de la arquitectura en forma de U mostrada en el artículo.

Consiste en un camino que contrae y otro camino que expande:

1. El camino que contrae las imágenes de entrada en la red, es la parte izquierda donde si nos fijamos bien no deja de ser la típica arquitectura de una red convolucional, el cual consiste en realizar dos convoluciones de 3x3, seguidas por una función de activación ReLU y una capa de *pooling* (donde se toma el máximo) de 2x2 con un ***stride*** de 2 para que la resolución vaya disminuyendo a medida que aplicamos este conjunto de capas de forma repetida. Además cada vez que contraemos la dimensión de las imágenes, doblamos el número de canales.
2. En cambio, el camino que expande los mapas de características obtenidos al final del camino 1 consiste en realizar el ***upsampling*** de los canales seguido de una deconvolución de 2x2 que divide en dos el número de canales, donde hay una concatenación con el correspondiente mapa de características recortado del camino de expansión con el mapa de características del camino de contracción recortado (deben tener la misma dimensión ambos mapas).Después de la concatenación se realizan dos convoluciones de 3x3 seguidos por su función de activación ReLU. La parte del recorte es necesario debido a que pierden varios pixeles en los bordes de cada convolución.
3. La parte final o en este caso la capa final es una convolución de 1x1 que mapea cada vector de características de en este caso 64 componentes al número deseado de clases que tenemos, en este caso 2.

La arquitectura U-Net ha logrado avances en el rendimiento en los diferentes campos de la segmentación de imágenes biomédicas, es por eso que será nuestro modelo de este Trabajo Final de Grado aplicando además características vistas en este bloque y en los anteriores (Ronneberger et al., 2015).

BLOQUE 4: BASE DE DATOS BIÓMEDICA BRATS.

4. BASE DE DATOS BIOMÉDICA BRATS

Respecto a la introducción del bloque 1, la realización de este Trabajo Final de Grado se ha basado en el desarrollo de una aplicación para segmentar imágenes biomédicas mediante *TensorFlow*.

Por este motivo, es necesario el desarrollo de un bloque para explicar con qué tipo de imágenes biomédicas hemos trabajado, más bien con que base de datos, además de poder explicar de una forma sencilla que es lo que buscamos en la segmentación de las imágenes.

Por otro lado se ha intentado seguir un dataset acorde a las necesidades del grupo GTI ya que se necesitaba desarrollar una aplicación para detección de tumores cerebrales y conseguir una base de datos de este tipo no es fácil, debido a que no están segmentadas.

Tras consultar los documentos proporcionados (B. H. Menze *et al, 2015) y* (M. Havei et al, 2017) se ha utilizado el dataset BRATS debido a que principalmente las imágenes proporcionadas tienen una máscara segmentada por profesionales especializados en el campo neurológico.

4.1. INTRODUCCIÓN

BraTS siempre ha estado centrado en la evaluación del estado del arte para métodos de segmentación de tumores cerebrales mediante el uso de las Imágenes por Resonancia Magnética, las cuales se suponen que son imágenes anatómicas tridimensionales detalladas, sin el uso de una radiación dañina (Imagen por Resonancia Magnética (IRM), n.d.).

La base de datos BraTS 2017 utiliza resonancias preoperatorias multiinstitucionales y se centra en la segmentación de tumores cerebrales intrínsecos heterogéneos los cuales tiene diferente forma, apariencia e histología, estos tumores cerebrales son conocidos como gliomas.

4.2. LOS GLIOMAS

Los **gliomas** son frecuentemente los principales tumores cerebrales que una persona adulta puede desarrollar, normalmente se originan por las células gliales las cuales proporcionan el soporte estructural y metabólico a las neuronas, la principal característica es que las células cancerosas se infiltran en los tejidos adyacentes provocando la creación de los tumores cerebrales (M. Havei et al, 2017).

A pesar de los múltiples avances en las investigaciones contra los gliomas, el diagnóstico de los pacientes que sufren esta enfermedad es muy pobre. La población clínica con el tumor más agresivo de la enfermedad se ha clasificado como **HGG** o ***high-grade gliomas***, los pacientes que son clasificados con esta etiqueta tienen una tasa de supervivencia media de dos años o menos y requieren tratamiento inmediato. Por otro lado el desarrollo de esta enfermedad de una forma más lenta en la población clínica es catalogada como **LGG** o ***low-grade glioma*** (M. Havei et al, 2017).

Para ambos grupos, los protocolos se utilizan antes y después de aplicar un tratamiento para evaluar la progresión de la enfermedad y ver si ha habido éxito en la estrategia del tratamiento elegido. Es así como los estudios clínicos estudian las imágenes y se evalúan.

Las rutinas de procesamiento de imágenes que pueden analizar automáticamente las exploraciones de los tumores cerebrales pueden llegar a tener un gran valor potencial respecto a la mejora del diagnóstico, la planificación del tratamiento y el seguimiento de dichos pacientes individuales (M. Havei et al, 2017).

Sin embargo el desarrollo de las técnicas automatizadas para la segmentación de los tumores es técnicamente bastante complicada, debido a que las áreas que contiene la lesión solo se pueden definir con cambios de intensidad que son relativos al tejido normal (en otras palabras se puede confundir la superficie sana con la superficie que contiene la enfermedad).

Además las estructuras tumorales son totalmente diferentes entre los pacientes y más si pertenecen a diferentes categorías ya que en términos de tamaño, forma y localización cada corte de cada paciente no se asemeja a otro corte perteneciente a otro paciente. Estas limitaciones hacen que la aplicación aumente en dificultad.

Esto nos proporciona una gran variedad de diferentes modalidades en las imágenes proporcionadas en el dataset, lo que permite dentro de la dificultad el poder segmentar de una forma precisa los tumores para nuestra aplicación.

4.3. SECUENCIAS DE LAS IMAGENES DE RESONANCIA MÁGNETICA.

El dataset clínico de BraTS proporcionado consiste en 210 pacientes pertenecientes a la población que tiene un **glioma** o **glioblastoma** bastante expandido a lo largo del cerebro (**HGG**) y 75 pacientes pertenecientes a la población que tienen un tumor cuya velocidad de expansión y forma es mucho más lenta y pequeña (**LGG**) (M. Havei et al, 2017).

Las imágenes representan un volumen de 150 cortes de cada paciente y una máscara segmentada de forma manual para corte, además las imágenes de resonancia magnética tienen diferentes contrastes, lo que hace que tengamos el mismo corte en diferentes tonalidades en la escala de grises.

Los contrastes son los siguientes:

- **T1.**
- **T1c.**
- **T2.**
- **FLAIR.**

Como podemos ver en las siguientes figuras:

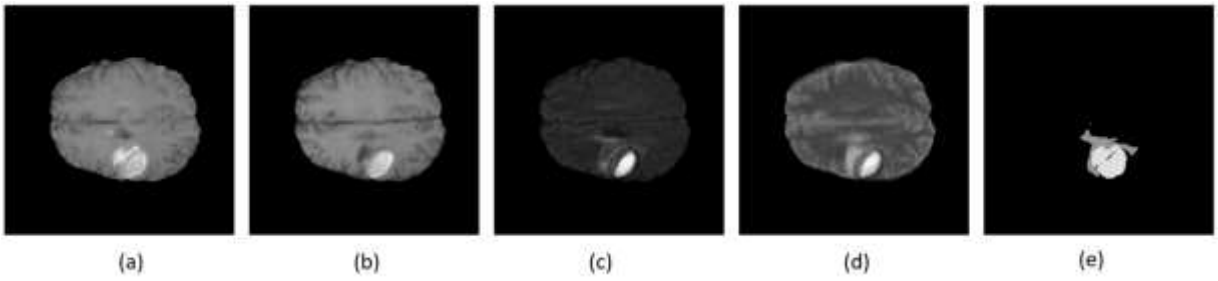

Figura 21 (a): T1c, (b): T1, (c): FLAIR, (d): T2, (e): Máscara Segmentada.

Los cerebros que disponemos para nuestra aplicación vienen con una máscara segmentada con 4 etiquetas de las estructuras tumorales las cuales son:

- *Non-Tumor*
- *Necrotic & Non-Enhancing Tumor*
- *Peritumelar Edema*
- *Enhancing Tumor*

Siguiendo las propuestas de (M. Havei et al, 2017), trabajaremos con los cortes en 2D debido a que el dataset no dispone de la resolución isotrópica y el espacio en la tercera dimensión (No se recomienda trabajar con volúmenes).

Además utilizaremos métricas para valorar los resultados. Principalmente utilizaremos las fórmulas de ***Senstivity, Specificity*** *y* ***Dice*** *o* ***F1-Score*** para saber el rendimiento de la aplicación. Por otro lado las etiquetas agrupadas de cierta manera generan las **regiones tumorales** las cuales también vamos a evaluar con las fórmulas anteriores.

Las **regiones tumorales** son las siguientes:

a. La región **completa** del tumor, la cual incluye todas aquellas etiquetas marcadas con las estructuras tumorales.
b. La región del **núcleo** o ***core*** del tumor, incluyen todas aquellas etiquetas marcadas con las estructuras tumorales excepto el edema.
c. La región **enhancing** del tumor, la cual incluye solamente aquella etiqueta con la estructura tumoral **enhacing tumor**.

Para cada región utilizaremos las formulas vistas anteriormente que serán explicadas en un bloque posterior.

Finalmente en este bloque hemos presentando un set de datos acorde a nuestra aplicación para segmentación de imágenes biomédicas basada en resonancias magnéticas donde hay unas estructuras tumorales segmentadas por profesionales que nos permiten hacer uso de la tecnología del *Deep Learning*, más concretamente en el uso de las redes neuronales convolucionales y cómo valoraremos los resultados conseguidos.

BLOQUE 5: IMPLEMENTACIÓN Y DESARRILLO DE LA APLICACIÓN.

5. IMPLEMENTACIÓN Y DESARROLLO DE LA APLICACIÓN

En este bloque 5 utilizaremos todo aquello que necesite y requiera nuestra aplicación para segmentación de imágenes biomédicas, desde cómo se van a leer los datos hasta cómo va a ser nuestra salida mostrando los resultados pertinentes.

A lo largo del bloque veremos punto por punto, los distintos ficheros que se han desarrollado en *TensorFlow* y veremos cada uno con detalle.

5.1. PREPROCESAMIENTO DE LOS DATOS.

En este primer apartado como en el título se indica, vamos a trabajar con nuestra base de datos BraTS, antes de empezar a utilizar los conceptos de *Deep Learning*. Procedemos a leer las imágenes de cada paciente mediante la librería ***nibabel*,** la cual nos permite trabajar con imágenes procedentes de resonancias magnéticas tanto en lectura como en escritura ya que el formato de nuestro dataset está realizado mediante el estándar **DICOM** (Digital Imaging and Communications in Medicine).

Primeramente juntaremos todos los pacientes tanto del **HGG** como del **LGG** ya que antes de leer las imágenes es necesario hacer una partición de los datos.

Para ello nuestro dataset de 285 pacientes va a ser dividido en tres, ya que necesitaremos tres tipos de sets, uno para el ***training*** o **entrenamiento**, otro para la **evaluación** o **validación** y un último para el ***test*.**

Tipos de set:

- **Set de Entrenamiento:** Es el set que contiene el gran grueso de los datos con mayor proporción que el resto, lo utilizaremos para que el modelo vaya aprendiendo y mejorando, habrá ciertos parámetros que no podremos modificar, y se actualizan de forma autónoma.
- **Set de Evaluación o Validación:** Es el set que utilizaremos como medida para ver como evoluciona el rendimiento del entrenamiento. Este set se ejecutara tras completar las iteraciones del entrenamiento y se evalúa el modelo.
- **Set de Test:** Es el set que menos proporción de datos tiene, los datos tienen que ser totalmente disjuntos a los anteriores sets es decir no se han podido usar para ninguno de los otros sets, esto se debe a que se puede producir un "dopaje" a la hora de sacar los resultados ya que nuestro modelo ha podido memorizar dichos datos con lo cual ya estarían contaminados. En este set solo se realiza una iteración sobre los datos, no como puede ocurrir en el entrenamiento o en la validación.

Para dividir los datos hay que tener en cuenta un par de recomendaciones, la primera es el número de muestras (cortes) en nuestros datos y la segunda es el modelo con el que vamos a entrenar.

En el caso de modelos con un número de hiperparámetros menor a otros modelos serán más fácil de afinar y validar, así que probablemente se podrá reducir el tamaño del

dataset de validación, pero en el caso de que nuestro modelo tenga múltiples hiperparámetros que afinar, lo que necesitaremos tener es un dataset algo mayor.

La norma general índica dividir primeramente de forma aleatoria los datos en dos sets, uno para el entrenamiento y otro para el test. Después de hacer esta división el dataset de test no se toca más. Por otro lado se vuelve a dividir el dataset de entrenamiento del que surgirá el set de validación.

En nuestro caso hemos decidido hacer una primera división de un 80% de los datos para el dataset de entrenamiento y un 20 % para el dataset de test. Después se produce la división del set de entrenamiento de una forma aleatoria con las siguientes proporciones un 75% de los datos finalmente son para entrenamiento y un 25% de los datos son para validación.

Si hacemos el cálculo general para los 3 sets de datos las proporciones son las siguientes:

- Un 60 % para el set de entrenamiento.
- Un 20 % para el set de evaluación.
- Un 20 % para el set de test.

Una vez realizados las divisiones de los sets de datos de la aplicación es necesario conseguir los mejores cortes de cada paciente, debido a que cada uno tiene un gran número de cortes, es decir se pueden producir fallos de hardware debido a la falta de memoria RAM.

Este problema plantea que aunque tengamos una gran cantidad de cortes debemos optimizar desde el principio el uso de los recursos de nuestra máquina ya que no disponemos de un hardware infinito. Para ello vamos a realizar un algoritmo muy simple y muy sencillo.

De cada paciente vamos a coger los **N** mejores cortes, como podemos observar en la Figura 22:

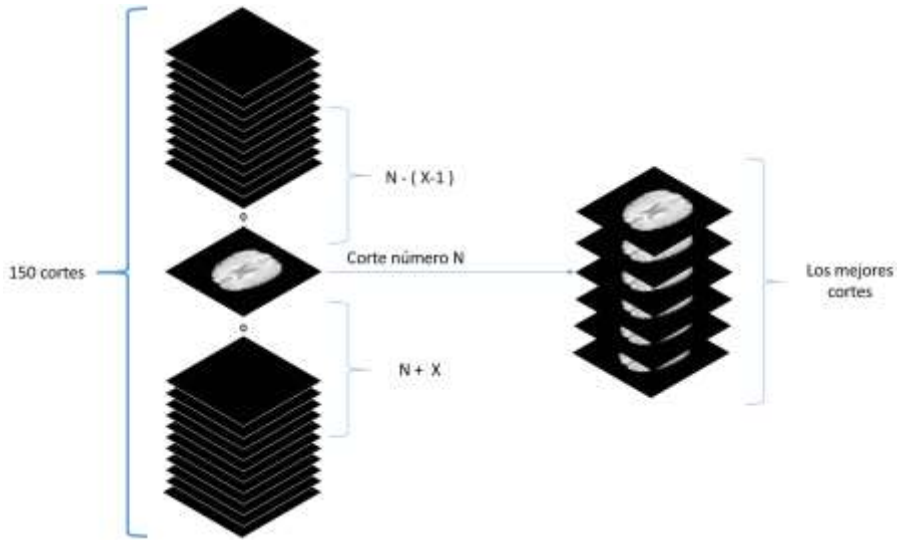

Figura 22 Algoritmo de los N mejores Cortes

Para ello lo primero es buscar que corte tiene mayor número de etiquetas distintas a cero, es decir buscar el corte que tenga más pixeles etiquetados con valores mayores a 1 ya que la clase 0, recordemos que era la clase que no tenía ningún tipo de tumor.

Una vez encontrado el mejor corte realizamos acorde al hardware de nuestra máquina la elección del número de cortes que queremos por paciente. En mi caso después de varias pruebas realizadas empíricamente, llegue a la conclusión de que se podían coger unos 30 cortes por paciente (dependiendo de la máquina el número de cortes puede ser flexible).

En la parte de preprocesado, se realiza la normalización de los datos ya que es una técnica para facilitar al modelo el aprendizaje, además de que los valores de nuestros cortes sin normalizar están demasiado distanciados con lo que el tiempo que puede llevar a la hora de realizar el entrenamiento puede ser mayor.

Así que utilizaremos la siguiente fórmula 24 de normalización:

$$imagen\ normalizada = \frac{imagen - imagen.mean()}{imagen.std()} \tag{24}$$

Hay que tener también en cuenta otro punto que no se menciona en la base de datos BraTS 2017, las etiquetas de la máscara son 0, 1, 2 y 4 en la versión de la base de datos decidieron eliminar la etiqueta 3, así que será necesario hacer un cambio en todas las máscaras de los cortes seleccionados, para que los valores vayan de 0 a 3 para nuestras etiquetas.

Por último es necesario dividir al menos en 3 splits los datos del entrenamiento para facilitar la introducción en nuestro modelo.

Utilizaremos un tipo formato especial llamado ***TFRecord*** cuyos creadores son los desarrolladores de *Tensorflow*, este formato novedoso plantea una idea para almacenar los datos en secuencias binarias en crudo llamado records. Permitiendo comprimir los datos de una forma eficiente especialmente para aquellos *datasets* que son demasiado grandes para mantenerlos en memoria, esto es una ventaja ya que solamente los datos (cada ***batch***) son utilizados cuando se necesitan en cada iteración.

Para ello se utiliza un componente a la hora de programar que es el ***Example*.** Antes de nada hay que ver cuantas características son necesarias de almacenar. En nuestro caso son los vectores de imágenes con los cuatro canales y el vector de las máscaras. Así que lo siguiente es crear las listas para poder convertirlo a bytes.

A continuación usamos el diccionario de ***feautures*** que contiene las características que serán convertidas a su tipo de lista, después se crea el ejemplo de buffer que es el ***Example*** y por último se serializan las cadenas de bytes y se escriben en los ficheros .tfrecord.

La utilización de este tipo de ficheros es una manera conveniente de conseguir que nuestros datos puedan ser usados en un ***pipeline*** o tubería para nuestro modelo de aprendizaje y liberen en cierta medida un uso excesivo de memoria.

5.2. READER O COMPROBACIÓN DE LOS DATOS GENERADOS TIPO TFRECORDS

Antes de programar nuestro modelo de aprendizaje profundo para la segmentación de imágenes de resonancias magnéticas, es necesario el desarrollo de una pequeña aplicación de *TensorFlow* para poder comprobar nuestras imágenes y comprobar que nuestros ficheros de *tfrecords* se habían generado de una forma correcta.

Para ello simplemente hemos creado un grafo en el cual no hay ningún tipo de red sino que simplemente tenemos la carga en una cola FIFO de los ficheros que irán haciendo los pasos inversos al orden de creación, para poder volver a convertir las cadenas de bytes puestas de forma secuencial en los canales y cortes generados, que durante la sesión se irán leyendo de forma aleatoria y se mostraran por pantalla cada x número de pasos.

Esta aplicación nos sirve de antesala para poder desarrollar el modelo, nos muestra si ha habido algún fallo en la creación de los ficheros o en la lectura de los mismos.

5.3. ENTRENAMIENTO Y EVALUACIÓN DE LA APLICACIÓN

En esta parte nos centraremos en el núcleo de la aplicación, aquí es donde vamos a aplicar toda la parte teórica vista en los anteriores bloques para conseguir realizar tanto el entrenamiento como el validación.

Lo primero de todo será seleccionar los ficheros de *tfrecords* para el entrenamiento de una forma aleatoria para la entrada y seleccionar el fichero de *tfrecord* para la validación. A continuación fijamos todos los parámetros necesarios para el correcto funcionamiento de la aplicación:

- Longitud de *batch* del entrenamiento y del validación
- La anchura, altura de las imágenes y los canales de entrada a la red
- Los directorios para guardar los *checkpoints* de la red, imágenes y gráficas.
- El *Learning Rate* inicial con el que partiremos, el contador de paciencia, el contador de stop para la aplicación del *Reduce Learning Rate on Plateau* y el factor por el que se va a ir disminuyendo el valor del *Learning Rate*.
- El tamaño de *batch* que utilizaremos a la entrada de nuestra red no puede ser muy alto debido a que no tenemos memoria ni GPU suficiente para que sea más rápido el procesado de los cortes durante la red.
- El número de *epochs* máximo que podrá realizar nuestra aplicación.

Una vez configurados todos aquellos parámetros e hiperparámetros para la realización correspondiente de las pruebas procedemos a crear un grafo.

Al crear el grafo, lo primero que vamos a hacer es la carga de los ficheros de datos, para ello utilizaremos una lectura de los cortes de forma aleatoria para ser introducidos en nuestro modelo.

Después de tener los cortes elegidos en lotes de tamaño *batch*, en algunas pruebas se realizara el ***Data Augmentation*** que realizara diferentes procesados a las imágenes, para que nuestro algoritmo no se espera las misma entrada de forma constante.

Las principales modificaciones hechas en las pruebas son las siguientes:

- ***Crop*****:** Recorta la imagen de forma uniforme y aleatoria, además de centrar la foto con las dimensiones que demos.
- ***Flip_left_right*****:** Realiza o no de forma aleatoria el giro como si fuera un espejo de la imagen proporcionada de derecha a izquierda o de izquierda a derecha
- ***Flip_up_down*****:** Realiza o no de forma aleatoria el giro como si fuera un espejo de la imagen proporcionada de arriba a abajo o de abajo a arriba.
- ***Brightness*****:** Realiza una modificación en la imagen aumentando o disminuyendo el brillo entre un valor de 0 y 1, siendo el 0 el valor más oscuro y el 1 el más alto.

Una vez hayamos usado la técnica de *Data Augmentation* en algunas pruebas y en otras no, procederemos a ir introduciendo los lotes a la entrada del modelo de la red que estemos utilizando.

Cuando consigamos obtener los resultados de la última capa realizaremos por un lado la comparativa de las probabilidades de las predicciones tanto para el entrenamiento como para la validación con la máscara real, obteniendo una máscara generada en algunos casos similares a la real.

Por otro lado es necesario calcular la función de pérdida o coste para determinar el error entre los valores estimados y los valores reales, para ello obtenemos el valor del ***loss*** para ambos (entrenamiento y validación) ya que podremos ir visualizando cómo evoluciona, pudiendo aumentar o disminuir el valor, pudiéndose producir ***overfitting*** (sobre entrenamiento) o no.

Por último será necesario aplicar nuestro algoritmo de optimización para que se puedan actualizar nuestros pesos y sesgos durante todo el entrenamiento realizándose la propagación hacía atrás, pues la validación no realiza esta tarea.

La parte que corresponde al grafo está completamente unida y funcionando, ahora es necesario establecer una sesión para que se puedan introducir todos aquellos datos de forma correcta y ordenada en un bucle de repetición que realice la obtención de los resultados del entrenamiento y a continuación los de la evaluación.

Una vez conseguido todo lo anterior empezaremos a ver los diferentes valores de la función de ***pérdida***, nos centraremos en los valores que se obtienen en la validación**,** para poder aplicar la reducción del Learning Rate a través del contador de paciencia o parar el entrenamiento y la validación en caso de que se haya consumido el máximo de iteraciones del contador de stop.

Por otro lado cada vez que en un epoch la media de los valores de las perdidas en validación es mejor que cualquiera de las otras medias calculadas en los anteriores *epochs*, procederemos a realizar un checkpoint siendo este el mejor de todos.

El anterior *chekpoint* que hubiera en ese directorio desaparecería y solo se mostraría este nuevo checkpoint. Al producirse una mejora del valor de la pérdida desencadena el que se reinicie el contador de paciencia a cero y cuando la prueba finaliza también se procede a realizar un *chekpoint*, en caso de querer partir de este punto en un futuro.

Por último podemos observar las gráficas de la media de los valores de las pérdidas a lo largo del entrenamiento y de la validación así como la obtención de una gráfica que muestra la disminución del *Learning rate* en cada epoch de la prueba.

5.4. TEST DE LA APLICACIÓN

Una vez que hemos realizado el entrenamiento y la validación es necesario comprobar los resultados obtenidos de nuestra red y si pueden llegar a ser validos o no.

Utilizaremos un set de test ajeno a los otro dos set de datos, en este caso se utilizara el mejor *checkpoint* calculado, desde el que la red cargará sus pesos y sus sesgos, es decir utilizaremos el checkpoint cuyo valor medio de las perdidas obtenidas en cada epoch ha sido el mejor.

Para esta parte de la aplicación no es necesario realizar múltiples *epochs* simplemente recorreremos una única vez todo el set de datos del test para que no pueda haber ningún tipo de influencia a posteriori.

Simplemente es necesario conocer el número de *batch* y el número de cortes totales para ir cargando las imágenes en lotes de *batches* de dicho valor.

Tras realizar la configuración pertinente, Se realizan las pruebas de testeo donde al final del programa procesaremos las métricas de toda la prueba de test. Además se calcularan en cierta medida los diferentes resultados para las ecuaciones de *specifity, sensitivity y dice o F1-score* para cada clase y para cada región.

Se mostrará una matriz de confusión ubicando cada métrica obtenida en cada elemento de la matriz para cada clase predicha o real y una matriz de confusión normalizada a partir de la anterior para valores entre 0 y 1, esto permite poder valorar mejor los resultados de una forma gráfica. Estos conceptos se verán en profundidad en el siguiente bloque ya que se encargará de evaluar cada resultado para las diferentes pruebas que hagamos.

Por último aquí obtendremos una imagen de las predicciones que se guardará para cada corte analizado del *batch* y se procederá a guardarlas en un directorio.

También dispondremos de una matriz de confusión y una matriz de confusión normalizada para cada corte en formato “instantáneo” donde se ven los resultados obtenidos únicamente para dicho corte.

5.5. ARQUITECTURA DE LOS MODELOS DE RED DE LA APLICACIÓN

Veamos cómo van a ser los diferentes ejemplos de las redes neuronales convolucionales usados en *TensorFlow* para nuestra aplicación ya que aunque realicemos el grueso de las pruebas con un modelo de red, es necesario probar algún modelo más de red con algún tipo de modificación o planteándola desde cero.

Por otro lado aparte de los modelos es necesario concretar los valores que se necesitan a lo largo de las capas, tanto las capas de convolución como de *pooling* y de *dropout*.

5.5.1. MODELO COVNET

Este modelo de red consiste en una sucesión muy pequeña de capas convolucionales seguida de una capa de dropout, en el camino de *upsampling* utilizaremos un par de capas deconvolucionales. Como podemos observar en la Figura 23:

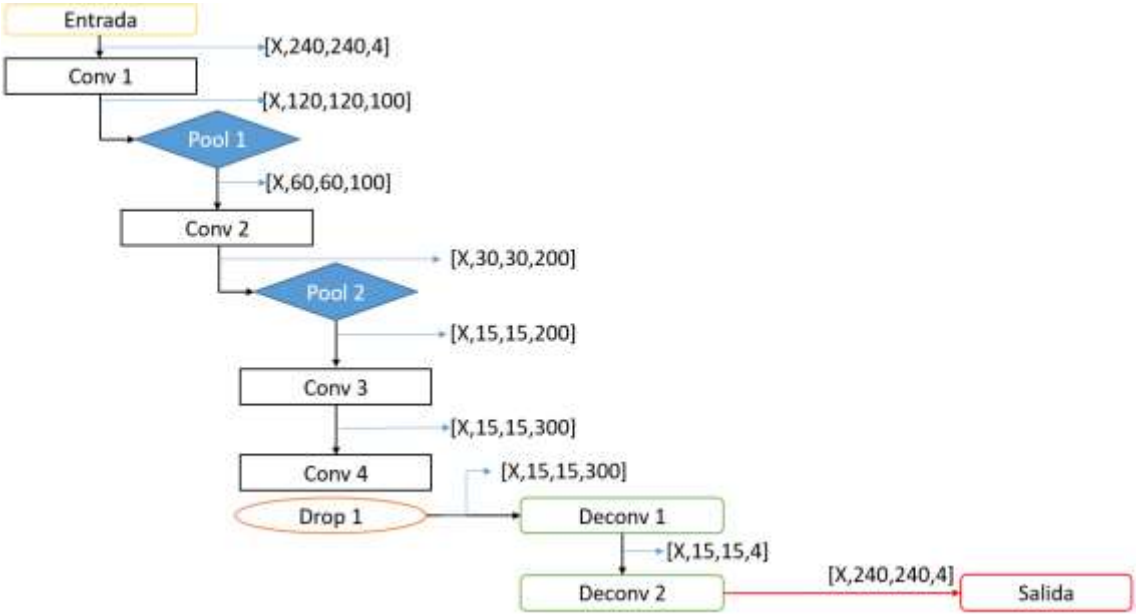

Figura 23 Modelo Covnet

Cuando realicemos las pruebas no nos vamos a centrar demasiado en este modelo de red, ya que su principal objetivo es conseguir que funcione toda la aplicación de una forma correcta, siendo los resultados esperados favorables o no.

5.5.2. MODELO U-NET MODIFCADO.

El principal grueso de las pruebas se ha realizado con este modelo, como podemos observar en la Figura 24, que principalmente consiste en una sucesión de capas, una capa convolucional seguida de una capa de dropout para la parte que realiza la reducción de los datos y aumenta el número de filtros usados para buscar mejor los mapa de características y por otro lado tenemos una sucesión de capas de deconvolución que lo que permiten es volver a reconstruir las imágenes a las

dimensiones originales que teníamos en un principio, disminuyendo el número de canales en esta parte de la red.

Para ello nuestras capas convolucionales utilizarán las siguientes características:

- **Filtros**: desde 128 en la primera capa hasta 2048 usada en la última capa.
- **Tamaño del kernel:** dependiendo de la capa, pero principalmente se usará una ventana de 3x3
- ***Strides:*** El valor por defecto será de 1 en cada dimensión, ya que al ser el tamaño de paso con el que se desliza la ventana, no nos interesa hacer grandes saltos, entre las ventanas.
- ***Padding*****:** Para todas las capas convolucionales se utilizara el valor *same*, lo que es lo mismo a "padear" o rellenar con ceros alrededor de la imagen en caso de que no haya valor en esa zona evaluada.
- ***Activation*****:** La función de activación podrá variar, siguiendo los pasos del documento (Ronneberger et al., 2015) por lo que aplicaremos la función ReLU en la mayoría de los casos, pero también se han utilizado las funciones Sigmoid y TanH.

Para las capas de dropout utilizaremos el mismo valor para todas ellas, aunque si variaremos el valor para las distintas pruebas realizadas.

En el caso de las capas de las capas deconvolucionales utilizarán las siguientes características:

- **Filtros:** En este caso es necesario utilizar tensores con la siguiente forma [*height, width, output_channels, in_channels*] además de que el valor de los canales de entrada tienen que coincidir con los canales del tensor que utilizamos como entrada. En este caso los filtros que utilizaremos tendrán un valor acorde a los canales de entrada y salida que dependan de cada capa
- ***Output Shape*****:** Simplemente hay que indicar la representación de las dimensiones para la salida que queramos de la operación deconvolución
- ***Strides*****:** El valor por defecto será de 2 en las dimensiones que tienen que ver con la anchura y altura de las imágenes, para las dimensiones que son acordes al número de *batches* y a los canales debe ser 1 siguiendo las recomendaciones del método *tf.nn.conv2d_transpose*
- ***Padding*****:** Para todas las capas deconvolucionales se utilizara el valor *same*, lo que es lo mismo a padear o rellenar con ceros alrededor de la imagen en caso de que no haya valor en esa zona evaluada.

A la hora de realizar las pruebas, se aprecia con bastante facilidad que las capas convolucionales es donde más memoria se necesita, esto se debe a que si mostráramos todos los parámetros del modelo que tienen que ser aprendidos veríamos que el consumo de memoria aumentaría, además de todas las matrices de pesos que hay y los sesgos, que en parte también realizan una carga en memoria.

Es importante saber que cuanta más profundidad tiene nuestro modelo más difícil es de conseguir ajustar la red correctamente.

Cuando veamos las imágenes en la sección de resultados observaremos que la red funciona correctamente para las características que buscamos, en algunos casos no obtendremos unos buenos resultados a nivel numérico pero el uso de esta red permite generar máscaras para un análisis visual más rápido e intuitivo.

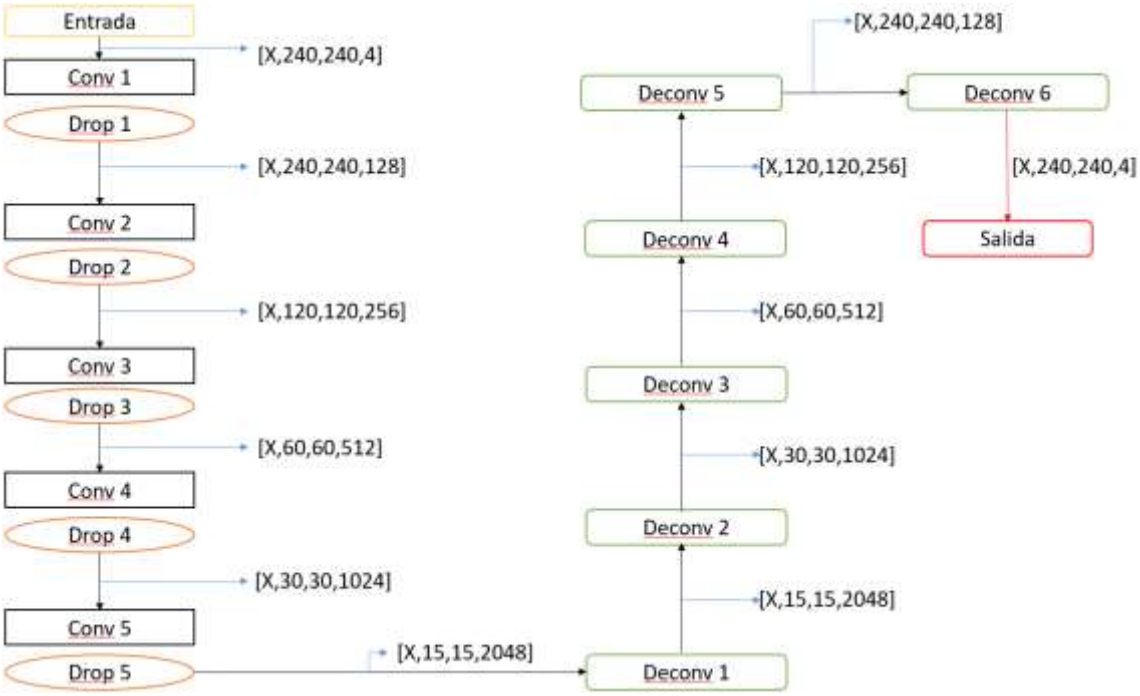

Figura 24 Modelo U-Net modificado

5.5.3. MODELO U-NET MODIFICADO CON CAPAS DE POOLING.

El planteamiento de la red anterior es correcto salvo porque pueden llegar a faltar las capas de pooling, recordemos que estas capas suelen ir a continuación de las capas convolucionales, en esta red irán a continuación de la capa de dropout ya que la proporción de neuronas que se apaguen pueden afectar a la operación de *max_pooling*. Los valores para las capas convolucionales y deconvolucionales serán similares a los mostrados del anterior modelo de Unet.

En cambio para este modelo de red se ha incluido las capas de pooling, las cuales tienen las siguientes características:

- **Operación:** Se ha elegido la operación de max pooling en vez de la de average pooling, ya que no deja de ser la operación más utilizada para las capas de pool
- ***Pool Size***: El tamaño de la matriz utilizada para disminuir el tamaño de nuestras imágenes pero consiguiendo mantener la relación espacial, usaremos matrices de 2x2.

- ***Strides*:** El valor por defecto de la longitud será de 1 ya que es el salto de la ventana de deslizamiento para cada dimensión del tensor de entrada.
- ***Padding*:** Al igual que en las capas convolucionales, en la capa de pooling se aplica el *padding* de la misma forma, utilizaremos igual que en el modelo anterior el valor *same*, lo que es lo mismo a padear o rellenar con ceros alrededor de la imagen.

Al igual que en el modelo anterior se han utilizado las capas de dropout de la misma manera, manteniendo la misma constante a lo largo de todas las capas. La siguiente figura muestra el esquema modificado de la U-net del modelo anterior con las capas de pooling:

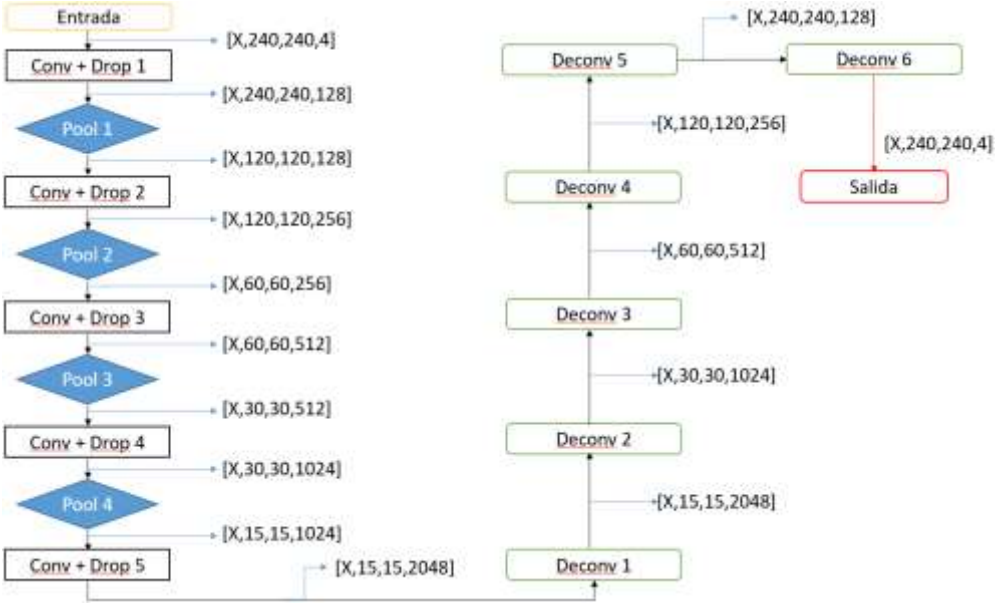

Figura 25 Modelo U-net modificado con capas de pooling

Podemos ver un par de cambios a mayores, la última capa convolucional 5 no realiza ningún tipo de pooling ni reduce más el mapa de características y ahora al liberarse las capas convolucionales de realizar la división del espacio implica que estas capas se centren en aumentar el número de canales para sacar mejor las características requeridas según vamos disminuyendo en la parte izquierda del modelo.

BLOQUE 6: RESULTADOS

6. RESULTADOS Y CONCLUSIONES.

6.1. RESULTADOS DE LAS PRUEBAS REALIZADAS

Después de haber configurado los hiperparámetros para nuestra red, se proceden a realizar diferentes pruebas con las diferentes modificaciones necesarias para que cada prueba sea completamente diferente a la anterior.

Antes de meternos en profundidad en la valoración de los resultados obtenidos en cada una de las pruebas, empezaremos explicando cuáles son las métricas utilizadas en cada una de ellas.

6.1.1. MEDIDAS ESTADÍSTICAS PARA LAS FUNCIONES DE CLASIFICACIÓN.

Para la correcta valoración de los resultados obtenidos en cada una de las pruebas, es necesario conocer qué parámetros son los que se van a evaluar. Pero antes de entrar en materia, y teniendo en cuenta que hablamos del entorno de la medicina, debemos conocer las siguientes definiciones:

- Verdadero Positivo (***True Positive***): Es un resultado en el que el modelo que estamos estudiando predice de forma correcta la clase positiva.
- Verdadero Negativo (***True Negative***): Es un resultado en el que el modelo que estamos estudiando predice de forma correcta la clase negativa.
- Falso Positivo (***False Positive***): Es un resultado en el que el modelo que estamos estudiando predice de forma incorrecta la clase positiva.
- Falso Negativo (***False Negative***): Es un resultado en el que el modelo que estamos estudiando predice de forma incorrecta la clase negativa.

A partir de este momento ya estamos en disposición de explicar qué valores son los que vamos a evaluar:

- Sensibilidad (***Sensitivity***)
- Especificidad (***Specificity***)
- Valores Predichos Positivos
- Precisión
- Dice o F1-Score

A continuación, explicamos cada uno de ellos con detalle:

Sensibilidad

La sensibilidad en una prueba que se basa en la **habilidad de detectar los tumores correctamente** en los pacientes analizados, su fórmula matemática es la siguiente:

$$Sensitivity = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{25}$$

En una prueba donde conseguimos un 100 % de sensibilidad correcta significa que nuestro modelo ha detectado pixeles tumorales en todos los cortes de los pacientes analizados, en cambio en una prueba donde no tengamos un 100% de sensibilidad significa que ha habido cortes en los que no se han detectado pixeles de tumores, es decir hay Falsos Negativos en los valores predichos para ciertas clases (Lalkhen & McCluskey, 2008).

Especificidad

La especificidad en una prueba se basa en la **habilidad de no detectar los tumores de forma correcta** en los cortes de los pacientes analizados, su fórmula matemática es la siguiente:

$$Specificity = \frac{True\ Negatives}{True\ Negatives + False\ Positives} \tag{26}$$

En una prueba donde conseguimos un 100 % de especificidad, esto significa que nuestro modelo es capaz de identificar de forma correcta todos los cortes de los pacientes donde no hay ningún tipo de pixel clasificado como tumor, en cambio en una prueba donde no consigamos un 100 % de especificidad significa que ha habido cortes en los cuales se han detectado pixeles clasificados como tumores cuando no los había, es decir que hay Falsos Positivos en los valores predichos para ciertas clases (Lalkhen & McCluskey, 2008).

Valores Predichos Positivos y Negativos

Estos valores PPV (***Positive Predicted Value***) y NPV (***Negative Predicted Value***) se utilizan de forma paralela a la Especificidad y la Sensibilidad ya que es necesario conocer con que probabilidades las pruebas que hemos realizado son correctas. La Sensibilidad y la Especificidad no nos proporcionan esta información así que en vez de tener en cuenta estos valores usaremos las predicciones positivas y negativas para acercarnos más a unos resultados mejores.

- Valor Predicho Positivo: Es la proporción de tumores con resultados positivos durante las pruebas que han sido identificados o diagnosticados de forma correcta (Lalkhen & McCluskey, 2008).

- Valor Predicho Negativo: Es la proporción de tumores con resultados negativos durante las pruebas que han sido identificados o diagnosticados de forma correcta (Lalkhen & McCluskey, 2008) (Lalkhen & McCluskey, 2008).

Las fórmulas matemáticas que magnifican dichos valores son las siguientes:

$$PPV = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{27}$$

$$NPV = \frac{True\ Negatives}{True\ Negatives + False\ Negatives} \quad (28)$$

Precisión

La Precisión es la medida que indica la exactitud con la que obtenemos los resultados verdaderos entre el número total de casos analizados.

Se trata de otra medida estadística que utilizaremos para saber si nuestro modelo realiza una clasificación multiclase donde es capaz de identificar la clase buscada o excluir el resto de clases.

La fórmula matemática que magnifica la Precisión es la siguiente:

$$Precisión = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives} \quad (29)$$

Dice o F1-Score:

La última medida estadística que utilizaremos para la clasificación multiclase es él ***Dice*** *o* ***F1-Score*** y se define como la **media del valor positivo predicho o PPV y la sensibilidad**, los valores van comprendidos entre 0 y 1 donde si conseguimos un 1 de valor quiere decir que tenemos una sensibilidad y un PPV perfectos mientras que si obtenemos un 0 como valor, los valores de las anteriores variables no son lo que buscaríamos.

La ecuación matemática que permite calcular él ***Dice*** o ***F1-Score*** es la siguiente para nuestra aplicación:

$$Dice = \frac{2 \times True\ Positives}{2 \times True\ Positives + False\ Positives + False\ Negatives} \quad (30)$$

Más adelante cuando se muestren los resultados usaremos cada una de estas medidas estadísticas para mostrar los resultados de nuestra aplicación y poder observar como de cerca o de lejos estamos de lograr nuestro objetivo para la detección de los tumores cerebrales en los cortes que hemos pre-procesado previamente (Sasaki, 2007).

6.2. MATRIZ DE CONFUSIÓN

En el anterior punto hemos podido ver cuáles son las medidas estadísticas principales para valorar y evaluar nuestra aplicación, para completarlas utilizaremos la herramienta conocida como **Matriz de Confusión** utilizada en el campo del Machine Learning y del *Deep Learning*.

Esta herramienta permite mostrar el desempeño del algoritmo para la clasificación binaria o multiclase ya que dará una idea de cómo se está clasificando dicho algoritmo, principalmente usa el conteo de los aciertos y errores pertenecientes a las clases que se pretenden clasificar, en cierta medida podemos verificar si nuestro algoritmo está clasificando mal las clases y en qué medida.

Para simplificar la explicación vamos a ver primero como se muestra una **Matriz de Confusión** para una simple clasificación binaria, es decir una matriz de dimensiones 2x2:

	VALOR DEL RESULTADO DE LA PREDICCIÓN	
VALOR REAL DE LA CLASE	VERDADERO POSITIVO (TP)	FALSO NEGATIVO (FN)
	FALSO POSITIVO (FP)	VERDADERO NEGATIVO (TN)

Tabla 1: Matriz de confusión para una clasificación binaria

Como podemos observar en la matriz, la clasificación se basa en los valores que tenemos en las filas y las columnas, por un lado los valores de las columnas son los valores de los resultados predichos para cada clase y por el otro lado tenemos el valor real de la clase en las filas.

Si cogemos las definiciones básicas del anterior punto, cada elemento de la matriz pertenecerá a un caso verdadero o falso y si este es positivo o negativo. Además si vamos a un punto más allá cualquiera de las ecuaciones vistas en el punto anterior se puede calcular fácilmente con esta matriz, así que es una herramienta que ayuda a ver de una forma más fácil o con un simple vistazo el desempeño.

Vista una matriz de un clasificador binario con dos clases, procederemos a ir un punto más allá con el ejemplo real de una matriz de una de las pruebas calculada para uno de los cortes donde ya no es una clasificación binaria sino ya que se produce una clasificación multiclase.

Supongamos que tenemos la siguiente matriz de confusión:

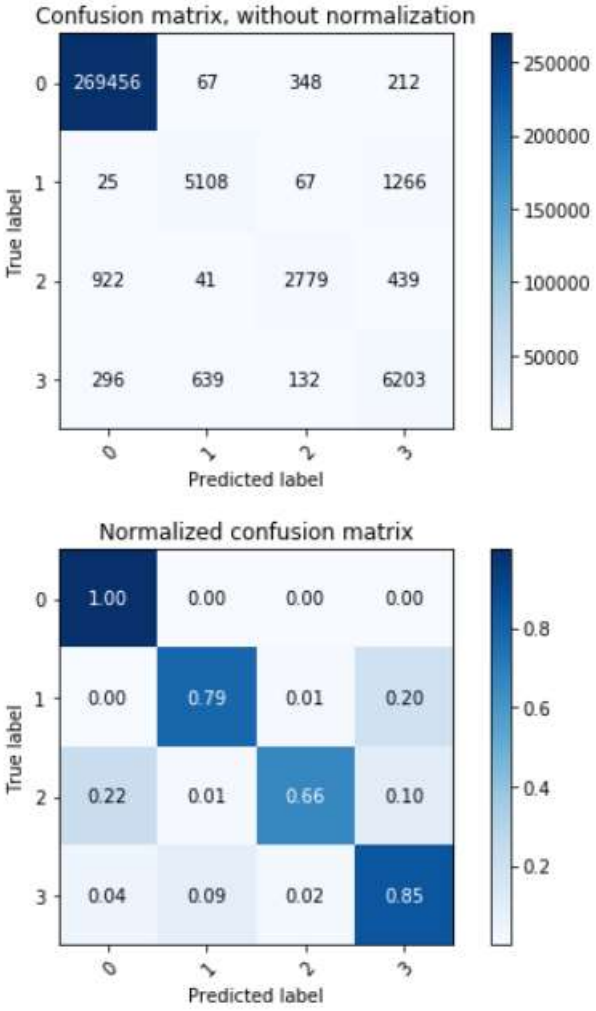

Figura 26: Matriz de Confusión Multiclase y Matriz de Confusión Normalizada Multiclase

En la Figura anterior se muestran dos matrices la matriz normal donde este cada valor del algoritmo de clasificación y en el caso de la segunda matriz la que tiene valores decimales comprendidos entre 0 y 1 corresponde con una matriz de confusión normalizada la cual permite ver de una forma mucho más sencilla y con un vistazo rápido como ha ido nuestro algoritmo para la clasificación (Learn, n.d.).

Basándonos en la tabla 1 veremos para cada clase cómo calcular los valores correspondientes a los **Verdaderos Positivos, Verdaderos Negativos, Falsos Positivos y Falsos Negativos**, para ello utilizaremos la matriz sin normalizar cuatro veces una para cada clase (Learn, n.d.).

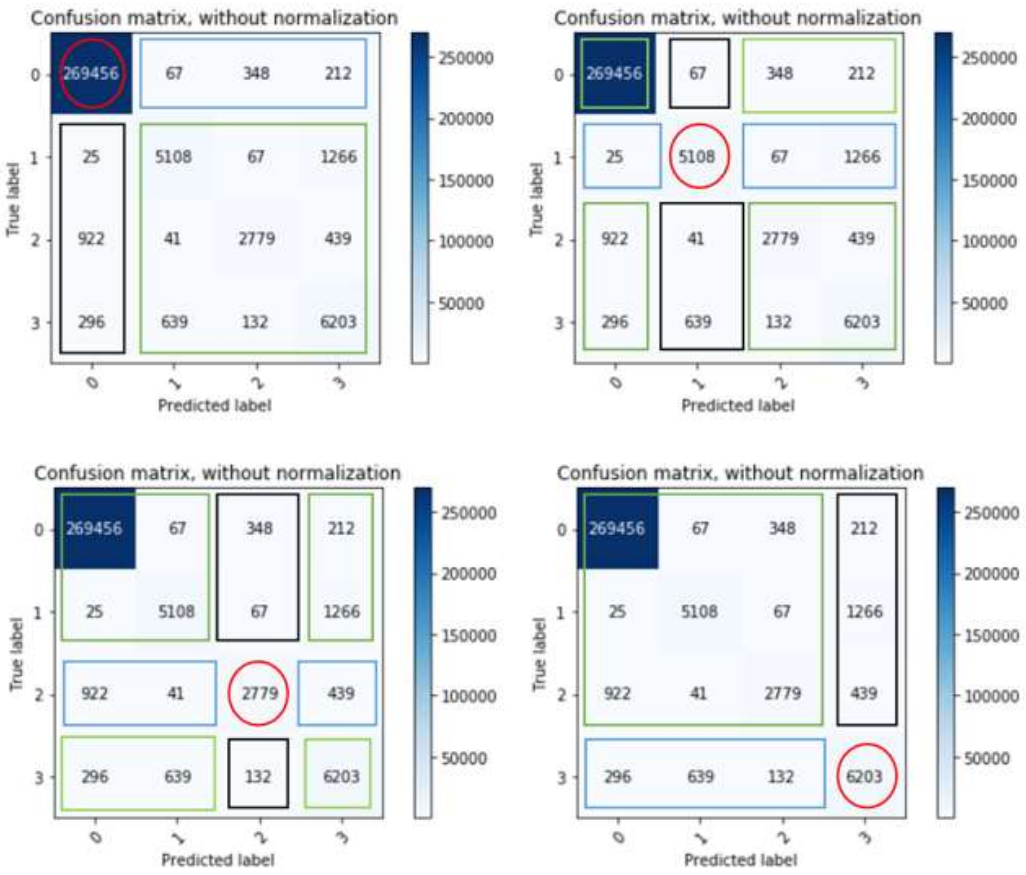

Figura 27 Cálculo de las métricas a partir de la matriz de confusión

Los valores con el círculo rojo corresponden a los valores que se han predicho de forma correcta para cada una de las clases, es decir, los **Verdaderos Positivos**. Cómo podemos apreciar son los que están en la diagonal principal en el elemento de la matriz que corresponde con la predicción y valor real de esa clase (i,i).

A continuación, si tomamos toda la fila menos ese valor es decir (i,…), los cuales son los que pertenecen a un cuadrado azul, tendremos los valores correspondientes con los **Falsos Negativos.**

Y si en vez de tomar la fila tomamos la columna menos el valor de los **Verdaderos Positivos** (…,i), los cuales son los valores que pertenecen a un cuadrado negro, conseguiremos los valores correspondientes a los **Falsos Positivos**.

Por último, si cogemos el resto de valores de la matriz, los cuales están marcados con un cuadrado verde, son los correspondientes a los valores negativos predichos de forma correcta es decir los **Verdaderos Negativos.**

Así que con todas las definiciones estadísticas vistas en el anterior punto y la herramienta visual que nos permite ver el rendimiento con un simple vistazo de nuestro clasificador, vamos a ver cómo se han llevado las pruebas de nuestra aplicación y vamos a ver los resultados de las mismas (Learn, n.d.).

6.3. PRUEBAS REALIZADAS Y RESULTADOS

Durante el bloque 3 hemos visto las diferentes operaciones llevadas a cabo por las capas convolucionales y en el bloque 5 hemos definido nuestros modelos de prueba, además en este bloque se han introducido las diferentes fórmulas que evaluaran nuestros modelos. Para dicho objetivo es necesario realizar un conjunto de pruebas.

Las pruebas están divididas en dos partes, una primera donde se ejecuta el entrenamiento y la validación de nuestro modelo, donde obtenemos las gráficas que proporcionaran como ha ido la prueba realizada siendo esta con más detalle la media de los valores de la función de pérdida del entrenamiento y la validación a lo largo de las iteraciones. Por otro lado obtendremos las diferentes imágenes predichas en entrenamiento y validación con sus respectivas matrices de confusión que valoran cada pixel de forma instantánea y no global.

La segunda parte de las pruebas tiene que ver con la parte del test, como hemos visto anteriormente se utilizara un set de datos que no han sido usados durante el entrenamiento y la validación para obtener aquí los resultados de las métricas con más precisión y sin necesidad de que estén adulterados debido a que se haya producido algún tipo de "memorización de los datos". Al final del test obtendremos una matriz confusión normalizada y sin normalizar que contendrá los diferentes valores del conjunto general de las métricas, es decir, la matriz no normalizada tiene los valores de todas las métricas de todos los cortes del test evaluados para cada pixel y clase predicha.

Para las pruebas utilizaremos tres modelos diferentes. Empezaremos por el modelo más básico de todos e iremos explicando las diferentes pruebas que se han realizado y el porqué de los cambios en ciertos parámetros.

Modelo Covnet

Este modelo consiste en un conjunto de capas muy limitadas, ya que en primer lugar lo que hay que buscar es la posible identificación de los problemas que puedan llegar a ocurrir desde el principio de la aplicación hasta el final, para ello tenemos este modelo al cual le hemos sometido a muy pocas pruebas donde no esperamos que los resultados puedan ser favorables, para ello las pruebas que hemos realizado son las siguientes:

Arquitectura	Dataset	Función de activación	Valor Dropout	Tamaño Kernels
Covnet 1	Completo y Normalizado	ReLU	0,2	1x1 y 5x5
Covnet 2	Completo y Normalizado	ReLU	0,2	3x3
Covnet 3	Completo y Normalizado	ReLU	0,2	3x3

Tabla 2 Pruebas realizadas en el modelo Covnet

En primer lugar el set de datos que utilizaremos estará normalizado ya que es de buena práctica realizar esta operación sobre los cortes que vamos a utilizar. Al utilizar capas de dropout, el valor que utilizaremos en estas pruebas es de 0,2. Respecto a la aplicación del descenso de la tasa de aprendizaje por el método RoP, usaremos los valores recomendados en el documento de (Kingma & Ba, 2015), del algoritmo de optimización Adam el cual es 10^{-4} con una reducción de la tasa por un factor de 0.1, al tener en las

pruebas la paciencia activada, quiere decir que el número de iteraciones será mucho mayor ya que no aplica unas medidas tan restringidas cuando se mejora el valor de la perdida por iteración.

Respecto a la parte de las propias capas convolucionales, las funciones de activación serán ReLU esto se debe a que son las funciones de activación menos complejas lo cual acorta el tiempo máquina de la ejecución del entrenamiento. Si nos centramos en los tamaños del kernel con los que se realizarán las convoluciones utilizaremos 5x5, 3x3 y 1x1 normalmente este tamaño de kernel se utiliza para la primera y última capa de la red.

Como podemos observar en la Figura 28 mostramos las diferentes pruebas realizadas para este modelo de red, las líneas referentes a los valores de las pérdidas por iteración en el entrenamiento y la validación muestran que hay sobreentranimiento (*overfitting*) ya que aumenta al principio la diferencia entre ellas en los primeros *epochs* y se deben a dos principales causas:

La primera es el set de datos que estamos utilizando; se recomienda observar los datos que estamos suministrando a la red ya que tienen que estar perfectamente pre procesados

La segunda causa, que es la más plausible es que esta arquitectura desarrollada no se ajusta a las necesidades de nuestros objetivos.

Por eso es necesario la realización de otro tipo de arquitectura que sea más acorde a la que trata nuestros datos.

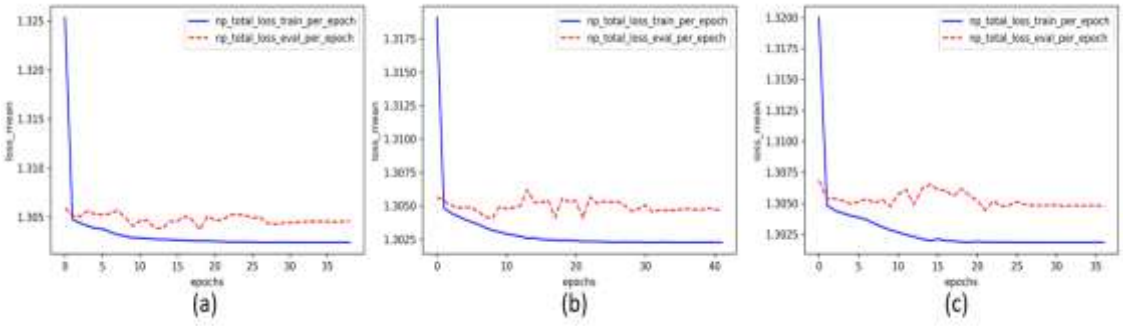

Figura 28 Gráficas de entrenamiento de las pruebas realizadas en el modelo Covnet; (a) Prueba 1; (b) Prueba 2; (c) Prueba 3

Con este modelo de red hemos demostrado que la aplicación funciona correctamente a pesar de no conseguir unos resultados favorables

Modelo U-net modificado.

Tras observar unos resultados desfavorables vamos a aplicar una arquitectura más elaborada y que se basa en la red U-net aunque con algunas capas menos, este modelo no tiene capas de pooling ya que va a ser la propia capa de convolución la que se va a encargar de realizar las tareas de la parte de *downsampling,* el objetivo de este modelo es intentar mejorar los resultados obtenidos del anterior modelo y observar si con menos capas es capaz de realizar su labor de una forma más o menos correcta.

Para este segundo modelo el número de pruebas realizadas aumenta, para estas pruebas cambiaremos la utilización de ciertos parámetros y en otro caso habrá ciertos parámetros que permanezcan fijos.

Al igual que en el anterior modelo, la utilización de la tasa de aprendizaje o *learning rate* será la misma en el comienzo de las pruebas de entrenamiento 10^{-4} y de la misma manera el factor con el que se reduce la tasa de aprendizaje es la misma.

Los datos proporcionados a la red en este caso serán tres ficheros de *tfrecords* de 68 pacientes cada uno, un fichero de 23 pacientes para la validación y por último para la parte de obtención de resultados en el test utilizaremos 57 pacientes distintos a los anteriores bloques de datos. Eso quiere decir la longitud total del entrenamiento será de 6120 cortes, la longitud total de validación será de 690 cortes y la longitud total de test es de 1710 cortes.

Respecto al número de *batches* utilizados para alimentar la red siguiendo las recomendaciones proporcionadas por los tutores, se utilizará una cantidad acorde al máximo que nos permita el uso de la GPU, en mi caso con el equipo del que dispongo, el número de *batch* tanto para el entrenamiento como para la validación será 5.

Centrándonos en detalles más técnicos una vez configurados los parámetros más ajenos a las capas de red, es necesario centrarnos en las propias capas, los cuales son los siguientes:

Lo primero de todo será decidir qué tipo de funciones de activación vamos a utilizar como hemos visto en el bloque 3, las cuales son la función *ReLU*, la función sigmoidea y la función tangente hiperbólica, siguiendo las indicaciones del documento (Ronneberger et al., 2015) los autores recomienda el uso de la función *ReLU* no obstante se realizarán pruebas con el uso de las otras dos funciones, el uso de la función *ReLU* es tan popular en las redes convolucionales debido a como se indica en el siguiente documento (Krizhevsky, A et al ,2012) las redes convolucionales trabajan mucho mejor ya que su uso disminuye el tiempo de entrenamiento en modelos más profundos y con bases de datos con un número importante de elementos, comparado con el uso de las otras funciones que tardan mucho más tiempo.

- Los valores del tamaño del *kernel* en nuestras capas convolucionales permanecerán fijo esto se debe a que el tamaño usado más comúnmente es 3x3

para las capas intermedias y 1x1 en la primera y última capa. Los valores están tomados de la figura 20

- La inicialización de los pesos y de los sesgos, en el caso de los sesgos utilizaremos una inicialización de un vector de ceros ya que es la forma natural de inicializarlos, en el caso de los pesos utilizaremos la inicialización con una distribución normal en la mayoría de los casos, en algún caso se procederá a la no inicialización de los mismos.

Y respecto a las técnicas de regularización aplicadas:

- En el caso de las capas de regularización de dropout habrá diferentes valores pero principalmente fijaremos 0,2 y 0,8. En algunos habrá pruebas lo realizaremos con un valor 0,5 para ver la diferencia entre los 3 casos. No se realizaran pruebas con los valores máximos y mínimos debido a que si seguimos la explicación de la API de TensorFlow sobre dicha capa, veremos que para el valor 0 es como si no hubiéramos aplicado la técnica de regularización y en el caso de 1 la salida sería nula.

- Por último, el uso del *Data Augmentation* en cierto número de pruebas esto se debe a que si realizamos diferentes procesados la aplicación no va a recibir siempre los cortes de la misma forma en la entrada. La explicación de los procesados a los cortes han sido explicados en el bloque 5 pero habrá pruebas donde no se haya aplicado un procesado, el hecho de usar un aumento de brillo en las imágenes puede producir un desbalanceo de clases mayor del que ya hay.

- El uso del *early stopping* con la reducción de la tasa de aprendizaje con el uso del *Reduce Learning on Plateau*.

A continuación en la Tabla 3 se muestran las pruebas realizadas con los diferentes parámetros usados, hay que indicar que a lo largo de las pruebas el número máximo de *epochs* o iteraciones durante el entrenamiento y la validación ha sido de 50 pero normalmente no llegamos a esa cifra debido al algoritmo de *Reduce Learning Rate on Plateu* ya que se alcanza una situación de *overfitting* mucho antes.

En la tabla 13 que podemos ver en el anexo, se muestran una serie de pruebas iniciales con el fin de aprender el funcionamiento de los hiperparámetros que podemos utilizar. Constatando que la función de activación *Tanh* o tangente hiperbólica no es efectiva en este tipo de redes, además el uso de dropouts tan agresivos demuestra que pueden verse alterados los resultados; siguiendo la recomendación del tutor en las pruebas sucesivas se ha disminuido el valor de la tasa de aprendizaje para prevenir el sobre entrenamiento.

Tras realizar las pruebas anteriores se puede observar un patrón de tiempo que de media en realizar una prueba con este modelo de red puede rondar perfectamente 24 horas, ya que los tiempos que se manejan van desde las 13 horas para los casos más favorables o hasta las 30 horas en los casos que más tardíos, esto principalmente se produce por el gran volumen de imágenes de las que disponemos y la profundidad de nuestro modelo usado.

Para poder valorar nuestro segundo modelo al igual que en el caso anterior es necesario utilizar un set que no se haya utilizado en la primera parte de las pruebas (entrenamiento y validación).

Una vez realizado todo, las tres partes con los diferentes sets de datos para cada una de las pruebas es necesario utilizar las medidas estadísticas explicadas anteriormente en este bloque, además para poder realizar una buena práctica a la hora de comparar los resultados, utilizaremos como tablas las que vienen utilizadas por los autores de los (B. H. Menze *et al, 2015) y* (M. Havei et al, 2017).

La primera tabla se obtiene con la herramienta scikit learn y más concretamente con la función *sklearn.metrics.classification_report* la cual nos genera un resumen de texto con la precisión, recall y F1 score para cada clase, para ello sus valores de entrada serán las máscaras de los cortes reales (y_true) y las máscaras de los cortes predichos (y_pred).

La segunda tabla la obtenemos nosotros calculando la *sensitivity, specifity* y *dice* de cada una de las regiones mencionadas en la base de datos BraTS, al igual que en la anterior tabla, aquí veremos realmente cómo se comporta nuestro modelo y si nuestros resultados son favorables o desfavorables.

Por último y no menos importante tenemos también las gráficas del desempeño de los entrenamientos al igual que las matrices de confusión globales del test.

Con el objetivo de poder profundizar y justificar mejor la elección de los parámetros se han realizado más pruebas bajo las indicaciones del tutor.

Arquitectura	Dataset	Función de activación	Valor Dropout	Paciencia activada	Data Augmentation	Learning Rate
Unet Modificada 8	Completo y Normalizado	ReLU	0,1	Si	Si y con Brillo	1,00E-04
Unet Modificada 9	Completo y Normalizado	ReLU	0,2	Si	Si y con Brillo	1,00E-04
Unet Modificada 10	Completo y Normalizado	ReLU	0,2	Si	Si y con Brillo	1,00E-05
Unet Modificada 11	Completo y Normalizado	ReLU	0,2	Si	Si y sin Brillo	1,00E-05
Unet Modificada 12	Completo y Normalizado	ReLU	0,5	Si	Si y sin Brillo	1,00E-05

Tabla 3 Características de las pruebas del modelo Unet.

En las pruebas se ha mantenido la función de activación ReLU, por ser la función de activación en redes convolucionales. El *dropout* utilizado en las pruebas va a estar entre 0,1 y 0,5 debido a que la aplicación de un dropout tan agresivo puede llegar a corromper los resultados obtenidos.

PRUEBAS	CLASES											
	Clase 0			Clase 1			Clase 2			Clase 3		
	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice
Unet Modificada 8	0,98	1	0,99	0,29	0,44	0,35	0,41	0,08	0,13	0,45	0,01	0,01
Unet Modificada 9	0,99	1	0,99	0,55	0,54	0,55	0,64	0,49	0,56	0,69	0,46	0,55
Unet Modificada 10	0,99	0,99	0,99	0,31	0,29	0,3	0,41	0,32	0,36	0,46	0,1	0,16
Unet Modificada 11	0,99	0,99	0,99	0,4	0,34	0,37	0,69	0,67	0,68	0,53	0,67	0,59
Unet Modificada 12	0,99	1	1	0,66	0,57	0,61	0,79	0,74	0,76	0,76	0,78	0,77

Tabla 4 Resultados de las regiones para las pruebas realizadas en el modelo Unet modificado.

PRUEBAS	REGIONES								
	Complete -WT			Core -TC			Enhancing - ET		
	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice
Unet Modificada 8	0,2	0,99	0,25	0,31	0,99	0,3	0	0,99	0
Unet Modificada 9	0,5	0,9	0,55	0,51	0,99	0,55	0,46	0,99	0,55
Unet Modificada 10	0,28	0,99	0,31	0,25	0,99	0,28	0,1	0,99	0,16
Unet Modificada 11	0,6	0,99	0,6	0,51	0,99	0,49	0,67	0,99	0,58
Unet Modificada 12	0,72	0,99	0,74	0,7	0,99	0,71	0,78	0,99	0,77

Tabla 5 Resultados de las métricas de las clases para las pruebas realizadas en el modelo Unet Modificado

En las Tablas 4 y 5 al igual que en las pruebas del anexo, observamos los resultados obtenidos para las clases y para las regiones. La última prueba realizada con un *dropout* de 0.5 y un *Learning Rate* de 10e-5 arroja los mejores resultados.

Esto indica que las funciones ReLU para nuestra problemática funcionan correctamente y la aplicación de un *Learning Rate* más pequeño hace que converja más lentamente. Provocando que nuestras pruebas aumenten nuestro tiempo máquina mucho más del que ya teníamos.

El *Data Augmentation* provoca que los resultados sean muy variables debido a que las modificaciones al ser aleatorias generan entradas diferentes en cada test. Las pruebas sin brillo en general arrojan mejores resultados, debido a que no produce tanto desequilibrio entre clases.

En las siguientes graficas se plasman los resultados del entrenamiento:

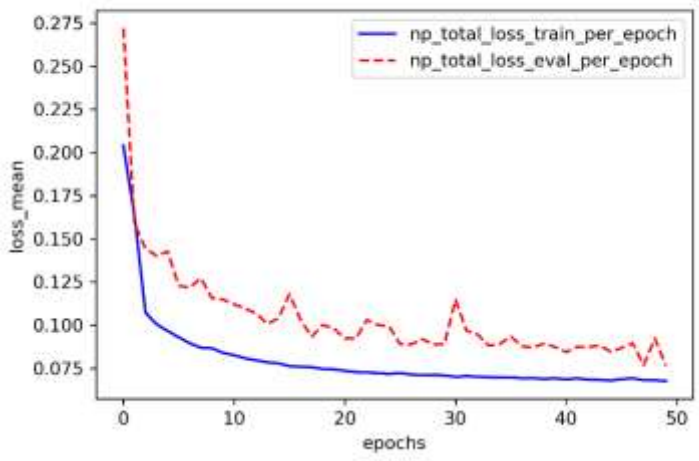

Figura 29 Progreso del entrenamiento del modelo U-net modificado

En la gráfica del entrenamiento podemos ver como se reduce el valor de la perdida media a lo largo de la validación como del entrenamiento, en este caso se observa como se ha completado el entrenamiento de una forma satisfactoria, nada que ver con las Figura 28 mostradas en el primer modelo donde no se producía ningún tipo de cambio en los valores, además podemos ver ciertos picos donde se produce un aumento del valor que en algunos casos se debe al sobre entrenamiento (*overfitting*).

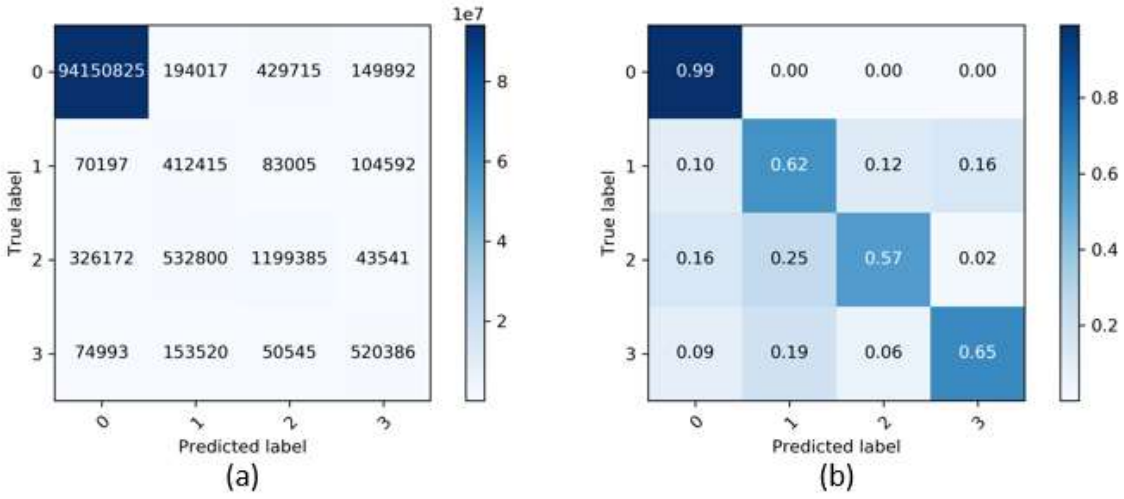

Figura 30 Matrices de confusión; (a) Valores de los pixeles clasificados, (b) Valores normalizados

En la Figura 30 se muestran las matrices de confusión de los datos obtenidos durante el proceso de test, con el que obtenemos las métricas de las clases, en el caso de la figura (a) los valores representan cada uno de los pixeles clasificados a lo largo del test. La figura (b) muestra los valores normalizados de la figura (a).

Mostramos los cortes calculados por nuestro modelo gráficamente:

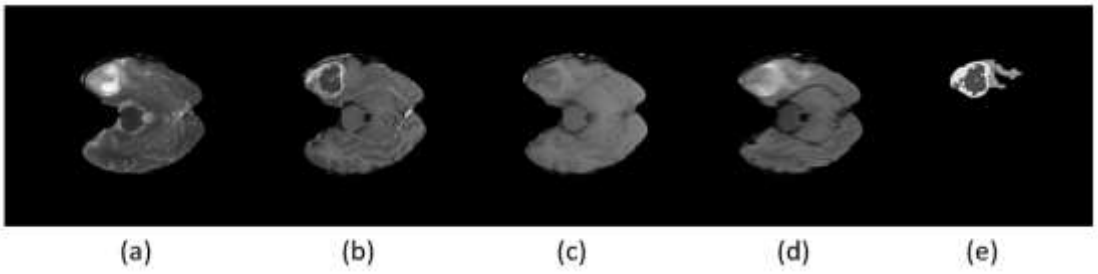

Figura 31 Ejemplo de corte para el test de la prueba 3; (a) Secuencia T2;(b) Secuencia T1c;(c) Secuencia T1;(d) Secencia Flair;(e) Máscara Segmentada

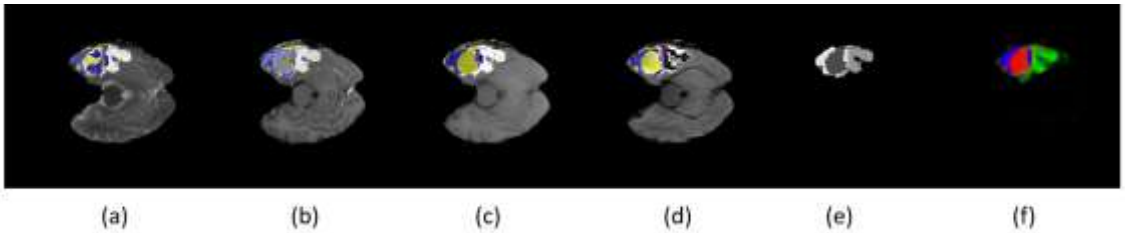

Figura 32 Resultados del test de la prueba 3 donde las secuencias tienen superpuestas la máscara generada; (a) Secuencia T2 ;(b) Secuencia T1c;(c) Secuencia T1;(d) Secuencia Flair;(e) Máscara Predicha; (f) Máscara Predicha a color

Este ejemplo es uno de los 1710 cortes calculados para cada una de las pruebas realizadas, en este caso es el ejemplo de la mejor prueba realizada con el modelo u-net modificada.

Modelo U-net modificado con capas de pooling.

En este modelo se realizan pruebas similares al modelo anterior, manteniendo la mayor parte de los parámetros sin modificar, dejando las capas de dropout con el mismo valor 0.5 para todas las pruebas, modificando las funciones de activación de las capas convolucionales y modificando las funciones de procesado de las imágenes a la entrada del modelo de red.

Arquitectura	Función de activación	Data Augmentation
Unet Modificada con capas de pool 1	ReLU	Si y sin Brillo
Unet Modificada con capas de pool 2	ReLU	Si y con Brillo
Unet Modificada con capas de pool 3	ReLU	No
Unet Modificada con capas de pool 4	Tangente Hiperbólica	No
Unet Modificada con capas de pool 5	Tangente Hiperbólica	Si y con Brillo
Unet Modificada con capas de pool 6	Tangente Hiperbólica	Si y sin Brillo

Tabla 6 Pruebas realizadas con el modelo U-net modificado con capas de pooling

Analizando las pruebas de la tabla 6 como en el modelo anterior, los resultados obtenidos son los siguientes.

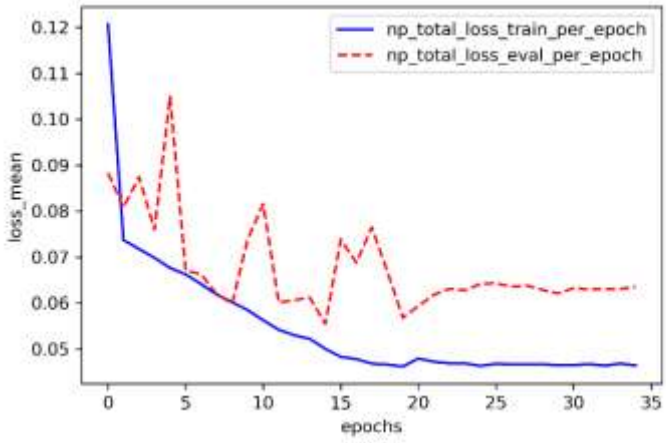

Figura 33 Progreso del entrenamiento del modelo U-net modificado con capas de pool

En la figura 33, podemos observar cómo se realiza el entrenamiento de una forma correcta y normal, a pesar de que se produzcan fluctuaciones y haya demasiados picos, en líneas generales se observa como el valor de la perdida en el caso del evaluación va de la mano junto a la línea del entrenamiento, esto indica que a lo largo de las iteraciones realizadas su número va disminuyendo hasta obtener una línea constante en la mayor parte de las pruebas realizadas.

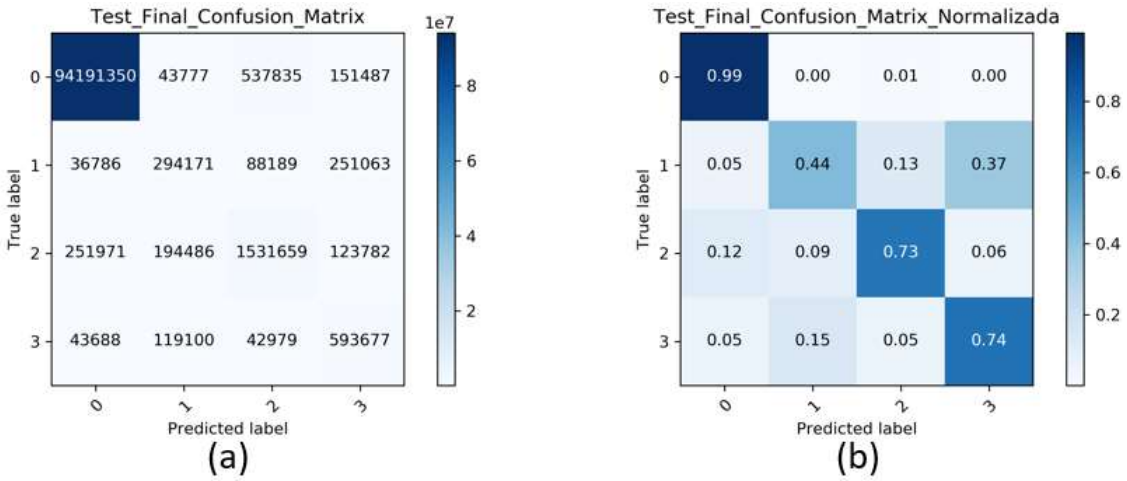

Figura 34 Matrices de confusión; (a) Valores de los pixeles clasificados, (b) Valores normalizados

Los valores observados en la diagonal principal de la figura 32 (b) muestran los valores de la sensibilidad o el *recall* para cada una de los clases, en principio son resultados muy favorables además de que mejoran los resultados de la figura 28 (b), en ambos casos las matrices utilizadas corresponden a las mejores pruebas.

Aunque perdamos un 18 % respecto al primero modelo para la clase 1 en la sensibilidad, aumentamos los valores en 16 % para la clase 2 y 9 % para la clase 3, en general mejores porcentajes que en el modelo anterior.

Al igual que en el anterior modelo presentamos las tablas de resultados de los porcentajes obtenidos para las clases y las regiones como podemos ver a continuación:

	CLASES											
PRUEBAS	Clase 0			Clase 1			Clase 2			Clase 3		
	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice
Unet Modificada 1	1	0,99	0,99	0,45	0,44	0,45	0,7	0,73	0,71	0,53	0,74	0,62
Unet Modificada 2	0,99	0,99	0,99	0,47	0,46	0,47	0,54	0,54	0,54	0,5	0,62	0,55
Unet Modificada 3	1	0,99	0,99	0,3	0,64	0,41	0,66	0,6	0,63	0,56	0,4	0,47
Unet Modificada 4	1	0,99	0,99	0,38	0,52	0,44	0,7	0,7	0,7	0,55	0,61	0,58
Unet Modificada 5	0,99	0,99	0,99	0,39	0,55	0,45	0,54	0,34	0,42	0,44	0,34	0,38
Unet Modificada 6	1	0,99	0,99	0,32	0,45	0,38	0,71	0,62	0,66	0,51	0,74	0,6

Tabla 7 Resultados de las pruebas para las clases.

PRUEBAS	REGIONES								
	Complete -WT			Core -TC			Enhancing - ET		
	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice
Unet Modificada 1	0,68	0,99	0,64	0,6	0,99	0,55	0,74	0,99	0,62
Unet Modificada 2	0,51	0,99	0,51	0,49	0,99	0,49	0,62	0,99	0,55
Unet Modificada 3	0,56	0,99	0,54	0,51	0,99	0,43	0,4	0,99	0,47
Unet Modificada 4	0,65	0,99	0,62	0,57	0,99	0,51	0,61	0,99	0,57
Unet Modificada 5	0,43	0,99	0,43	0,5	0,99	0,44	0,33	0,99	0,38
Unet Modificada 6	0,61	0,99	0,58	0,61	0,99	0,5	0,74	0,99	0,6

Tabla 8 Resultados de las regiones evaluadas durante las pruebas de test

Podemos ver en la Tabla 7 como los resultados de las clases para los valores de precisión varían entre 30 y un 40 % para la clase 1 mientras que para las clases 2 y 3 aumentan dichos valores rondan entre un 50 y 70 % para la clase 2, y un 45 y 55 % para la clase 3.

Si observamos el *recall* los valores aumentan algo más para las clases, esto se debe a que la sensibilidad obtenida suele tener mayor porcentaje que los valores predichos positivos. Los valores del dice son favorables también, ya que cumple su función aunque la clase 1 sea la que peores resultados obtiene de forma general durante las pruebas.

En la Tabla 8 observamos los resultados calculados de las regiones donde la especificad siempre va a ser alta, se debe a que nuestros valores de los verdaderos negativos son mucho mayores respecto a los valores que obtenemos de los falsos positivos, esto produce que la fracción que lo calcule se aproxime a 1 en la mayoría de los casos.

Si nos centramos en la sensibilidad de cada región podemos observar que en el caso de la prueba 1 supera el 60 % de acierto para los casos favorables obtenidos de las métricas de los verdaderos positivos. Para el resto de pruebas disminuye en la mayoría de casos pero aun así se obtienen resultados que superan el 50 %.

Tras analizar los resultados de las diferentes pruebas, para este modelo la mejor prueba obtenida tiene una función de activación (*ReLU*) diferente a la del modelo anterior (*tanH*).

En cuanto al procesado de las imágenes a la entrada de la red, observamos que las pruebas realizadas con brillo y sin brillo muestran notables diferencias, al no aplicar la modificación del brillo en las imágenes, y si el resto de modificaciones hace que se obtengan mejores resultados.

Como hemos realizado en el modelo anterior realizaremos una segunda tanda de pruebas, en este caso únicamente trabajando con la función de activación *ReLU*, para ver si modificando los hiperparámetros podemos conseguir mejores resultados que con las anteriores pruebas.

Arquitectura	Función de activación	Data Augmentation
Unet Modificada con capas de pool 7	ReLU	Si y sin Brillo
Unet Modificada con capas de pool 8	ReLU	Si y con Brillo
Unet Modificada con capas de pool 9	ReLU	Si y sin Brillo
Unet Modificada con capas de pool 10	ReLU	Si y con Brillo

Tabla 9 Segunda tabla de pruebas para el modelo con capas de pool

Para estas 4 pruebas de esta segunda tanda han sido eliminados muchos otros intentos ya que la variación principal ha sido en la tasa de aprendizaje la cual hemos ido disminuyendo hasta un punto en el que no es capaz de mejorar los resultados sino que empeorarlos.

	REGIONES								
PRUEBAS	Complete -WT			Core -TC			Enhancing - ET		
	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice
Unet Max Pool 1	0,36	0,99	0,43	0,19	0,99	0,27	0,32	0,99	0,4
Unet Max Pool 2	0,167	0,99	0,24	0,2	0,99	0,27	0,2	0,99	0,04
Unet Max Pool 3	0,362	0,99	0,46	0,26	0,99	0,35	0,4	0,99	0,46
Unet Max Pool 4	0,19	0,99	0,26	0,1	0,99	0,1	0,1	0,99	0,1

Tabla 10 Resultados de las regiones evaluadas durante la prueba de red para la segunda tanda

	CLASES											
PRUEBAS	Clase 0			Clase 1			Clase 2			Clase 3		
	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice
Unet Max Pool 1	0,99	1	0,99	0,24	0,02	0,04	0,56	0,47	0,51	0,52	0,33	0,4
Unet Max Pool 2	0,98	1	0,99	0,42	0,26	0,32	0,41	0,12	0,19	0,52	0,02	0,04
Unet Max Pool 3	0,98	1	0,99	0,42	0,1	0,17	0,68	0,43	0,52	0,54	0,41	0,46
Unet Max Pool 4	0,98	1	0,99	0,44	0,06	0,11	0,43	0,28	0,34	0,51	0,05	0,09

Tabla 11 Resultados de las pruebas para las clases para la segunda tanda de pruebas.

En ambas tablas 10 y 11 observamos los resultados para esta segunda tanda de pruebas, el mayor cambio en estas pruebas ha sido el cómo hemos modificado los anteriores *Learning rates* hasta un máximo de 10e-6 donde ya las pruebas empezaban a mostrar peores resultados que las mostradas en la primera tanda para este modelo y se acentuaba con mayor medida a través de esta disminución, arrojando unos resultados donde los valores obtenidos para él *Dice* rondaban cero tanto para las clases como para las regiones calculadas.

Dependiendo de las modificaciones realizadas en la entrada de la red, los tiempos para realizar el entrenamiento varían desde las 10 horas hasta las 20 horas.

En las siguientes Figuras 36 y 37 observamos cómo es capaz de realizar de una forma correcta nuestro modelo la tarea de segmentar las imágenes, para ello utilizamos la mejor prueba mostrando uno de los cortes donde obtenemos las predicciones de la máscara generada a escala de grises y a color así como la superposición sobre las diferentes secuencias de los cortes originales.

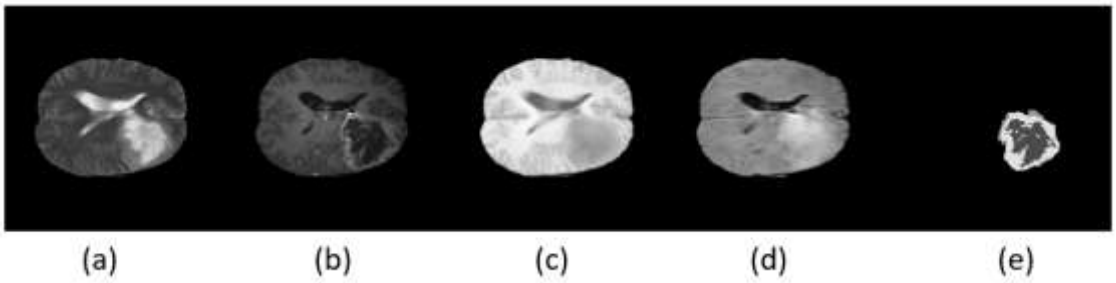

Figura 36 Ejemplo de corte para el test de la prueba 1; (a) Secuencia T2;(b) Secuencia T1c;(c) Secuencia T1;(d) Secuencia Flair;(e) Máscara Segmentada

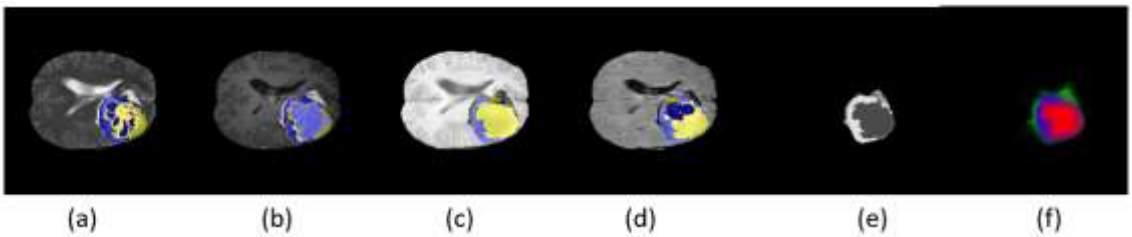

Figura 35 Resultados del test de la prueba 1 donde las secuencias tienen superpuestas la máscara generada; (a) Secuencia T2 ;(b) Secuencia T1c;(c) Secuencia T1;(d) Secuencia Flair;(e) Máscara Predicha; (f) Máscara Predicha a color

Analizando los resultados de los modelos testeados llegamos a varias conclusiones:

- Al añadir las capas de *pooling* en el tercer modelo, mejoramos notablemente los resultados obtenidos, no se obtienen ceros para ninguna de las pruebas.
- El uso de dropouts agresivos no es conveniente porque los resultados arrojados son muy diferentes a los obtenidos con dropouts comprendidos entre 0 y 0.5, siendo este rango el más conveniente.
- El uso del *data augmentation* a la entrada de la red es útil (porque el modelo no se espera que la forma de las imágenes siempre sea la misma) siendo recomendable pero no imprescindible.
- El uso de la función *ReLU* mejora los tiempos de las pruebas, en cambio el uso de la función tanH mejora los resultados obtenidos, por lo tanto descartamos el uso de la función sigmoide.
- El uso de la técnica del *Early Stopping* reduce los tiempos de entrenamiento, porque realiza menos iteraciones de las que marcamos como máximo.
- Comparando los mejores resultados de los modelos con la Tabla 12 de clasificación de la fase de entrenamiento de la competición MICCAI BRATS 2017 de (Spyridon Bakas, 2019), observamos que los resultados obtenidos en nuestras pruebas superan la media de dicha competición, podríamos decir que nuestros modelos son funcionales y operativos.

TEAMS	REGIONES								
	Complete -WT			Core -TC			Enhancing - ET		
	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice
Alexander	0.76693	0.85744	0.53275	0.0	1.0	0.0	0.08696	1.0	0.08696
Ashi	0.67758	0.98968	0.68882	0.63532	0.98482	0.4878	0.63773	0.98927	0.45598
BCVUniandes	0.84205	0.99598	0.8688	0.67272	0.99634	0.67842	0.70653	0.99824	0.66664
Bern	0.83597	0.99718	0.88029	0.78077	0.99616	0.75508	0.77749	0.99697	0.7099
BIGS2	0.89096	0.98695	0.8268	0.72698	0.99353	0.67991	0.76504	0.996	0.64611
biomedia1	0.89478	0.99538	0.90088	0.76166	0.99849	0.79723	0.78287	0.99832	0.73755
biomedia1_BHN	0.91667	0.97694	0.75198	0.94828	0.94623	0.41223	0.08696	1.0	0.08696
BMSSA	0.83389	0.99552	0.86537	0.69312	0.99804	0.73782	0.72858	0.99848	0.70081
BRATZZ27	0.85664	0.99607	0.88001	0.62678	0.9987	0.66693	0.73688	0.99839	0.69378
BrICLab	0.89091	0.995	0.89031	0.69655	0.99805	0.73242	0.74853	0.99804	0.7104
CamMIA	0.77979	0.99671	0.82903	0.65186	0.99835	0.69242	0.67395	0.9985	0.65655
CIAN	0.89253	0.99477	0.89322	0.68432	0.99879	0.73489	0.73548	0.99834	0.71122
CISA	0.85482	0.99465	0.87337	0.68609	0.99818	0.74126	0.72905	0.99855	0.71469
CMR	0.81119	0.9968	0.85696	0.65972	0.99895	0.71558	0.68677	0.99899	0.70555
CNEA	0.86684	0.99274	0.85898	0.72805	0.99558	0.69436	0.75625	0.99746	0.65747
CUMED	0.90128	0.99193	0.87606	0.67969	0.99855	0.70983	0.70726	0.99829	0.67359
CVLab	0.6789	0.97931	0.67997	0.65862	0.9837	0.57718	0.54439	0.99317	0.44131
Drcubic	0.99818	0.00522	0.11883	0.99777	0.0051	0.05747	0.99846	0.0073	0.02378
Dundee	0.86014	0.99537	0.87971	0.69805	0.99824	0.73442	0.75735	0.99832	0.73425
fly100	0.79256	0.9923	0.80788	0.66979	0.99655	0.67262	0.05833	0.99939	0.07899
HPI-Ultimate	0.75233	0.99623	0.80757	0.5575	0.99813	0.61156	0.6192	0.99847	0.61552
inpm	0.90887	0.99389	0.89982	0.80363	0.99733	0.80845	0.83613	0.99743	0.77233
justdoit	0.65922	0.98985	0.69808	0.54007	0.99289	0.52794	0.51156	0.99718	0.5092
LoVE	0.82565	0.99586	0.86303	0.7226	0.9985	0.77526	0.57412	0.99923	0.65762
MBI	0.83158	0.99243	0.82356	0.66851	0.9939	0.62565	0.59523	0.99876	0.57496
MIA-Uminho	0.88321	0.99332	0.88051	0.76206	0.99712	0.76562	0.75256	0.99817	0.69097
MIC_DKFZ	0.90186	0.99572	0.90269	0.78588	0.99893	0.81935	0.80289	0.99829	0.77555
MiMSeg_team	0.89214	0.99373	0.88354	0.68185	0.99618	0.64192	0.77769	0.99712	0.66071
MIRL	0.84157	0.99249	0.83428	0.70188	0.99682	0.7192	0.7016	0.99838	0.68921
MISPL	0.81899	0.98157	0.77806	0.72679	0.98867	0.63573	0.73837	0.99434	0.59603
Mountain	0.83969	0.93963	0.55988	0.7501	0.96484	0.41464	0.70374	0.98754	0.43199
naomi.fridman	0.77343	0.96906	0.68844	0.55922	0.99163	0.5501	0.3975	0.99851	0.44079
neuro.ml	0.90449	0.99106	0.84608	0.70857	0.99768	0.72653	0.75201	0.99806	0.67433
NIC-VICOROB	0.74293	0.99221	0.77745	0.48888	0.99889	0.54643	0.58724	0.99876	0.59337
niftynet	0.89308	0.99357	0.88345	0.78019	0.99721	0.75606	0.779	0.99815	0.7246
NPU	0.83458	0.99309	0.85306	0.6768	0.99605	0.70056	0.72148	0.99733	0.65463
NUS_MPR	0.81872	0.99657	0.86413	0.68961	0.99823	0.74577	0.71069	0.99844	0.70331
OnePiece	0.88656	0.99347	0.88413	0.68731	0.99859	0.71016	0.85315	0.99705	0.74795
PADAS	0.92492	0.98014	0.83028	0.57165	0.99529	0.59029	0.53005	0.99607	0.47785
pvg	0.8941	0.99498	0.88854	0.75583	0.99795	0.77105	0.76763	0.99804	0.73527
QTIM	0.87932	0.99459	0.88265	0.70201	0.99774	0.73005	0.72766	0.99857	0.7325
ROB	0.86292	0.99266	0.86075	0.76867	0.98658	0.59582	0.71228	0.99729	0.68593
Rocky	0.87375	0.99066	0.85476	0.71359	0.99584	0.70329	0.75735	0.99732	0.67763
SaraS	0.92653	0.99097	0.87736	0.78675	0.99458	0.71289	0.79816	0.99731	0.67815
SCUT_EE_	0.87642	0.99561	0.88647	0.7599	0.99785	0.78966	0.78157	0.99819	0.75944
SegLZ	0.88741	0.99499	0.89507	0.75509	0.99838	0.78396	0.72909	0.99008	0.56965
SJTU	0.90482	0.99426	0.89569	0.74554	0.99707	0.7512	0.77118	0.99812	0.73438
STH	0.83468	0.99669	0.87346	0.70275	0.99885	0.7632	0.69404	0.99888	0.69072
stryker	0.89757	0.99543	0.90076	0.77014	0.99776	0.78282	0.78954	0.99796	0.75549
Succubus	0.91033	0.99936	0.94381	0.71673	0.99944	0.77338	0.7834	0.99856	0.68328
tkuan	0.8909	0.99375	0.88917	0.74135	0.998	0.78167	0.75039	0.99869	0.76501
UCCS	0.91858	0.97277	0.78331	0.73782	0.99267	0.68537	0.65688	0.99664	0.56669
UCLM_UBERN	0.90038	0.9951	0.90135	0.75961	0.99814	0.79051	0.8006	0.99755	0.74922
UCL-TIG	0.91525	0.99472	0.90499	0.82159	0.99796	0.83779	0.77134	0.99852	0.78585
UPC_DLMI	0.85054	0.99617	0.87979	0.6097	0.99802	0.63415	0.71288	0.99849	0.71178
VisionLab	0.78073	0.99134	0.79574	0.57917	0.99636	0.60478	0.67314	0.9979	0.59704
whatapain	0.88671	0.99431	0.89035	0.80591	0.99445	0.7876	0.74557	0.99808	0.74343
xfeng	0.90868	0.99405	0.89217	0.76075	0.99874	0.79912	0.79396	0.99807	0.75114
YZZ	0.0	0.99946	0.0	0.99998	1340485	0.0	0.0	0.98563	0.0
ZejuLi	0.82785	0.99414	0.8449	0.71193	0.99464	0.69025	0.64864	0.99708	0.5897
Zhao	0.93982	0.98718	0.87192	0.8018	0.99685	0.78939	0.81606	0.99682	0.75925
Zhouch	0.90328	0.99531	0.90386	0.84105	0.99578	0.82792	0.80141	0.99799	0.77841
ZLM	0.73592	0.99396	0.78236	0.56569	0.99687	0.59744	0.66362	0.99812	0.59468

Tabla 12 Tabla de resultados de la competición BraTS '17

BLOQUE 7: CONCLUSIONES Y LINEAS FUTURAS.

7. CONCLUSIONES FINALES.

7.1. CONCLUSIONES

Este proyecto ha buscado el desarrollo de una aplicación para segmentación de imágenes biomédicas mediante la exploración de las redes neuronales convolucionales, acorde a una base de datos correctamente segmentada de tumores cerebrales.

El estudio del problema de segmentación de imágenes es un campo que avanza muy rápido por lo que la problemática está estudiada y desarrollada mediante diferentes técnicas de *Deep Learning* y del pre procesamiento de los datos.

Para la ejecución de la aplicación precisamos de un hardware mínimo necesario, los principales componentes serían una tarjeta gráfica NVIDIA de altas prestaciones, una memoria RAM amplia para poder cargar más datos, un procesador lo más rápido posible, por último y no menos importante un sistema de disipación de alto rendimiento.

La tarea del preprocesado de los datos en muchos casos han sido tediosas debido principalmente al gran desconocimiento sobre este campo y la dificultad a la hora de trabajar con un nuevo tipo de formato de ficheros (*tfrecords*) usado a lo largo de la aplicación.

La librería *Tensorflow* está en desarrollo y se actualiza con cierta frecuencia, esto nos ha provocado en la mayoría de los casos dificultades en la programación. En teoría este tipo de librerías son capaces de resolver cualquier problema, pero tampoco ayudan en el manejo y aprendizaje por lo que no son para nada fáciles de usar, adaptar ciertos errores nos ha retrasado en el desarrollo de la aplicación.

En la implementación de las redes convoluciones observamos que no hay una única solución para el problema planteado. A lo largo del trabajo hemos utilizado 3 modelos con una configuración de capas y parámetros diferentes, pero analizando en profundidad, los modelos siendo muy semejantes utilizando los mismos datos, pueden darnos resultados diferentes.

La conclusión final de este trabajo es que la realización de una aplicación de segmentación de imágenes biomédicas permite ver el gran potencial de las redes convolucionales. Los resultados de nuestra aplicación demuestran un rendimiento, precisión o tasa de acierto cercanas a los resultados reflejados en BRATS ‘17 sobre el mismos dataset y aplicando herramientas y técnicas de *Deep Learning* diferentes cumpliendo el principal objetivo de este proyecto.

7.2. MEJORAS DE LA APLICACIÓN PARA EL FUTURO.

Tras la realización de los principales objetivos de este Trabajo Final de Grado y de cara a mejorar los resultados obtenidos proponemos ciertas mejoras.

El uso de la herramienta *TensorFlow* adaptada a la versión 2.0 junto con la API *Keras*, esta permitiría la implementación de una forma más sencilla y efectiva de los diferentes modelos de red así como todo el pre procesamiento de los datos y su carga.

Otra alternativa sería probar con las diferentes versiones de BraTS o el poder utilizar en nuestra aplicación otras bases de datos similares a la utilizada en este trabajo.

Disponer de una mejora de hardware debido a que esto nos permitiría usar más cortes por paciente y utilizar más técnicas de *data augmentation* para generar nuevos cortes, implicando una disminución de los tiempos en las pruebas de entrenamiento.

Perfeccionaríamos los resultados con distintos modelos de redes convolucionales con distinto planteamiento al nuestro pero buscando el mismo objetivo.

Finalmente una posible vía de desarrollo a partir de este trabajo sería el uso de una base de datos propia, con el uso de mejores modelos de red para esa base de datos y cuyo fin la implementación de una aplicación real, en el campo de la biomedicina.

8. ANEXO

Pruebas Iniciales

Al comenzar el estudio las primeras pruebas fueron descartadas por el uso de diferentes valores recomendados para los hiperparámetros y el uso incorrecto de las funciones de activación para las redes neuronales convolucionales, además de no aplicar en la mayoría de las pruebas el *data augmentation*.

Dichas pruebas nos sirvieron para aprender el método más adecuado para la aproximación de este trabajo.

Las pruebas se llevaron a cabo en el modelo número dos estudiado en el bloque 5 obteniendo unos resultados insatisfactorios, orientando el estudio a los nuevos parámetros como la disminución del *Learning rate,*no sobrepasar el valor de 0.5 para las capas de dropout e incluir el uso del *data augmentation* en las pruebas posteriores.

Arquitectura	Dataset	Función de activación	Valor Dropout	Paciencia activada	Data Augmentation	Learning Rate
Unet Modificada 1	Completo y Normalizado	Tangente Hiperbólica	0,2	Si	No	0,0001
Unet Modificada 2	Completo y Normalizado	Tangente Hiperbólica	0,8	Si	No	0,0001
Unet Modificada 3	Completo y Normalizado	ReLU	0,8	Si	Si y con Brillo	0,0001
Unet Modificada 4	Completo y Normalizado	ReLU	0,2	Si	Si y con Brillo	0,0001
Unet Modificada 5	Un tercio y Normalizado	ReLU	0,5	No	No	0,0001
Unet Modificada 6	Completo y Normalizado	ReLU	0,8	No	No	0,0001
Unet Modificada 7	Completo y Normalizado	ReLU	0,2	No	No	0,0001

Tabla 13 Características de las pruebas realizadas con la arquitectura U-net modificada

	REGIONES								
PRUEBAS	Complete -WT			Core -TC			Enhancing - ET		
	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice	Sensitivity	Specifity	Dice
Unet Modificada 1	0,6	0,99	0,57	0,64	0,99	0,52	0,65	0,99	0,64
Unet Modificada 2	0,49	0,99	0,48	0,55	0,99	0,44	0,51	0,99	0,54
Unet Modificada 3	0,55	0,99	0,57	0,58	0,99	0,57	0,54	0,99	0,58
Unet Modificada 4	0,41	0,99	0,43	0,5	0,99	0,45	0,32	0,99	0,4
Unet Modificada 5	0,2	0,99	0,2	0,3	0,98	0,21	0,05	0,99	0,01
Unet Modificada 6	0,48	0,99	0,5	0,04	0,99	0,06	0	0,99	0,01
Unet Modificada 7	0,32	0,97	0,18	0,25	0,98	0,16	0,35	0,98	0,22

Tabla 14 Resultados de las regiones en las pruebas realizadas con el modelo U-net modificado.

	CLASES											
PRUEBAS	Clase 0			Clase 1			Clase 2			Clase 3		
	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice	Precision	Recall	Dice
Unet Modificada 1	1	0,99	0,99	0,32	0,62	0,42	0,68	0,57	0,62	0,64	0,65	0,64
Unet Modificada 2	0,99	0,99	0,99	0,26	0,59	0,36	0,65	0,44	0,53	0,57	0,51	0,54
Unet Modificada 3	0,99	0,99	0,99	0,54	0,59	0,56	0,63	0,51	0,56	0,63	0,54	0,58
Unet Modificada 4	0,99	0,99	0,99	0,4	0,55	0,46	0,52	0,31	0,39	0,56	0,32	0,4
Unet Modificada 5	0,99	0,99	0,99	0,16	0,6	0,25	0,47	0,13	0,21	0,18	0,05	0,08
Unet Modificada 6	0,99	0,99	0,99	0,27	0,08	0,12	0,53	0,75	0,63	0,74	0,01	0,02
Unet Modificada 7	0,99	0,94	0,96	0,06	0,13	0,08	0,13	0,37	0,2	0,17	0,35	0,23

Tabla 15 Métricas de las pruebas realizadas con el modelo U-net modificado.

9. BIBLIOGRAFIA:

- Agarwal, R. (2019). *Demystifying Object Detection and Instance Segmentation for Data Scientists*. Towards Data Science. Retrieved 20 February 2020, from https://towardsdatascience.com/a-hitchhikers-guide-to-object-detection-and-instance-segmentation-ac0146fe8e11.
- Aluminium, W. (2020). *Woxter Notebook Cooling Pad 2200 Aluminium - woxter*. Woxter.es. Retrieved 20 February 2020, from https://woxter.es/esp/es/otros/328-woxter-notebook-cooling-pad-2200-aluminium-8435089011883.html.
- *Anaconda Python/R Distribution - Free Download*. Anaconda. (2020). Retrieved 20 February 2020, from https://www.anaconda.com/distribution/.
- *ASUS ZenBook Pro UX550VD | Portátiles | ASUS España*. ASUS España. (2020). Retrieved 20 February 2020, from https://www.asus.com/es/Laptops/ASUS-ZenBook-Pro-UX550VD/Tech-Specs/.
- B. H. Menze *et al*. (2015), The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS), in *IEEE Transactions on Medical Imaging*, vol. 34, no. 10, pp. 1993-2024. Retrieved 21 February 2020, from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6975210
- *Características de Windows 10: Qué hay en Windows 10 | Microsoft*. Microsoft.com. (2020). Retrieved 20 February 2020, from https://www.microsoft.com/es-es/windows/features.
- Chollet, F. (2017). *Deep Learning With Python* (1st ed., pp. 4-19). Manning Publications.
- *CUDA Zone*. NVIDIA Developer. (2020). Retrieved 20 February 2020, from https://developer.nvidia.com/cuda-zone.
- Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochasticoptimization.The Journal of Machine Learning Research, 12:2121–2159, 2011.
- *General Python FAQ — Python 3.8.2rc2 documentation*. Docs.python.org. (2020). Retrieved 20 February 2020, from https://docs.python.org/3/faq/general.html#what-is-python.
- Goodfellow, I., Bengio, Y. y Courville A. (2016) Deep Learning. Cambridge, MA (EEUU): MIT Press. Retrieved 20 February 2020 from http://www.deeplearningbook.org/
- Grosse, R. (2018). *Linear Clasifiers*. Department of Computer Science, University of Toronto. Retrieved 20 February 2020, from https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/readings/L03%20Linear%20Classifiers.pdf.
- Grosse, R. (2018). *Multilayer Perceptron*. Department of Computer Science, University of Toronto. Retrieved 20 February 2020, from

https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/readings/L05%20Multilayer%20Perceptrons.pdf
- Grosse, R. (2018). *Training a classifier*. Department of Computer Science, University of Toronto. Retrieved 20 February 2020, from https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/readings/L04%20Training%20a%20Classifier.pdf.
- Imagen por Resonancia Magnética (IRM). Retrieved 21 February 2020, from https://www.nibib.nih.gov/espanol/temas-cientificos/imagen-por-resonancia-magn%C3%A9tica-irm
- *Introducción — Tutorial de Python 3.6.3 documentation*. Docs.python.org.ar. (2020). Retrieved 20 February 2020, from http://docs.python.org.ar/tutorial/3/real-index.html.
- Kingma, D., & Ba, J. (2015). ADAM: A METHOD FORSTOCHASTICOPTIMIZATION. *Conference Paper At ICLR 2015*. Retrieved 20 February 2020, from https://arxiv.org/pdf/1412.6980.pdf.
- Krizhevsky, A., Sutskever, I., & E. Hinton, G. (2012). ImageNet Classification with Deep ConvolutionalNeural Networks. Retrieved from https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
- Lalkhen, A., & McCluskey, A. (2008). Clinical tests: sensitivity and specificity, *8*(6), Pages 221–223.
- Learn, S. *Confusion matrix — scikit-learn 0.22.1 documentation*. Scikit-learn.org. Retrieved 21 February 2020, from https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html.
- *Linux y GNU - Proyecto GNU - Free Software Foundation*. Gnu.org. Retrieved 20 February 2020, from https://www.gnu.org/gnu/linux-and-gnu.es.html.
- Murat, M. (2018). *Cross Entropy for Tensorflow*. Mustafa Murat ARAT. Retrieved 20 February 2020, from https://mmuratarat.github.io/2018-12-21/cross-entropy.
- M. Havei et al. (2017) , Brain tumor segmentation with Deep Neural Networks, Retrieved 21 February 2020, from https://arxiv.org/pdf/1505.03540.pdf
- Parmar, R. (2018). *Detection and Segmentation through ConvNets*. Towards Data Science. Retrieved 20 February 2020, from https://towardsdatascience.com/detection-and-segmentation-through-convnets-47aa42de27ea.
- *Procesador Intel® Core™ i7-7700HQ*. Intel. (2018). Retrieved 20 February 2020, from https://www.intel.es/content/www/es/es/products/processors/core/i7-processors/i7-7700hq.html.
- *Project Jupyter*. Jupyter.org. (2018). Retrieved 20 February 2020, from https://jupyter.org/.
- Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. Retrieved 20 February 2020, from https://arxiv.org/pdf/1505.04597.pdf
- Sasaki, Y. (2007). The truth of the F-measure.

- Spyridon Bakas, P. (2019). MICCAI-BraTS 2017 Leaderboard. Retrieved 29 February 2020, from https://www.cbica.upenn.edu/BraTS17/lboardTraining.html
- *Tarjeta gráfica GeForce GTX 1050*. NVIDIA. (2018). Retrieved 20 February 2020, from https://www.nvidia.com/es-la/geforce/products/10series/geforce-gtx-1050/.
- *TensorFlow*. TensorFlow. (2020). Retrieved 20 February 2020, from https://www.tensorflow.org/.
- *Tesla Piloto Automático*. Tesla Piloto Automático. (2020). Retrieved 15 April 2020, from https://www.tesla.com/es_ES/autopilot
- Tieleman, T. and Hinton, G. Lecture 6.5 - RMSProp, COURSERA: Neural Networks for Machine Learning.Technical report, 2012.