Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de un método de control de calidad de imágenes de RMN cerebral usando Deep learning

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alejandro Adolfo Kohan

Tutor: José Vicente Manjón

Curso 2018-2019

Resumen

En medicina, se utilizan imágenes Resonancia Magnética Nuclear para obtener información volumétrica de las distintas partes del cuerpo humano. Para obtener esta información se han desarrollado sistemas informáticos basados en inteligencia artificial (como las redes neuronales), que extraen esta información de forma automática a partir de grandes bases de datos de imagen etiquetadas. Uno de los problemas es que entre esas imágenes de entrenamiento en ocasiones hay algunas de mala calidad que no sirven para entrenar la red, y el gran número de datos hace que filtrarlas manualmente lleve a gastar una enorme cantidad de tiempo en esta tarea.

Este proyecto consiste en la creación de una herramienta que facilite y automatice este proceso de control de calidad de imagen.

La herramienta tiene que ser capaz de identificar en una gran cantidad de imágenes pasadas por el usuario, en este caso RMN cerebrales, las imágenes de buena calidad y de mala calidad separando unas de otras con la mayor cantidad de aciertos posible. Para la implementación se usan tecnologías de aprendizaje profundo (*Deep Learning*).

Finalmente, el problema se ha dividido en dos etapas, un primer acercamiento simplificado con imágenes de 2 dimensiones un segundo acercamiento más complejo usando las imágenes originales en 3 dimensiones.

Palabras clave: *Deep learning*, Keras, Tensorflow, Matlab, RMN, Redes neuronales convolucionales.

Abstract

In medicine, Nuclear Magnetic Resonance images is used to obtain volumetric information from different parts of the human body. To obtain this information, computer systems based on artificial intelligence (such as neuronal networks) have been developed, which automatically extract this information from large tagged image databases. One of the problems is that among those training images sometimes there are some of bad quality that are not useful to train the network, and the large number of data makes that filtering them manually leads to spend a huge amount of time in this task.

This project consists on the creation of a tool that facilitates and automates this image quality control process.

The tool must be able to identify in a big data base of images passed by the user, in this case brain MRI, the images of good quality and bad quality, separating them from each other with as many hits as possible. Deep Learning technologies are used for the implementation.

Finally, the problem has been divided into two stages, a first simplified approach with 2-dimensional images and a second more complex using the original 3-dimensional images.

Keywords: *Deep learning*, Keras, Tensorflow, Matlab, NMR, Convolutional neural networks

Tabla de contenido

TABLA DE FIGURAS **7**

1. INTRODUCCIÓN **8**

1.1. MOTIVACIÓN 8
1.2. OBJETIVOS 9
1.3. ESTRUCTURA DE LA MEMORIA 10

2. ESTADO DEL ARTE **11**

2.1. RESONANCIA MAGNÉTICA NUCLEAR 11
2.2. DEEP LEARNING 13
2.2.1. Redes neuronales *13*
2.3. REDES NEURONALES CONVOLUCIONALES 15
2.3.1. Convolución *15*
2.3.2. Batch Normalization *16*
2.3.3. Funciones de activación *17*
2.3.4. Función de pérdida *18*
2.3.5. Optimizador *19*
2.3.6. Pooling *20*
2.3.7. Dropout *21*
2.3.8. Flattening *21*
2.3.9. Dense layer *22*
2.4. ENTRENAMIENTO DEL MODELO 22
2.4.1. Forward-propagation *22*
2.4.2. Back-propagation *23*
2.4.3. K-Fold Cross validation *23*
2.4.4. Mixup *23*
2.5. TESTEAR MODELO 24
2.5.1. Bayesian Dropout *24*

3. TECNOLOGÍA, PROBLEMA E IMPLEMENTACIÓN **25**

3.1. TECNOLOGÍAS IMPLEMENTADAS 25
3.2. PROBLEMA 27
3.3. IMPLEMENTACIÓN 28
3.3.1. Creación del dataset *28*
3.3.2. Tratamiento de los datos *29*
3.3.3. Creación de un modelo de red neuronal convolucional *30*
3.3.4. Entrenamiento del modelo *31*
3.3.5. Testeo del modelo *34*

4. EXPERIMENTACIÓN **36**

4.1. TOMA DE CONTACTO EN DOS DIMENSIONES 36
4.1.1. Desarrollo de la experimentación *36*
4.1.2. Resultados más relevantes *38*
4.1.3. Conclusiones *39*
4.2. EXPERIMENTACIÓN EN TRES DIMENSIONES 39
4.2.1. Desarrollo de la experimentación *39*
4.2.2. Resultados finales *41*
4.2.3. Resultado final *42*

5. CONCLUSIONES **43**

5.1. RESULTADOS 43
5.2. VALORACIÓN PERSONAL 43
5.3. POSIBLES MEJORAS 43

6. BIBLIOGRAFÍA 44

Tabla de figuras

FIGURA 1: IRM CEREBRAL CORTES AXIAL, CORONAL Y SAGITAL RESPECTIVAMENTE. 11
FIGURA 2: IMÁGENES POTENCIADAS EN T1, T2 Y DP RESPETIVAMENTE. 12
FIGURA 3: IMÁGENES CON ERRORES QUE NO SE PUEDEN CORREGIR CON ALGORITMOS. 12
FIGURA 4: ESQUEMA DE UNA NEURONA BIOLÓGICA (IZQUIERDA) Y ESQUEMA DE UNA NEURONA COMPUTACIONAL (DERECHA). 13
FIGURA 5: ESQUEMA DE UNA RED NEURONAL. 14
FIGURA 6: RED NEURONAL CONVOLUCIONAL. 15
FIGURA 7: PRODUCTO ESCALAR DE UNA MATRIZ IMAGEN CON UN KERNEL Y SU RESULTADO DESPUÉS DE COMPLETAR EL PROCESO AL MOVER EL KERNEL POR TODA LA MATRIZ IMAGEN. 15
FIGURA 8: ALGORITMO DE NORMALIZACIÓN BATCH NORMALIZATION. 16
FIGURA 9: FUNCIÓN LINEAR (IZQUIERDA) Y FUNCIÓN NO-LINEAR SIGMOIDE (DERECHA). 17
FIGURA 10: REPRESENTACIÓN GRÁFICA DE LA FUNCIÓN RELU. 17
FIGURA 11: REPRESENTACIÓN MATEMÁTICA DE LA FUNCIÓN SOFTMAX (IZQUIERDA). EJEMPLO RESULTADO DE LA FUNCIÓN SOFTMAX PARA UNA ENTRADA DADA (DERECHA). 18
FIGURA 12: VALORES PARA FUNCIÓN DE PÉRDIDA DE ENTROPÍA CRUZADA. 19
FIGURA 13: EJEMPLO DE FUNCIONAMIENTO DE LA FUNCIÓN MAX-POOLING. 20
FIGURA 14: DROPOUT. MODELO DE RED NEURONAL ESTÁNDAR SIN DROPOUT (IZQUIERDA). MODELO DE RED NEURONAL CON DROPOUT, LAS NEURONAS ROJAS HAN SIDO DESACTIVADAS(DERECHA). ..21
FIGURA 15: EJEMPLO DE FLATTENING. 21
FIGURA 16: 5-FOLD CROSS VALIDATION. 23
FIGURA 17: EJEMPLO IMÁGENES DE BUENA CALIDAD (ARRIBA) Y MALA CALIDAD (ABAJO). 27
FIGURA 18: ALGUNOS ARCHIVOS IRM. 28
FIGURA 19: CÓDIGO DEL TRATAMIENTO DE NORMALIZACIÓN. 29
FIGURA 20: ESQUEMA DEL MODELO DE 3 DIMENSIONES. 31
FIGURA 21: EJEMPLO DE GENERADOR. MIXUP_GENERATOR_DIS UTILIZA EL MÉTODO MIXUP. 32
FIGURA 22: MÉTODOS FIT Y FIT_GENERATOR 32
FIGURA 23: DATOS EN PANTALLA DURANTE ENTRENAMIENTO 32
FIGURA 24: GRÁFICAS DE PRECISIÓN Y FUNCIÓN DE PÉRDIDAS 33
FIGURA 25: SALIDA POR PANTALLA DEL TESTEO. 34
FIGURA 26: EJEMPLO DE RESULTADOS CON OVERFITTING. 37
FIGURA 27: EJEMPLO DE UN NIVEL DE CONVOLUCIÓN. 37

1. Introducción

En el campo del aprendizaje automático se han hecho grandes avances con la aparición de los algoritmos de aprendizaje profundo (*Deep Learning*) que han dado un gran paso adelante en la capacidad de aprendizaje de las máquinas.

Para que estos algoritmos aprendan deben ser entrenados con una gran cantidad de datos para realizar distintas funciones. Ahora con el aprendizaje profundo se puede entrenar una red neuronal para poder reconocer rasgos de caras humanas, distintos tipos de animales, vehículos…

También se utilizan para reconocer distintos patrones en una imagen como puede ser los distintos hemisferios del cerebro, la materia gris, materia blanca, etc. Para que estos procesos de reconocimiento sean precisos se necesitan miles de imágenes para entrenar la red neuronal. Aquí surge el problema de que de entre estos miles de imágenes puede haber algunas de mala calidad ya que los RMN cerebrales son un proceso delicado que puede dar lugar a errores en las imágenes resultantes.

Este trabajo de final de grado presenta una solución a este problema mediante la creación de una red neuronal convolucional con el objetivo de distinguir RMN’s cerebrales de buena y mala calidad para hacer este proceso mucho más rápido.

1.1. Motivación

Los avances e hitos de la inteligencia artificial en los últimos años hacen que la *IA* se vuelva una realidad en nuestra vida diaria, tanto, que incluso se la utiliza sin ser consciente de ello y cada vez más empresas invierten en esta tecnología para buscar soluciones más eficientes para sus problemas.

Una de las motivaciones para elegir la carrera de Ingeniería informática, fue este campo e imaginar todo lo que podría llegar a pasar con el avance de este. Por esto mismo, la elección de la rama de Computación, para poder entender mejor esta tecnología y poder formar parte en estos avances en un futuro cercano.

Seleccionar este trabajo me ayuda a entender más como se trabaja y que tecnologías puedes utilizarse para trabajar en este campo y poder realizar un aprendizaje de manera autodidacta, con la ayuda de mi tutor y compañeros del laboratorio y videos de y tutoriales de la *Standford Computer Science* que aportan unas clases online gratuitas donde se explica desde lo más básico a lo más complejo sobre redes neuronales y convolucionales.

La motivación para seleccionar el tema de este trabajo aparte del interés por la inteligencia artificial es el problema actual que existe en muchos ámbitos profesionales tanto para la inteligencia artificial como para otros ámbitos donde la calidad de los datos es de una mucha importancia, esto se reduce al control de calidad que para muchos proyectos se traduce en una gran gasto de tiempo, infraestructura, personal y dinero que se podría ahorrar o invertirlo en otras partes

del proyecto que lo necesitaran más. Aquí es donde entra la utilización del *Deep Learning* en este trabajo, cuyo objetivo es que mediante el entrenamiento de una red neuronal convolucional se pueda encontrar una solución al problema del control de calidad automático y por tanto no necesitar a una persona para realizar el control de calidad y así ahorrar ese tiempo y dinero.

1.2. Objetivos

El objetivo de este trabajo es el desarrollo de una red convolucional que haga un control de calidad sobre RMN's cerebrales, siendo esta forma más rápida y eficiente que hacerla de forma manual.

El trabajo está dividido en dos grandes objetivos que se dividen en subobjetivos:

- **Entrenamiento con imágenes en 2 dimensiones.**
  - Conseguir el dataset necesario
  - Clasificar el dataset manualmente
  - Crear un modelo de red neuronal convolucional
  - Entrenar modelo
    - Encontrar la combinación de parámetros más eficiente
    - Guardar modelos más eficientes
  - Testear modelo
    - Seleccionar modelo más preciso
- **Entrenamiento con imágenes en 3 dimensiones**
  - Modificar modelo de 2 dimensiones
  - Entrenar modelo
    - Encontrar la combinación de parámetros más eficiente
    - Guardar modelos más eficientes
  - Testear modelo
    - Seleccionar modelo más preciso

El primer objetivo es un acercamiento simplificado al objetivo final, que ayuda a que el proceso de creación de la red neuronal convolucional, el entrenamiento, el testeo de esta sea más rápido y también a la comprensión de conceptos y a la creación de un método de trabajo.

Una vez finalizado el primer objetivo, se modifica el modelo elegido en este proceso, se lo adapta a un modelo en 3 dimensiones y se pasa a entrenar con los datos originales.

1.3. Estructura de la memoria

El presente trabajo está formado por 5 capítulos. A continuación, se resumen los temas tratados en cada capítulo:

- **Capítulo 1. Introducción**: En el primer capítulo se introduce el tema que se tratará durante el proyecto, así como la motivación que ha dado paso al desarrollo del trabajo y los objetivos a alcanzar.

- **Capítulo 2. Estado del arte**: En este capítulo se introduce de forma teórica al lector sobre la resonancia magnética cerebral, el *Deep Learning* y los procesos que forman parte de este.

- **Capítulo 3. Tecnologías, problema e implementación**: Aquí se explica las tecnologías que se han utilizado para el desarrollo del trabajo, se explica extensamente el problema a resolver y se explica cómo se han implementado los modelos de redes neuronales.

- **Capítulo 4. Experimentación**: Se exponen las pruebas realizadas durante la experimentación del trabajo y los problemas surgidos y como se solucionaron.

- **Capítulo 5. Conclusión**: En el último capitulo se aportan las conclusiones sobre el trabajo realizado y posibles mejoras a realizar en el futuro.

Se añade una bibliografía con las referencias consultadas durante la realización del trabajo.

2. Estado del arte

En este capítulo se repasan las tecnologías utilizadas y la actualidad sobre ellas, se hablará sobre la resonancia magnética nuclear (RMN), el control de calidad, las redes neuronales y las redes neuronales convolucionales, a las que se prestará mayor atención.

2.1. Resonancia magnética nuclear

La resonancia es un fenómeno físico en el que ciertos sistemas reaccionan cuando se le aplica energía a una frecuencia específica. Por ejemplo, cuando unas copas vibran y se aplica un sonido a la frecuencia correcta, estas pueden llegar a romperse.

El fenómeno de la resonancia magnética nuclear, o RMN, fue descubierto por Felix Bloch y Edward Purcell en 1946, dos físicos que por separado realizaron mediciones de campos magnéticos en el núcleo atómico. Los dos compartieron el premio nobel de Física en 1952. [12]

En líneas generales el RMN consiste en que determinados núcleos atómicos, como el hidrógeno, utilizado por Bloch y Purcell en sus respectivas mediciones, al someterlos a un campo magnético intenso estos pueden absorber energía en forma de radiofrecuencias de cierta frecuencia específica. Esta energía es devuelta cuando el sistema vuelve al estado de equilibrio como una señal eléctrica que es captada por una bobina de recepción o una antena que al analizar y procesar dicha información se forman imágenes de RMN (IRM). En la *Figura 1* se muestra un IRM cerebral. [13]

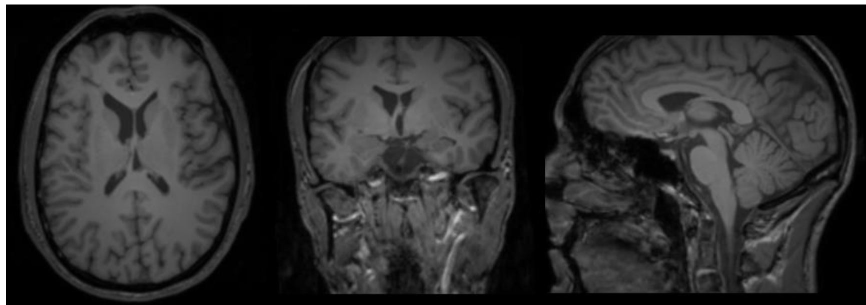


Figura 1: IRM cerebral cortes axial, coronal y sagital respectivamente.

En concreto, la imagen utilizada en la *Figura 1* es de tipo T1, esto es una constante de relajación transversal, también existe la T2 y densidad de protones (DP) entre otras. Estos se forman al excitar el núcleo de hidrógeno con pulsos de radiofrecuencias a su frecuencia natural de rotación, esto produce que algunos átomos cambien su orientación cargándose de energía que devolverán en forma de ondas de radiofrecuencia. Este proceso se conoce como relajación y depende de las condiciones que rodean al átomo como la interacción entre procesos de relajación y su movilidad. Según la duración y momento en el que aplicamos los pulsos de radiofrecuencia conseguimos estos fenómenos que determinan el contraste de la

imagen. [14] [15] En la *Figura 2* se puede ver unas imágenes del mismo sujeto potenciada en T1, T2 y DP respetivamente.

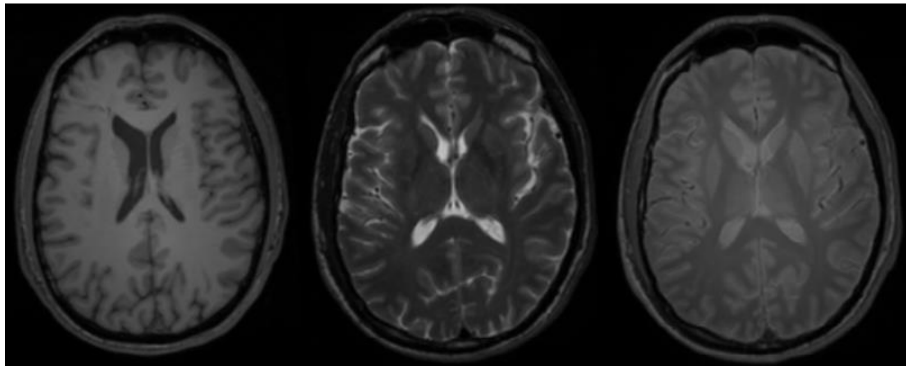


Figura 2: Imágenes potenciadas en T1, T2 y DP respetivamente.

Las imágenes RMN no están exentas de errores en el proceso de adquisición, como ruidos aleatorios y la no-homogeneidad de la señal, para este tipo de errores existen algoritmos que pueden corregirlos, pero existen errores que no pueden ser corregidos por algoritmos y son descartadas.

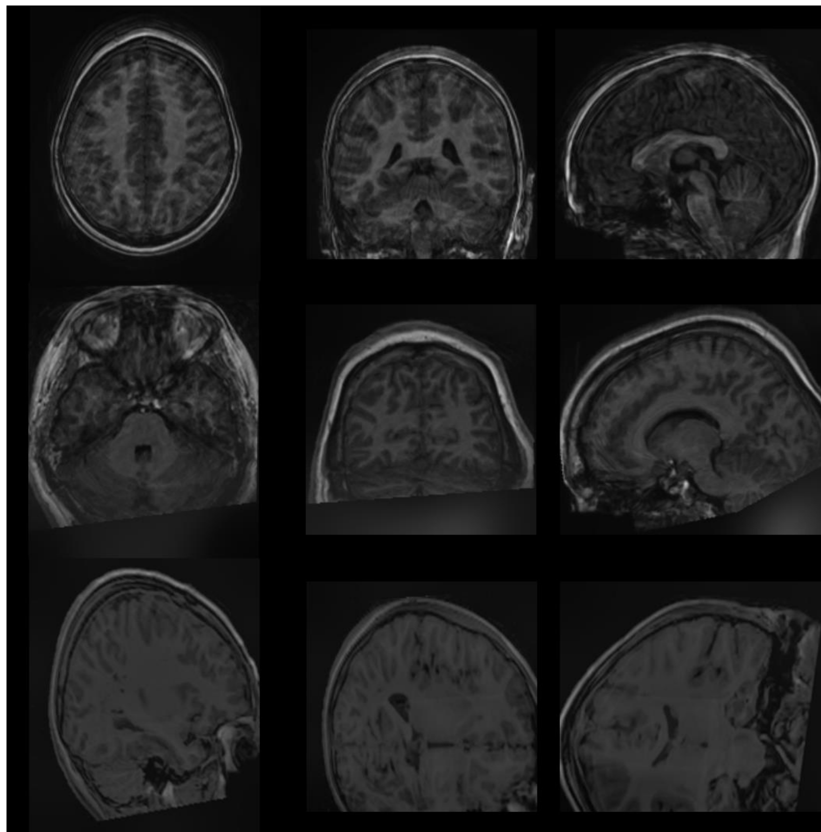


Figura 3: Imágenes con errores que no se pueden corregir con algoritmos.

En la *Figura 3* tenemos tres imágenes con distintos errores, estos errores no se pueden corregir mediante algoritmos por lo que no se pueden utilizar, este tipo de imágenes se descartarían.

En la primera fila, se puede ver mucho ruido correlado en forma de ondas, el ruido se puede eliminar hasta cierto punto, pero en este caso no se puede hacer nada.

En la segunda fila se puede ver un caso en el que falta una parte de la imagen y la corrección de inhomogeneidad ha fallada dando como resultado una iamgen inusable.

En la última fila los cortes están mal orientados y la reconstrucción es parcial, faltando mucha información.

2.2. *Deep learning*

El *Deep Learning* o aprendizaje profundo forma parte del aprendizaje automático o *Machine Learning* que a su vez forma parte de la inteligencia artificial (IA).

El papel principal de la inteligencia artificial es ofrecer soluciones con algoritmos y técnicas que puedan resolver tareas que las personas pueden resolver de forma automática. Como puede ser el control de calidad de imágenes o el reconocimiento de rostros.

El *deep learning* se puede dividir en varias ramas, como las redes neuronales, redes neuronales convolucionales, redes generativas adversarias, etc. Estas arquitecturas se usan en la visión por computador, en la realización de imágenes realistas al ojo humano creadas por "maquinas", en el reconocimiento del lenguaje, etc.

2.2.1. Redes neuronales

Una red neuronal es un modelo que intenta simular de forma simplificada el funcionamiento de las neuronas en un cerebro humano.

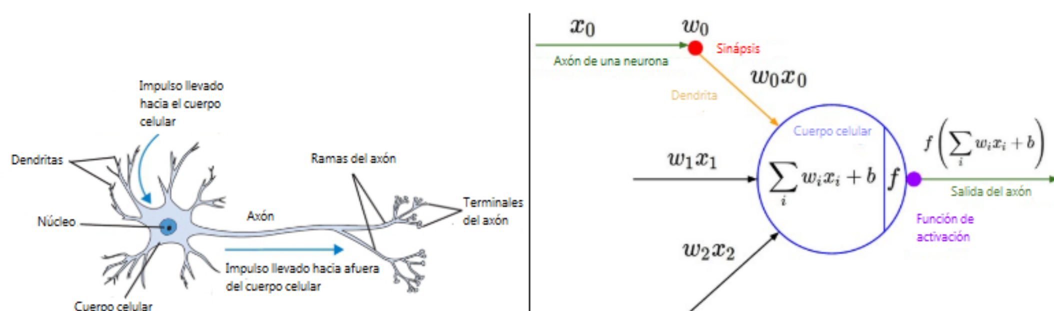


Figura 4: Esquema de una neurona biológica (Izquierda) y esquema de una neurona computacional (Derecha).

Como se puede ver en la figura 4, las neuronas computacionales (Unidades de procesamiento) tienen una gran similitud a una neurona biológica, como hemos dicho antes las redes neuronales intentan simular el funcionamiento de una red

neuronal biológica por lo que aquí se pueden ver esas similitudes. Las dos tienen dendritas desde las cuales les llega la información del resto de neuronas o la información de entrada, unos axones que permiten la conexión con el resto de las neuronas y un cuerpo celular al que le llega la información. En el caso computacional se nos presenta la forma en la que le llega la información, que está formada por una entrada X_i, donde i es el número de entradas, y unos pesos W_i, estos pesos indican la influencia de la entrada X_i sobre el resultado final y inicialmente tienen valores aleatorios que a medida que avanza el proceso de entrenamiento se van cambiando para acercarse a la solución requerida. Esto simula el proceso de aprendizaje de las neuronas reales. En un modelo básico las neuronas suman toda esta información y según cierto límite, la neurona envía la información. En el modelo computacional esto lo simula una función de activación a la cual se le pasa la información recibida y según su valor envía o no la información recibida, esto se explicará más extensamente más adelante.

La unión e estas neuronas forma una red neuronal que está formada por una capa de entrada, a la que se le introducen los datos iniciales, seguida de una capa o varias capas ocultas que realizan ponderaciones de los valores sobre los pesos y le pasan el valor a la siguiente capa que puede ser otra capa oculta o la capa de salida, que dependiendo de la cantidad de valores de salida que se requieran puede estar formada por una o más neuronas. Entre cada capa todas las neuronas están conectadas entre ellas, formando una enorme red. [1]

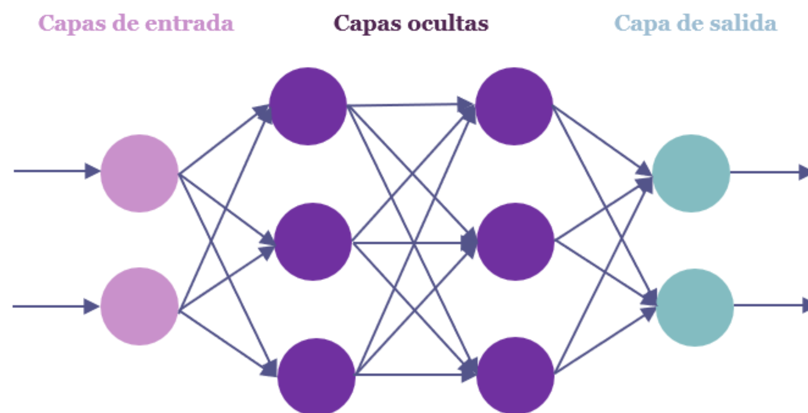


Figura 5: Esquema de una red neuronal.

2.3. Redes neuronales convolucionales

Las redes neuronales convolucionales funcionan prácticamente igual que las nombradas anteriormente, la diferencia es que las convolucionales operan la entrada mediante operaciones de convolución usando filtros que se aprenden durante el entrenamiento de la red. Además, la convolución reduce el número de pesos de la red lo cual permite entrenar redes con un gran número de capas.

Las primeras capas de una red neuronal convolucional extraen los bordes o esquinas de una imagen, en capas posteriores se detectan formas y al final se detectan características de alto nivel. Las últimas capas están totalmente conectadas y se dedican a dar una predicción a partir de los detalles de alto nivel.[1]

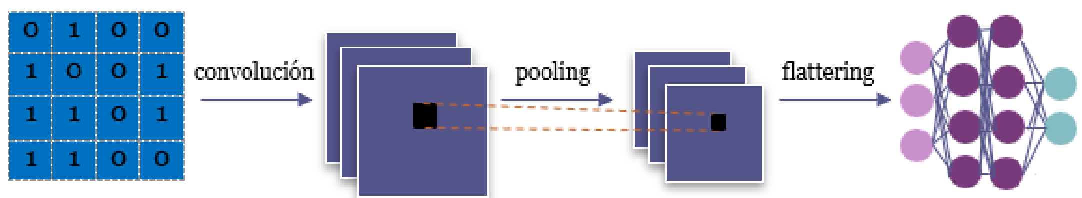


Figura 6: Red neuronal convolucional.

2.3.1. Convolución

La convolución es una operación cuyo objetivo es extraer las características de alto nivel de una imagen, en el primer nivel se extraen datos como los bordes, el color... Es necesario utilizar más de una capa de convolución para extraer características de alto nivel, en las últimas capas la red tendrá una comprensión total de la imagen.

Para realizar este proceso, se realiza una serie de operaciones sobre la matriz que representa la imagen, utilizando un kernel, también llamado filtro, se realiza un producto escalar con la matriz de la imagen y el kernel, en la siguiente figura se puede ver un ejemplo de este proceso. [2]

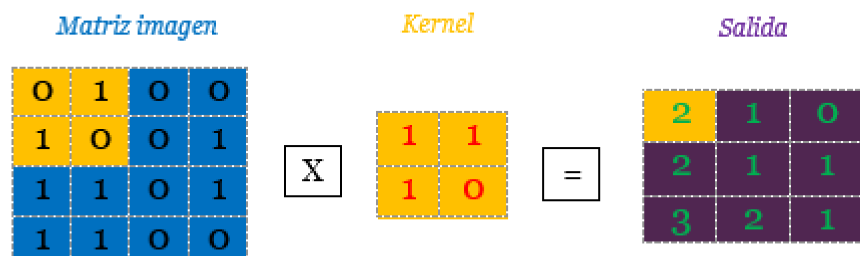


Figura 7: Producto escalar de una matriz imagen con un kernel y su resultado después de completar el proceso al mover el kernel por toda la matriz imagen.

2.3.2. *Batch Normalization*

Al insertar valores de entrada a una red, se normalizan sus valores ya que puede haber valores que se muevan entre 0 y 1 y otros entre 0 y 1000, esta normalización hace que la red pueda entrenar más rápido, y dado que entre las capas ocultas este también sucede se suele usar un método para normalizar estos valores entre las capas llamado *Batch Normalization*.

Este método normaliza la salida de una capa restando la media de los valores de salida y dividiendo por la desviación estándar del mismo batch. Al realizar esta normalización los pesos de la capa siguiente no son óptimos para estos valores de entrada, para ello *Batch Normalization* tiene dos parámetros de entrenamiento, de esta forma la salida se multiplica por una desviación estándar(gamma) y se le suma una media(beta) de esta forma se deja al descenso de gradiente se puede encargar de la renormalización de este valor y así adaptarlo a la siguiente capa, así no se pierde la estabilidad de la red. [8]

Input: Valores de x sobre el mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parámetros a aprender: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// Media del mini-batch}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// Varianza del mini-bach}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// Normalización}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// Escalar y tranformar}$$

Figura 8: Algoritmo de normalización Batch Normalization.

2.3.3. Funciones de activación

La función de activación es el paso anterior a que la neurona envie la información que ha recibido, esta función decide si el valor de la neurona estará entre 0 y 1(no activada o activada) o entre -1 y 1 dependiendo de la función.

Existen funciones lineares y no lineares, las primeras se suelen usar en problemas de regresión ya que no tienen un rango definido, suelen variar entre menos infinito y más infinito. Las funciones más utilizadas son las no-lineares, ya que permiten un mayor rango de variación a los datos y están limitadas a valores específicos, las más conocidas son *Sigmoide*, *ReLU* (Rectified Linear Unit) y *Softmax* que se abordarán en los siguientes puntos. [3]

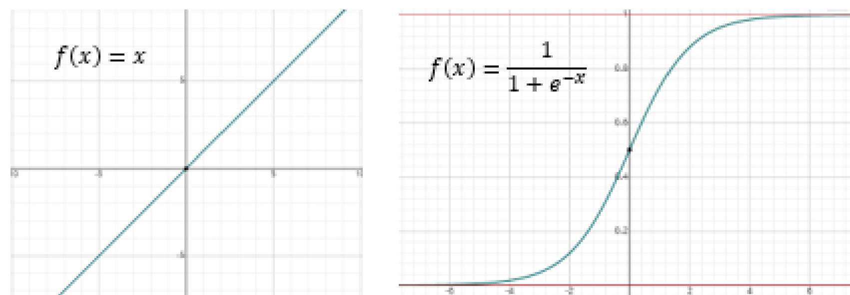


Figura 9: Función linear (Izquierda) y función no-linear Sigmoide (Derecha).

Función Sigmoide

Como podemos ver en la gráfica de la derecha de la *Figura 6*, la funcion sigmoide tiene una forma de *"S"* que se ve esta acotada entre 0 y 1, por lo que suele ser utilizada para problemas de clasificación. Valores negativos resultan en valores proximos a zero mientras que valores positivos se acervan a 1, el principal problema de estas funciones es que pueden dar lugar al famoso problema gradient vanishing, ya que variaciones pequeñas de la salida dan a su vez valores pequeños de los gradientes que permiten el entrenamiento de la red.

Función ReLU

La función ReLU es de las más utilizadas en aprendizaje máquina, ya que su coste computacional es muy bajo y mínima el problema dado en la función Sigmoide. La función tiene la siguiente representación.

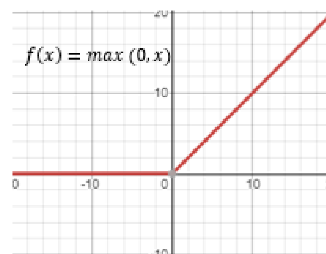


Figura 10: Representación gráfica de la función ReLU.

Como se puede apreciar, para valores negativos la función toma el valor cero y para valores positivos toma el valor *x, esto hará que la neurona se active para todo valor positivo y no lo haga para todo valor negativo.*

ReLU puede dar problemas en ocasiones, ya que, si en algunas neuronas el valor de x es negativo al entrenar, al hacer *backpropagation* estas permanecen inactivas y sus pesos no se actualizan, por lo que la red podría no estar aprendiendo, este puede reducirse si la ratio de aprendizaje se ajusta apropiadamente. [3]

Función Softmax

Softmax es una función cuyo objetivo es devolver una serie de probabilidades según las entradas que le llegan. Softmax puede trabajar con varias clases a la vez, al trabajar con probabilidades los valores que devuelve están entre 0 y 1 y la suma de los valores es 1. [4]

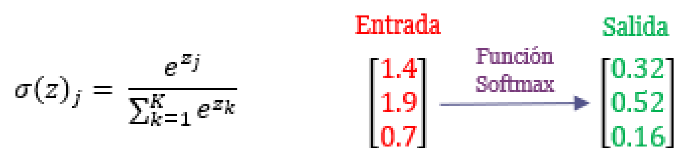


Figura 11: Representación matemática de la función Softmax (Izquierda). Ejemplo resultado de la función Softmax para una entrada dada (derecha).

Esta función es muy útil en los problemas de clasificación, como en la clasificación de imágenes donde se tiene que clasificar el objeto que aparece en la imagen entre varias posibilidades, en estos casos devuelve varias probabilidades para los distintos casos dados, siendo el más prometedor el que más probabilidad tiene si la red neuronal está bien entrenada.

2.3.4. Función de pérdida

La función de pérdida es la que se encarga de calcular cuan lejos se está del valor deseado, existen varias formas de calcularla, desde formas simples como el error cuadrático medio que es la suma de la diferencia entre la salida deseada y la salida obtenida al cuadrado, como otros más complejos como es la entropía cruzada.

La entropía cruzada se utiliza cuando los valores de salida se mueven entre 0 y 1, cuando el valor predicho se aleja del valor real más grande es el resultado de la función.

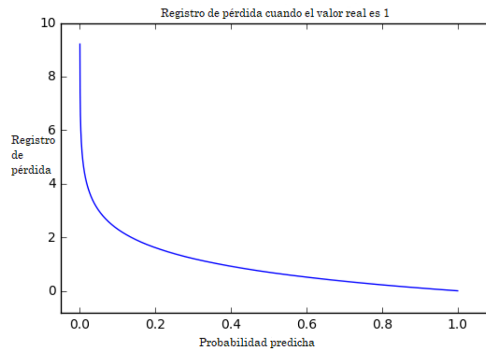


Figura 12: Valores para función de pérdida de entropía cruzada.

Existen dos variaciones de esta, la entropía cruzada binaria (*Binary cross-entropy*) y la entropía cruzada categórica (*Categorical cross-entropy)*

$$Binary\ cross - entropy = -(y \log(p) + (1 - y) \log(1 - p)) \quad (1)$$

$$Categorical\ cross - entropy = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) \quad (2)$$

Donde:

- y es el valor deseado
- p es el valor predicho
- M es el número de clases

Como se puede observar la ecuación de *binary cross-entropy* calcula la probabilidad de que el valor predicho sea el valor deseado más la probabilidad de que el valor deseado no sea el predicho. Esta solo sirve cuando solo existen 2 clases.

Para más clases se utiliza la *categorical cross-entropy*, que como se puede ver en la segunda ecuación se calcula la función de perdida para cada clase y se suma. [16]

2.3.5. Optimizador

Un optimizador se encarga de calcular el valor al que se han de cambiar los pesos para acercarse al valor deseado, de esta forma los optimizadores utilizan la información de la función de pérdida y de los pesos para saber si se está acercando o alejando del valor deseado. Al principio los optimizadores van a ciegas ya que el valor de los pesos comienza siendo aleatorio, con el tiempo van acercándose al valor deseado.

El optimizador más conocido es el de descenso de gradiente que estima como afectará pequeñas variaciones en el valor de los pesos a la función de pérdida y así decide qué valor dar a cada peso, repitiendo esto acaba acercándose al valor deseado, los valores que analiza dependen del tamaño del *batch* que indica cada cuanto se actualizarán los valores de los pesos. Existen variaciones como el descenso de gradiente estocástico (SGD siglas en inglés) que para evitar tener que hacer grandes cálculos cuando el *batch* es muy grande, este algoritmo elige una muestra aleatoria del batch y así se acerca al valor deseado, este algoritmo es mas

lento y produce más ruido durante el aprendizaje, pero acaba cumpliendo el mismo objetivo. [18]

Otro optimizador muy utilizado es el ADAM (*Adaptative momentume estimation*) este utiliza valores de gradientes anteriores para calcular el gradiente actual y añade el concepto de *momentum*, este ayuda a encaminar más rápidamente al gradiente añadiendo fracciones del gradiente que se actualizó anteriormente. [17]

Existen otros optimizadores que se acercan al valor deseado utilizando distintos métodos, entender las ecuaciones de estos puede ser una tarea tediosa, pero el objetivo de todos es el mismo, minimizar la función de error.

2.3.6. *Pooling*

El *Pooling* tiene la función de reducir la representación espacial y así liberar la carga de parámetros y computacional para la red, adicionalmente ayuda a reducir el sobredimensionamiento(*overfitting*).

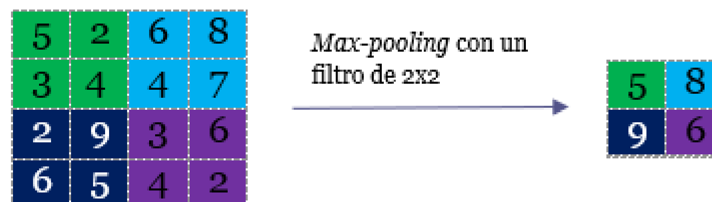


Figura 13: Ejemplo de funcionamiento de la función max-pooling.

En la *Figura 12* se aprecia el funcionamiento de la función max-pooling una de las funciones más utilizadas para hacer *pooling* esta función utiliza una matriz de NxN para dividir la matriz resultante de la convolución en varias secciones y de cada sección seleccionar el valor más grande, la dimensión que más se utiliza es la 2x2, de esta forma descarta el 75% por ciento de las activaciones reduciendo enormemente la carga computacional de la red como se dijo anteriormente, la función afecta también en profundidad a las imágenes de forma independiente, pero la dimensión resultante de la matriz es la misma en todos los niveles de profundidad.[1]

Como se puede ver, en la figura 9 una matriz de dimensión 4x4 se ve reducida a una de 2x2 y un desplazamiento de 2 donde quedan solo los valores más grandes de cada sección que serían los más relevantes.

Pero todo el tamaño resultante se calcula de la siguiente forma:

Entra una matriz de tamaño $\boldsymbol{W_1 x H_1 x D_1}$ en la figura 9 es 4x4x1

Con dos parámetros para el *pooling*, que son:

F, la dimensión espacial del *pooling*, comúnmente 2, como en la figura 9(2x2).

S, que indica cuantos pixeles se moverá la matriz (**F**x**F**) para dividir la imagen en X secciones, en el caso de la figura 9 este valor es 2.

Con estos datos se puede calcular la matriz **$W_2 x H_2 x$:**

$W_2 = ((W_1 - F) / S) + 1$, en el ejemplo: $W_2 = ((4-2)/2)+1 = 2$.

$H_2 = ((H_1 - F) / S) + 1$, en el ejemplo: $H_2 = ((4-2)/2)+1 = 2$.

$D_2 = D_1$, en el ejemplo: $D_2 = 1$.

2.3.7. *Dropout*

Esta función es muy simple, según la probabilidad X que se le indique de entrada, esta desactivará X% de las neuronas de la capa, en este caso desactivar la neurona es como eliminarla temporalmente junto a todas sus conexiones de entrada y salida. Esto se utiliza para regularizar la red evitando que sufra de *overfitting*. Por lo general esto solo se utiliza durante el entrenamiento, aunque más adelante se verá que en algunos casos utilizar dropuout durante test tiene sus ventajas. [5][6]

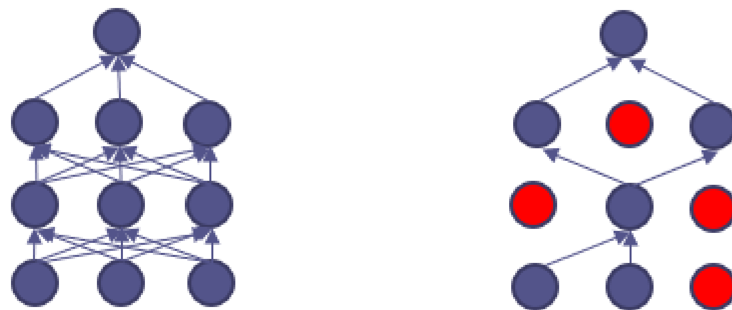


Figura 14: Dropout. Modelo de red neuronal estándar sin dropout (Izquierda). Modelo de red neuronal con dropout, las neuronas rojas han sido desactivadas(derecha).

2.3.8. *Flattening*

Al finalizar el proceso de convolución se tiene que realizar el proceso de clasificación el cual utiliza una red neuronal totalmente conectada, dado que el proceso anterior devuelve una matriz de valores hay que "aplanarla" (*Flattening*)¸de manera que quede un vector de valores que puedan ser insertados en la red. [9]

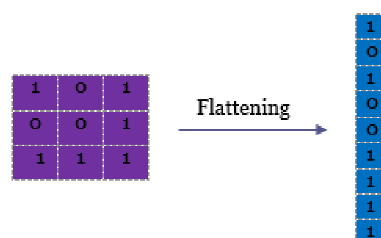


Figura 15: Ejemplo de flattening.

2.3.9. *Dense layer*

Como se indica en el anterior punto, al acabar el proceso de convolución se ha de hacer la clasificación, para esto se utilizan capas totalmente conectadas, aveces llamada *dense layer*, que tienen como entrada un vector lineal y lo mismo como salida.

Dependiendo de la cantidad de valores de entrada aveces se utilizan varias capas de este tipo para ir simplificando la salida cada vez a menores valores, según la necesidad del usuario. En la *figura 5* se puede ver un ejemplo de esto.

2.4. Entrenamiento del modelo

Una vez seleccionados los datos para el dataset de entrenamiento, pre-procesados los mismos y creado el modelo, se pasa a entrenarlo. Para esto se le pasan las imágenes y las etiquetas correspondiente a cada una. Durante el entrenamiento el modelo utiliza dos procesos importantes, *forward-propagation* y el *back-propagation*.

2.4.1. Forward-propagation

Este proceso implica el funcionamiento natural de una red neuronal desde el input hasta el output, donde para este caso se le ha dado una serie de imágenes de entrada con sus respectivas etiquetas y su objetivo es por cada imagen acertar en la predicción al compararla con la etiqueta en cuestión.

Dado que el modelo se inicia de forma aleatoria se utiliza la función de pérdida (*loss function)*, que indica el acierto de la red durante el entrenamiento, cuanto más falla la red en la predicción más grande es el resultado de la función. Una función de pérdida básica es:

$$pérdida = 1/2(VD - VP)^2$$

Donde VD es Valor Deseado y VP es Valor predicho. El objetivo de la red neuronal durante el entrenamiento es acercar a todo lo posible este valor.

En contraposición a la función de pérdida tenemos la precisión, la cual se mueve entre los valores 0 y 1, cuan más alto es este valor significa que la red está prediciendo mejor los valores. Este valor se calcula con la siguiente función:

$$precisión = \frac{VP}{VP + FP}$$

Donde VP es verdaderos positivos y FP es falsos positivos, cuantos más verdaderos positivos haya más alto será el resultado, como se puede apreciar en este caso se busca maximizar el valor de esta función a 1. [10]

2.4.2. *Back-propagation*

Tras la finalización del *forward-propagation* sobre un batch, se realiza el *back-propagation* cuyo objetivo es realizar las derivadas parciales $\partial C/\partial w$ y $\partial C/\partial b$ de la función de coste C respecto los pesos w y respecto a sus *biases* b, esto sobre cada neurona de la red.

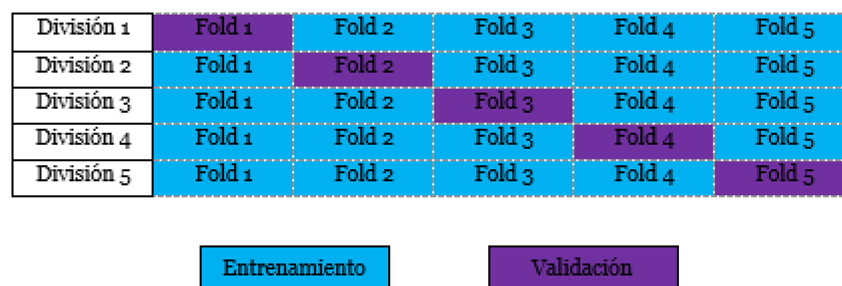
Para realizar estos cálculos primero tiene que calcular el error δ derivado de cada neurona, donde $\delta = \partial C/\partial z$ donde z es un valor que cambia el peso de entrada de la neurona, esto error se acaba propagando por el resto de las neuronas. El valor de este error indica hacía a donde se ha cambiado el valor del peso, si este valor es cero significa que no se ha cambiado el peso de la neurona. Tras calcular este valor para todas las neuronas al relacionarlo con las derivadas $\partial C/\partial w$ y $\partial C/\partial b$ esto nos da una respuesta de hacía a donde variar los valores de los pesos y las *biases*.

Back-propagation solo calcula el gradiente para un ejemplo de entrenamiento, por lo que es usual combinarlo con una función optimizadora como es *ADAM* para calcular el descenso de gradiente sobre varias muestras. [19]

2.4.3. *K-Fold Cross validation*

El k-fold cross validation sirve para tener un mejor indicador sobre como de bien predecirá el modelo sobre datos sobre los que no ha sido entrenada.

Para realizar esto, las muestras se dividen en K subconjuntos, cada conjunto tiene unas muestras diferentes para el entrenamiento y otras para la validación. Por lo que hay que realizar un entrenamiento por cada subconjunto de muestras. De esto resultan K modelos de los cuales se calcula la media de los errores de todas para conseguir una estimación del error. [11]



División 1	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
División 2	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
División 3	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
División 4	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
División 5	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5

Figura 16: 5-fold cross validation.

2.4.4. *Mixup*

Las grandes redes neuronales profundas son potentes, pero tienen comportamientos no deseables como la memorización, lo que puede producir que memorice un patrón de algunas muestras en vez de los patrones deseados en las muestras, lo que hará que falle al predecir sobre muestras sobre las que no ha sido entrenada.

Una forma de minimizar esta memorización es el uso de *mixup*, que es un método de data augmentation, que consiste en realizar cambios en los datos para

conseguir más variedad. *Mixup* lleva esto a un nuevo nivel cuyo método es tomar dos muestras y realizar una combinación lineal de estas, tanto de la entrada bruta, como de las etiquetas de cada una.

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \quad donde\ x_i, x_j\ son\ vectores\ de\ entrada\ brutos$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j, \quad donde\ y_i, y_j\ son\ etiquetas\ de\ codificación$$

En las formulas vistas se aprecia como la variable *lambda* es un valor aleatorio con distribución *Beta* por lo que produce mezclas aleatorias durante el entrenamiento. [7]

2.5. Testear Modelo

El proceso final a la hora de crear una red neuronal y entrenarla, es testear el modelo obtenido para comprobar que efectivamente los valores obtenidos durante el entrenamiento son reales, ya que puede haber algunos problemas como que la red haya aprendido patrones solo relevantes para los datos entrenados, razón por la que hay que probarla con datos con los que no se ha entrenado al modelo.

Para esto se pone a prueba el modelo pasándole dichos datos y comparando la predicción del modelo con los valores a obtener deseados, esto es prácticamente igual al proceso explicado en el punto *Forward-propagation*, la diferencia es que en este momento la red ya ha sido totalmente entrenada, al igual que antes obtenemos el valor de precisión, y añadimos los valores de sensibilidad, que nos da una estimación de cuantos verdaderos positivos predice la red, y la especificad, que nos da una estimación de los falsos positivos que estima la misma.

2.5.1. Bayesian Dropout

Normalmente durante el proceso de test, el dropout se inicializa a cero, que significa que no tiene ningún efecto sobre la red durante este proceso. Esto suele ser así dado que el dropout “desactiva” neuronas de forma aleatoria, cosa que puede afectar gravemente al resultado de test.

Bayesian dropout consiste en activar el *dropout* durante test lo que introduce aleatoriedad en la red, que al realizar varias predicciones sobre las mismas muestras y realizar un promedio, estas tienden al valor esperado y reducen la varianza del error.

3. Tecnología, problema e implementación

En este apartado se repasará la tecnología utilizada, como son las librerías y el entorno utilizado, el problema a resolver y que pasos son necesarios para la implementación de una red neuronal convolucional, empezando con la creación del dataset, el tratamiento de estos, la creación del modelo, su entrenamiento y testeo de este.

3.1. Tecnologías implementadas

Aquí se presentan las tecnologías software implementadas para la realización del proyecto.

Anaconda

Anaconda es una distribución de *Python* y *R* muy utilizada en ciencia de datos y aprendizaje automático.

Para este proyecto ha sido muy útil ya que da la posibilidad de crear varios entornos donde probar distintas librerías sin afectarse unos entornos a otros y así en caso de producirse algún error poder revertirlo fácilmente.

También permite seleccionar la versión de las librerías que se están usando de forma sencilla, lo que permite solucionar problemas de incompatibilidad de versiones.

Matlab

Es un sistema de cómputo numérico que tiene su propio lenguaje de programación y ofrece herramientas para la manipulación de matrices, representación de datos y funciones, la implementación de algoritmos…

Python

Python es un lenguaje de programación interpretado. Es un lenguaje de programación multiparadigma, porque soporta la orientación a objetos, la programación imperativa y la programación funcional.

En este proyecto se ha utilizado por tener acceso a varias librerías dedicadas al aprendizaje profundo, que son de gran utilidad dada la orientación de este.

Keras

Keras es una librería para *Python* centradas en las redes neuronales, puede funcionar junto a *tensroflow*, *theano* y *CNTK*, contiene varios métodos que ayudan tanto en la creación de modelos de redes neuronales, como en el entrenamiento y testeo de esta.

Numpy

Es una librería para *Python* que agrega soporte para vectores y matrices también funciones matemáticas de alto nivel.

Matplotlib

Librería para *Python* que provee herramientas para realizar gráficas, histogramas, graficas de barras, de errores… En pocas cantidades de líneas. Es muy eficiente y fácil de utilizar.

SciPy

SciPy es otra librería para *Python* que proporciona muchas rutinas numéricas fáciles de usar y eficientes, tales como integración numérica, interpolación, optimización, algebra lineal y estadística.

Scikit-learn

Es un software gratuito de aprendizaje máquina para *Python*. Cuenta con varios algoritmos de clasificación, regresión y agrupamiento, que incluye maquinas de vectores de soporte, aumento de gradiente, DBSCAN y herramientas para realizar *k-fold cross validation*. Esta diseñado para interactuar con las bibliotecas numéricas y científicas *Numpy* y *SciPy*.

Nibabel

Es una librería que provee métodos de lectura y escritura para los formatos de neuroimágenes más comunes como *GIFTI, NIfTI1, NIfTI2, CIFTI-2*… [15]

Os

Una librería que da acceso a *Python* a las instrucciones internas del sistema que se está utilizando.

Glob

Librería que da acceso a una función de búsqueda de archivos en ficheros en las que se pueden utilizar valores como ‘*’ o ‘?’ para realizar búsquedas más concretas.

Itertools

Módulo de *Python* que permite construir poderosos iteradores de forma eficiente y da potentes herramientas para el manejo de estos.

Sublime Text

Es un editor de texto especializado en la programación, tiene soporte para varios lenguajes de programación y acceso a algunas de sus *API’s* como la de Python. También tiene varias funciones para la edición de texto que lo hacen más eficiente que otros editores.

ITK-SNAP

Es una aplicación que sirve para realizar segmentaciones de estructuras 3D de imagen médica como para su visualización y navegación por ellas.

3.2. Problema

Las bases de datos que se utilizan para entrenar redes neuronales tienen imágenes que presentan distintos errores que pueden producir que esta aprenda patrones no correctos y por lo tanto que pierda precisión a la hora de dar el resultado de los análisis. Para evitar esto, se necesita de una o varias personas que dediquen tiempo e infraestructura en realizar un control de calidad de estas imágenes, esto conlleva perder una gran cantidad de tiempo y dinero del proyecto en una tarea que se realiza de una manera prácticamente sistemática, ya que consta de ir mirando cada imagen una a una y si se detecta algún error descartarla, como se ha dicho, para una persona esta tarea conlleva mucho tiempo realizarla a pesar de lo simple que es.

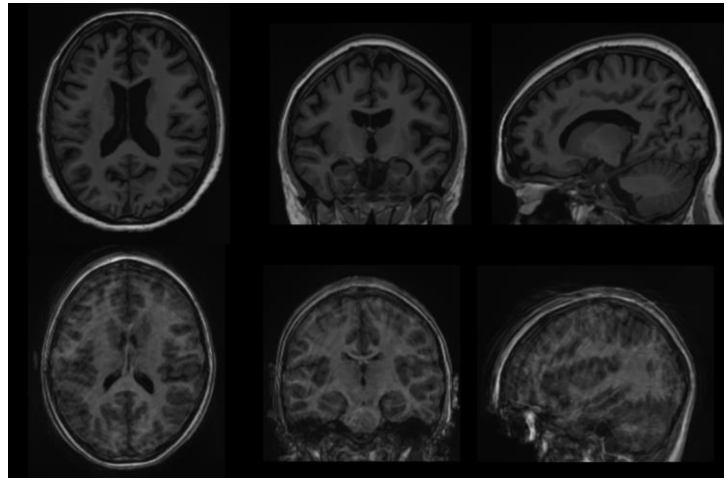


Figura 17: Ejemplo imágenes de buena calidad (arriba) y mala calidad (abajo).

Con el fin de solucionar este problema se propone realizar el control de calidad con *Deep learning* mediante el entrenamiento de una red neuronal convolucional que pueda distinguir de forma eficiente entre estas imágenes de buena y mala calidad como las que se muestran en la *Figura 17*. En caso de conseguir este objetivo, al ser un ordenador el que realiza el proceso, si este tiene un buen procesador, se puede reducir una tarea que, según la cantidad de imágenes, puede durar varias horas o incluso días, a unos minutos, dado que la eficiencia de clasificación de una red neuronal es mucho mayor que la de un humano. Por ende, esto reduce drásticamente el tiempo necesario para realizar esta tarea, el personal y el dinero y en este caso, ayudará a los sistemas automáticos de segmentación y análisis volumétrico a obtener mejores resultados y esto mejorará también el diagnóstico clínico que tendrá un impacto positivo en la salud de los pacientes.

3.3. Implementación

En este capítulo se explica cómo se ha implementado paso por paso la red neuronal convolucional que solventará el problema explicado en el capítulo anterior. Donde se explican los objetivos primarios presentados en el punto 1.2 del documento y se acabarán de tratar los últimos puntos de los objetivos en el capítulo de experimentación.

3.3.1. Creación del dataset

Para entrenar una red neuronal se necesita de datos de entrenamiento relacionados con el problema que se quiera solucionar y separados según las distintas clases que se necesiten como salida. Esto conlleva un trabajo de búsqueda y clasificación manual que puede llevar mucho tiempo.

En este caso el tutor del trabajo disponía de más de 3000 datos de acceso libre los cuales ya estaban separados entre los datos requeridos que eran IRM (imágenes de resonancia magnética) de tipo T1, que ya se han visto en el estado del arte, estos datos fueron obtenidos de la base de datos de *volbrain*.

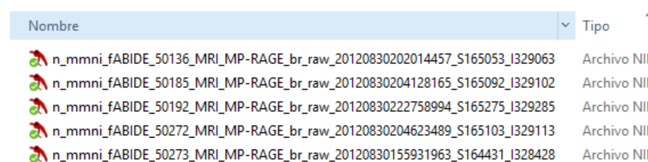


Figura 18: Algunos archivos IRM.

Estos archivos habían sido previamente preprocesado con el software volBrain y etiquetados manualmente como bueno y malos.

En un primer momento tras separar los archivos se obtuvieron unas 60 imágenes de mala calidad y más de 140 buena calidad, no se utilizaron más de buena calidad para no desbalancear las muestras en gran medida. Más adelante, por la necesidad de mejorar el resultado se buscaron más datos de mala calidad y añadieron algunos de buena calidad, para esto se hizo un script que abría uno a uno cada archivo y mostraba 3 cortes, axial, coronal y sagital, cada uno de una sección de la imagen y pulsando los botones “b” o “m” se decidía se enviaba cada archivo a su respectiva carpeta “buenas” y “malas”. Con la ayuda del tutor se fue filtrando cada archivo consiguiendo un conjunto de datos más grande con el que trabajar de 135 imágenes de mala calidad y se seleccionaron 160 de buena calidad para balancear el dataset.

Tras finalizar el proceso de selección de los datos se tiene que realizar un tratamiento sobre los mismos, que se explica en el siguiente punto.

3.3.2. Tratamiento de los datos

Tras tener los datos clasificados manualmente estos han de pasar por un tratamiento, dado que algunas imágenes tienen un brillo distinto y hay que normalizarlas y se han de poner las etiquetas correspondientes según sus respectivas clases.

En este caso se han dividido los datos en dos clases, la clase 0 para los datos buenos y la clase 1 para los datos malos, hecho de esta forma porque el objetivo es encontrar los datos de mala calidad y separarlos de los buenos.

Como la experimentación se dividió en dos partes, hay dos formas de tratar los datos, una para los archivos en dos dimensiones y otra para los de tres dimensiones.

Comenzaremos con el tratamiento para los datos en dos dimensiones:

- Primero con un script hecho en Matlab, cuya función es abrir una imagen tipo *nifti* y crear una imagen *jpg* de una sección de cada corte, resultando en tres imágenes. Como las que se ven en la *Figura 15*.

- Tras esto se creó un script Python que abre cada imagen *jpg* de tres en tres, cada corte de su correspondiente archivo, y realiza una normalización sobre las tres de esta forma quedan todas con un brillo y color parejo.

```
# Este método normaliza las imagenes.
def normalize_treatment(X):
    X_mean = np.mean(X)
    X_std = np.std(X, dtype=np.float32)
    X = (X - X_mean)
    X /= X_std

    return X
```

Figura 19: Código del tratamiento de normalización

- Después se realiza la normalización, este proceso junta las tres imágenes en un array numpy que a su vez se inserta dentro de otro quedando de esta forma dos arrays con imágenes de buena y mala calidad.

- Antes de devolver los datos se realiza una permutación de estos en orden aleatorio para que al realizar el *K-fold cross validation* los datos en test estén bien balanceados.

- Tras todo esto, el script devuelve seis datos, *X_train*, siendo este donde se encuentran el conjunto de imágenes tratadas, en cada posición se encuentra un array de tres niveles uno con cada corte, como se explica en el punto anterior, *T_train*, que son las etiquetas que están ordenadas con los datos de *X_train*.

Ahora se explicará el proceso a seguir para los archivos en tres dimensiones:

- Primero se abre la imagen con método de la clase *nibabel* para leer archivos tipo *nifti*, que devuelve un array que en su interior contiene una representación numérica de la imagen de dimensión 181x217x181 voxels.

- Acto seguido se realiza un diezmado dado que el procesador que se estaba utilizando no aguantaba el entrenamiento con el tamaño original de la imágene, el tamaño se divide en dos, resultando en imágenes con dimensión 91x109x91 voxels.

- Tras el diezmado se realiza la normalización de la imagen que realiza los mismos pasos que se ven en la *Figura 17*, se calcula la media de todos los valores de la imagen, la desviación estándar y tras esto se le resta la media a cada valor de la imagen y se divide entre la desviación estándar, resultando como se ha dicho anteriormente en una imagen con un brillo y color parejos.

- Finalmente se inserta esta imagen en un array y en otro se inserta la etiqueta correspondiente.

- Por este proceso pasan todas las imágenes, para hacerlo más fácil se leen primero las de buena calidad y después las de mala calidad y por cuestiones que se explican más adelante el script devuelve cuatro datos, por un lado, tenemos x_good y x_bad donde se encuentran los datos buenos y malos respectivamente y por otro, tenemos y_good y y_bad donde se encuentran las etiquetas de cada clase.

3.3.3. Creación de un modelo de red neuronal convolucional

Todas las redes neuronales vienen definidas por una modelo o arquitectura donde se pueden ver los procesos por los que pasarán los datos, esta arquitectura puede presentar los procesos explicados en el *punto 2.3, convoluciones, batch normalization, poolings, dropout, flattening y dense.*

Keras dispone de una serie de métodos que simplifican la utilización de estos procesos, se puede indicar el tipo de convolución que se quiere utilizar, según el dato de entrada *Conv2D, Conv3d*, como ejemplo para entradas de datos en dos o tres dimensiones respectivamente, a estos se les tienen que indicar ciertos parámetro como el número de tensores de cada capa, los pasos que se moverá el *kernel*, si se rellena o no la imagen y la función de activación que se va a utilizar, *Maxpooling2D y Maxpooling3D* al que en este trabajo solo se le indica el tamaño del kernel a utilizar, para el resto de procesos también existen sus respectivos métodos, durante le experimentación se explicará como se fueron cambiando los valores de los distintos parámetros, muchos de ellos se dejaron fijos, sobre todo se probaron distintos tipos de activaciones y números de filtros y cantidad de convoluciones a realizar por cada nivel de convolución.

Para el modelo en dos dimensiones, la arquitectura está formada por un nivel de convolución para la entrada y seis capas de convolución en las que antes de

cada una se realiza un batchnormalization y entre cada capa de convolución se realiza un max-pooling, finalmente se realiza otro batchnormalization, una capa de flatten y 2 de dense que van reduciendo el número de salidas desde 512 hasta 2.

El modelo en 3 dimensiones es prácticamente igual al anterior, aparte de cambiar algunos parámetros y métodos para adaptarlos a las 3 dimensiones, está formado por cinco capas de convolución, el resto es igual a la de dos dimensiones. En la siguiente figura se puede ver un esquema de este modelo, no presenta los cambios en los filtros tras cada convolución al entrar en el ciclo, que se van multiplicando en exponentes de 2 empezando desde 32, siguiéndoles 64, 128…

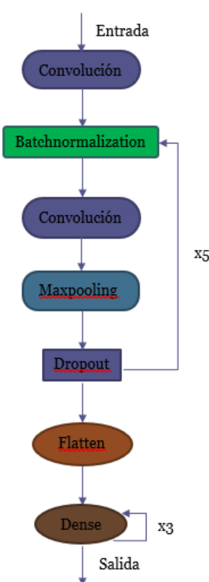


Figura 20: Esquema del modelo de 3 dimensiones.

3.3.4. Entrenamiento del modelo

Una vez se ha obtenido el dataset, tratado los datos y creado el modelo, se pasa a entrenarlo.

Keras tiene varias herramientas para entrenar un modelo, un método llamado *fit* que empieza el proceso de entrenamiento, este método necesita de ciertos parámetros como son los datos de entrada de entrenamiento y las etiquetas, los datos de validación, el tamaño del batch, que indica cada cuanto se actualizará el gradiente, el número de epochs, que indica cuantas veces se entrenará el modelo sobre todo las muestras dadas, también permite indicar si se quiere barajar los datos tras cada epoch, lo que permite que la red no pueda tomar el orden de los datos como un valor a aprender y también insertar algunas funciones, como puede ser que cada vez que se alcance el valor mas alto en la precisión de la validación se guarde ese modelo de la red. Este método se utiliza en el trabajo para el modelo de dos dimensiones.

Para el modelo de tres dimensiones es necesario utilizar otro método que permite cargar los datos de uno en uno o en pequeños paquetes, ya que al ser datos muy pesados cargarlos todos a la vez sobrecarga la memoria. Para esto se utiliza el método *fit_generator*, al cual como primer parámetro se le inserta una función,

que puede ser algo tan simple como una función que pase de uno en uno los datos de entrenamiento o que haga algún proceso a los datos como el que presentamos en el punto sobre el Mixup. Durante la experimentación se probaron las dos maneras, más adelante se indicarán los resultados de estos. En la siguiente figura se muestra uno de los generadores creados para realizar el mixup, presenta comentarios para hacer más fácil su comprensión.

```
# Definición de la funcion MixUp con disimulitud
def MixUp_Generator_dis(x1,y1,x2,y2,alfa):

        while(1):
                ind1=np.random.permutation(x1.shape[0]) # Devuelve un array de tamaño x1 con indices en orden aleatorio
                ind2=np.random.permutation(x2.shape[0]) # Devuelve un array de tamaño x2 con indices en orden aleatorio
                for n in range(0,x2.shape[0]):
                        #mixup
                        a=np.random.beta(alfa,alfa)     # devuelve un valor aleatorio entre alpha y 1
                        x3=a*x1[ind1[n]]+(1-a)*x2[ind2[n]]  # Mezcla de datos x1 y x2 según el valor de alpha
                        y3=a*y1[ind1[n]]+(1-a)*y2[ind2[n]]  # Mezcla de datos y1 y y2 según el valor de alpha
                        x3=np.reshape(x3,(1,x1.shape[1],x1.shape[2],x1.shape[3],x1.shape[4]))   # Normaliza la dimensión de los datos
                        y3=np.reshape(y3,(1)) # Normaliza la dimensión de las etiquetas
                        yield x3,y3  # devuelve los datos y las etiquetas mezlcadas
```

Figura 21: Ejemplo de generador. Mixup_Generator_dis utiliza el método mixup.

La mayor parte de parámetros necesarios para *fit_generator* son los mismos que para *fit* a diferencia del generador, explicado en el párrafo anterior y que en este caso hay que facilitarle el número de datos que tendrá que analizar en cada *epoch* en un parámetro llamado *steps_per_epoch*. En la siguiente figura se pueden visualizar las diferencias entre estos métodos

```
history = model.fit(X_train, T_train, batch_size, epochs, verbose=1, validation_data=[X_val, T_val], shuffle=True,
                        callbacks=[savemodel])

history = model.fit_generator(MixUp_Generator_dis(X_good, Y_good, X_bad, Y_bad,0.3), steps_per_epoch = X_train.shape[0],
    epochs = epochs, verbose=1, validation_data=(X_val,Y_val), shuffle=True, callbacks=[savemodel])
```

Figura 22: Métodos fit y fit_generator

Durante el entrenamiento se puede visualizar la cantidad de datos analizados, la barra de progreso y el valor de la función de pérdida y la precisión de cada epoch, se pueden añadir los valores en validación de la función de pérdida y la precisión. Analizando estos valores se puede ver como va progresando la red durante el tiempo, si ha llegado a converger, si no se estabiliza para ningún valor, lo cual podría significar que hay problemas, como puede ser overfitting, que puede darse por la red tener demasiada potencia o porque los datos dados no son lo suficientemente variados, o underfitting, que puede darse por la red no tener la potencia necesaria para solucionar el problema o por estar entrenándola para un caso particular y tener datos variados en la validación, en los ambos casos la red generaliza los patrones de las muestras dadas y da una respuesta errónea durante la validación.

```
256/256 [========================] - ETA: 30s - loss: 8.3174 acc: 0,5 – val_loss: 8,7364 val_acc: 0,4512
256/256 [========================] - ETA: 30s - loss: 7.3534 acc: 0,6422 – val_loss: 8,2828 val_acc: 0,4512
256/256 [========================] - ETA: 30s - loss: 7.3534 acc: 0,6422 – val_loss: 8,2828 val_acc: 0,4512
204/256 [====================>..........] - ETA: 30s - loss: 6.3174 acc: 0,7257
```

Figura 23: Datos en pantalla durante entrenamiento

Pero para analizar estos valores se necesita crear una visualización más gráfica ya que puede ser difícil imaginarse que está pasando solo leyendo los números y *Keras* no presenta facilidades para realizar gráficas y poder hacerlo más fácil de interpretar, por lo que se ha utilizado la librería matplotlib con la que se pueden realizar graficas con pocas líneas y de forma fácil.

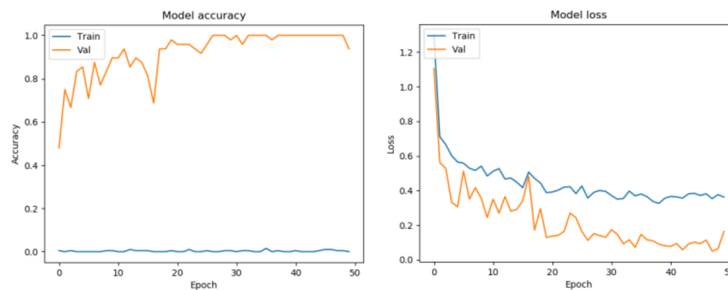


Figura 24: Gráficas de precisión y función de pérdidas

En estas gráficas se pueden visualizar los valores que han ido tomando la precisión y la función de pérdida tanto en validación como durante el entrenamiento.

En este caso la gráfica de la precisión no es muy útil porque esta gráfica se dibujo con datos sobre un entrenamiento usando mixup, lo que deforma mucho la precisión durante el entrenamiento dado que el mixup combina los valores lo que provoca que estos pasen a ser valores entre 0 y 1 y la red no pueda dar con esos valores exactos, pero como se puede ver durante a validación los resultados van creciendo progresivamente. En la gráfica de pérdida se puede ver como los valores de entrenamiento y validación empiezan estando muy cerca y se van separando mientras avanza el entrenamiento, en este caso esto es un funcionamiento correcto de la red.

Dado que la cantidad de datos que se tenían no era muy grande, para asegurarse de que a la hora de entrenar los resultados fuesen más fiables se ha utilizado *k-fold validation*, que como se explicó en el punto 2.4.3, lo que hace es crear un número de paquetes donde en cada uno se utilizan un conjunto de muestras y de testeo distintos. De esta forma se realiza un entrenamiento por paquete, para este caso se utilizaron 10 paquetes que suele ser lo más usual, de esta forma teníamos 10 paquetes que entrenar y por lo tanto 10 funciones resultados distintos, estos resultados se tratan después en el testeo.

Para realizar esto, se utilizó tanto en dos y en tres dimensiones la librería *scikti-learn*, que proporciona métodos para realizar los paquetes de una forma fácil, para el caso de dos dimensiones se utilizó el método KFold, que simplemente según el orden proporcionado en los datos separa los paquetes, por esto se hacía una permutación aleatoria durante el tratamiento de datos, durante la realización de los archivos para el modelo en tres dimensiones se descubrió el método StratifideKFold, que a diferencia del primero, al separar los datos entre entrenamiento y test, en estos últimos haya una distribución balanceada de las clases en cada paquete, así el testeo será más fiable, por esto mismo durante el tratamiento de los datos para tres dimensiones no se realiza ningúna permutación aleatoria.

La duración del entrenamiento varía según el procesador del ordenador, el tamaño de los datos, la cantidad de datos, la cantidad de epochs, el número de batch y la utilización de *k-fold cross validation*, que en este caso incrementa el tiempo de entrenamiento por 10 ya que tras finalizar todos los epochs para una red

empieza otro entrenamiento para otra y así hasta entrenar sobre los 10 paquetes, pudiendo llegar a durar desde unos pocos minutos a horas o días según estos parámetros.

3.3.5. Testeo del modelo

Después del entrenamiento hay que testear el modelo, que es un proceso simple y más rápido que los anteriores.

Para esto *Keras* tiene una serie de herramientas que facilitan esto, utilizando el método predict, una vez cargado el modelo que se va a probar, se le inserta como parámetros los datos que tiene que predecir, tras realizar la predicción el método devuelve un array con los valores predichos.

La implementación en dos y tres dimensiones es un poco distinta dado que los datos para el segundo caso tienen una mayor dimensión espacial y hay que pasárselos de uno en uno por lo que se tuvo que ir insertando estos en un array, en el primer caso se le podía pasar una lista con todos los datos y este devolvía un array con todos los valores predichos.

En cualquiera de los casos tras tener la lista resultante de la predicción se hace una comparación con una lista con las etiquetas que deberían haber salido, con estos datos se calcula la precisión y se pueden calcular la especificidad y la sensibilidad.

Como se ha utilizado *K-fold cross-validation* al testear se prueban todos los modelos y se hace una media de la precisión para ver la precisión general entre todos los paquetes.

Como último añadido durante la fase final de experimentación se incluyó el *bayesian dropout*, que activa las capas de dropout durante test. Dado que el dropout es un proceso que desactiva neuronas de manera aleatoria se realiza una función a la que se le pasa como parámetros los datos, el modelo y un número de iteraciones, que indica cuantas veces se va a hacer la predicción y tras eso se hace una media del resultado según el número de iteraciones, esto mejora el resultado en algunos casos como veremos durante la experimentación.

```
Fold- 9
best_model3D9.h5
[1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1]
Precision:  0.9629629629629629
Fold- 10
best_model3D10.h5
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0]
Precision:  0.9629629629629629
Precisión media:  0.9888888888888889
Sensibilidad:  0.9851851851851853
Especificidad:  0.9926470588235294
```

Figura 25: Salida por pantalla del testeo.

Para visualizar los datos se sacan por pantalla el paquete(*fold)* que se está testeando, el nombre del modelo, la lista de datos deseados, debajo de esta, la lista

con los valores predichos por la red y debajo o arriba la precisión del modelo. Tras finalizar todos los folds se muestran al final la precisión media y en algunos casos la sensibilidad y la especificidad.

4. Experimentación

En este capítulo se exponen los resultados de los experimentos realizados durante el desarrollo del trabajo. Como se ha comentado la experimentación se dividió en dos partes, un acercamiento con un modelo en dos dimensiones ya que comenzar trabajando con datos en tres dimensiones podía acarrear que en las primeras instancias se tardara mucho tiempo en encontrar los valores óptimos de entrenamiento dado que el tamaño de estos datos hace muy lento el entrenamiento llegando a multiplicar por más de 10 el tiempo necesario para acabar uno. Por esta razón se optó por simplificar el problema comenzado por una versión en dos dimensiones. Otra razón para simplificar el caso fue comenzar a familiarizarse con las herramientas que se iban a utilizar y crear un método óptimo de trabajo.

Por lo que se comenzará exponiendo la experimentación sobre las dos dimensiones con los resultados y conclusiones que se sacaron de ella, al acabar esta se seguirá con el caso en tres dimensiones en el cual se explicará brevemente como se adaptó el modelo y presentarán los resultados obtenidos de los entrenamientos realizados a medida que se cambiaban algunos parámetros y se utilizaban nuevos métodos tanto durante el entrenamiento y el testeo.

En última instancia se indicarán algunos de los problemas que aparecieron durante el desarrollo del trabajo y como se solucionaron.

4.1. Toma de contacto en dos dimensiones

En este punto se explica como fue la primera toma de contacto con las herramientas empezando por un caso simplificado en dos dimensiones, de esto se aprendieron conceptos teóricos y prácticos que ayudaron a mejorar la experimentación en tres dimensiones.

4.1.1. Desarrollo de la experimentación

Como se explica en el punto 3.3.2 para la obtención de los datos en dos dimensiones se utiliza una sección de los datos originales en tres dimensiones mediante un script de Matlab.

Una vez obtenidos los datos se comenzó la documentación para crear el modelo de red adecuado, para esto se utilizo la API de *Keras* donde existen algunos ejemplos y otras paginas con distintas formas de utilizar redes convolucionales, tras esto se empezó la implementación a base de prueba y error con la ayuda del tutor que ya tenía experiencia en este campo.

Uno de los primeros problemas encontrados fue el no saber si la red tenía la suficiente capacidad para solucionar el problema(*underfitting)* para esto se utilizo un método llamado *babysitting,* en comprobar con pocos datos si consigue solucionar el problema, empezando por un solo dato, esto se comprobó porque al principio la red no subía del 30% de aciertos, tras esto se pudo ver que si acertaba con pocos datos. Por lo que en segundo lugar se comprobó la calidad de los datos lo

que dio lugar a encontrar que había datos de buena calidad clasificados como mala calidad lo que confundía a la red y no conseguía encontrar los patrones que diferenciaran unas imágenes de otras. Tras esto la red empezó a mejorar los resultados llegando hasta más del 60%.

Haciendo un análisis de los datos, se veía que mientras el valor de la función de pérdida del entrenamiento bajaba, la de validación subía por lo que se estaba dando un caso bastante claro de overfitting. Se supuso que esto era dado por la poca cantidad de datos y el desbalanceo de los mismos, habiendo más del doble de muestras de buena calidad que de mala calidad.

También se añadió la utilización del *k-fold cross validation* para evaluar la red a la hora de realizar predicciones sobre los datos con lo que no fue entrenada usando todos los datos posibles, de esta forma se entrena la red varias veces, 10 en este caso, y se realiza un promedio de las precisiones de todas.

```
Fold- 8
best_model8.h5
Test loss: -0.024703970461180717
Test accuracy: 0.9090909090909091
[1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1]
Fold- 9
best_model9.h5
Test loss: 0.930134097735087
Test accuracy: 0.48484848484848486
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

Figura 26: Ejemplo de resultados con overfitting.

Esta imagen pertenece a la red cuyo resultado fue de una precisión promediada del 85% donde en algunas redes la predicción aparentaba ser muy buena, pero en otras era realmente mala, llegando a tener precisiones menores al 50% como se puede ver en el *fold-9* de la imagen. Esto es un caso típico de overfitting dado que la red ha generalizado el problema a unos casos concretos y no clasifica bien.

Para solucionar esto se probó a utilizar dropout entre cada nivel de convolución, empezando con un valor de 0.5 que indica la probabilidad de una neurona de ser desactivada, lo que hace este es que cada vez que se entrena una red, la estructura interna de esta varía por lo que es como entrenar varias redes distintas. Esto produjo una mejoría en los resultados donde se alcanzó un máximo del 91,96%, este resultado fue muy bueno, pero dado a la poca experimentación que se había realizado hasta el momento se prosiguió realizando variaciones en los parámetros.

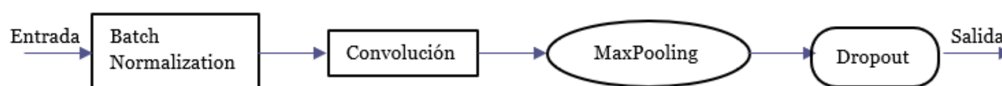


Figura 27: Ejemplo de un nivel de convolución.

La figura 9 muestra un esquema de los niveles de convolución, por cada nivel se realiza un batch normalization para normalizar los datos de entrada, una convolución para analizar los rasgos de las imágenes, seguido de una capa de *maxpooling* para reescalar los datos, los cuales se dividían en dos con una entrada (2,2) indicando el tamaño del kernel a utilizar y una capa de dropout para desactivar algunas neuronas y rebajar el overfitting, tras un nivel de estos podía venir otro nivel igual pero con los filtros de convolución de mayor tamaño o si es el último nivel realizar un último *batch normalization* para seguir con una capa flatten y acabar con el clasificador(*dense*).

Otro parámetro que se estuvo probando fue el de funciones de pérdida(*losses*) en principio se empezó con *binary cross entropy* dado que las muestras están separadas en dos clases, y está función está preparada para funcionar con dos clases, también se probó con *categorical cross entropy* por intentar otro acercamiento pensando que tal vez podría ser más potente que resultó no producir muchos cambios, y por último se utilizó una función llamada *mdice1_loss*, que el tutor había hecho, esta se basa en la función *dice coefficient* que tiene como aspecto positivo la posibilidad de darle más valor a una clase en la función de perdida por lo que es muy útil con datos mal balanceados como es el caso. Esta última función dio muy buenos resultados.

4.1.2. Resultados más relevantes

En la *Tabla 1* NC significa Niveles de Convolución que correspondería con los niveles mostrados en la *Figura 29,* CPN es Convoluciones Por Nivel, CCE es *Categorical Cross Entropy* y BCE *Binary Cross Entropy*.

Nº	Red neuronal					Precisión (%)
	Losses	**Dropout**	**NC**	**CPN**	**Parámetros**	
1	mdice1_loss	0.5	5	3	50M	**91.36**
2	CCE	0.25	4	3	50M	87.86
3	BCE	0	5	3	50M	85.39
4	BCE	0.5	5	2	40M	78.85

Tabla 1: Resultados más relevantes(datos promediados tras el testeo)

Estos datos hacen referencia a los mas relevantes el orden en el que están es el de precisión de mejor a peor, pero su relevancia es otra.

Siendo la red número 3 la primera realizada donde no se utilizaba dropout, tras esta se comenzó a utilizar y los resultados mejoraron como se explicó en el apartado anterior.

En la cuarta red se comprobó que utilizar dropout y bajar las convoluciones por nivel a 2 no daba buen resultado por lo que se volvieron a usar 3.

En la segunda se comprobó que utilizar *categorical cross entropy,* un dropout de 0.25 y bajar los niveles de convolución podría tener un buen resultado, pero tras más experimentaciones se vio que no se podía subir más la precisión y se volvieron a subir los niveles de convolución a 4.

La primera red fue la última en entrenarse dando un resultado bastante alto al utilizar la función mdice1_loss que en experimentos anteriores comprobó dar

buenos resultados y tras algunas pruebas se dio con que esta combinación de parámetros era los que mejor resultado daba.

4.1.3. Conclusiones

Tras realizar este primer acercamiento se vio que tratar con redes neuronales convolucionales conlleva tener en cuenta varios parámetros y que se ha de seguir en método a la hora de realizar las experimentaciones, porque si no se siguen unas pautas como son, apuntar todos los resultados y cambiar un parámetro por vez, la tarea de análisis se vuelve imposible de seguir.

Por lo que se aprendió a utilizar una serie de reglas que se seguirían a rajatabla en las experimentaciones en tres dimensiones para poder realizar un mejor trabajo y seguimiento de los experimentos.

Además, se aprendieron conceptos teóricos sobre aprendizaje profundo que no se tenían anteriormente y servirían para realizar el siguiente punto del trabajo.

4.2. Experimentación en tres dimensiones

Acabados los experimentos en dos dimensiones se pasa a resolver el objetivo principal de este trabajo con los datos originales en tres dimensiones.

En este capitulo se verá el desarrollo de la experimentación junto a algunos datos, después los resultados finales obtenidos y finalmente el modelo final elegido.

4.2.1. Desarrollo de la experimentación

Siguiendo el último modelo utilizado en el caso para dos dimensiones, se adapta este a 3 dimensiones fácilmente, en *Keras* los métodos para dos y tres dimensiones solo cambian en el número en cuestión, por ejemplo, Conv2D pasa a ser Conv3D y así con el resto de los métodos.

Como se comentaba en el *punto 3.3.1* antes de empezar a realizar el entrenamiento se buscaron más muestras para el entrenamiento con el fin de balancear estos.

Se subió el número de niveles de convolución a 6 pensando que por el tamaño de los datos actuales se iba a necesitar una red más potente, tras esto se comenzó a entrenar la red.

El primer resultado fue que la precisión de la red durante el entrenamiento no subió del 68%, resultado bastante malo teniendo en cuenta el resultado anterior en dos dimensiones. Tras esto se probó bajar el parámetro del *dropout* para solo conseguir una precisión del 72% con un valor de 0.25 en los *dropout*. Se subieron el número de convoluciones por nivel para ver si la red necesitaba más potencia, pero esto no hizo más que bajar la precisión del modelo. En la siguiente tabla se muestra como variaba la precisión según las pruebas que se hicieron al principio de la experimentación.

Nº	Dropout	CPN	Precisión (%)
1	0.5	3	68
2	0.25	3	**72**
3	0.1	3	50
4	0	3	66
5	0.25	4	52
6	0.5	4	50

Tabla 2: Primeros resultados variando dropout y Convoluciones por nivel.

Tras comprobar que los resultados no mejoraban se procedió a cambiar otros parámetros, se cambió la función de clasificación de softmax a sigmoide dado que esta podría dar un mejor resultado ya que no se necesita que el resultado sume 1.

Antes de continuar se realizan nuevamente pruebas de babysitting y se comprueba que está funcionando correctamente al llegar a una precisión del 100% con 20 muestras.

Acto seguido se probó un modelo con 6 niveles de convolución, con un *dropout* de 0.1 y 3 convoluciones por nivel, esto produjo un resultado mejoría respecto al resultado número 3 de la *Tabla 2*, que pasó a dar una precisión del 54%.

Se piensa que bajar los niveles de convolución, que produciría una red menos potente, podría mejorar el resultado así que se baja el número de convoluciones a 5, y se entrena sobre un modelo con las mismas características que el anterior salvo el nivel de convolución y se produce una gran mejoría con una precisión del 82% durante el entrenamiento. Esto parece indicar que bajar la potencia de la red puede producir una mejoría en los resultados, por lo que se bajan los niveles de convolución a 4 y se comprueba el resultado tras este cambio, produciéndose una pequeña mejora subiendo la precisión 0.35 puntos.

Viendo que esto podía llevar por buen camino se realizan varios cambios para reducir el número de parámetros que inicialmente eran 50 millones a 6 millones, los cambios fueron los siguientes:

- Bajamos el número de convoluciones por nivel dejándolo a 1
- Se añade un Maxpooling al final del ultimo nivel de convolución
- Se cambia el dense que había con un filtro de 512 parámetros a otros dos con 256 y 64 respectivamente.

Se comprueban estos cambios con un *dropout* de 0.3 y el resultado llega al 92% de precisión, esto demuestra que no se necesitaba tanta potencia para el problema que se intentaba resolver por lo que se estaba produciendo *overfitting*.

4.2.2. Resultados finales

Tras este último resultado en el capítulo anterior, damos paso a los últimos experimentos en los cuales se introdujo el método explicado en el *punto 2.4.4, mixup,* que introdujo una nueva forma de tratar con las muestras el cual, según un valor lambda dado, realiza una mezcla aleatoria de las muestras tanto de los datos de entrada como de las etiquetas respectivas a cada muestra.

Nº	Dropout final	Resto de Dropouts	Nº de filtros	Precisión (%)
1	0.2	0.2	32	87.53
2	0.3	0.3	32	86.09
3	0.1	0.2	32	**90.01**
4*	0.1	0.2	32	**93.71**
5	0	0.2	32	87.85
6	0.15	0.2	32	85.72
7	0.1	0.5	32	92
8	0.1	0.2	16	**98.51**

Tabla 3: Resultados de test de los entrenamientos finales.

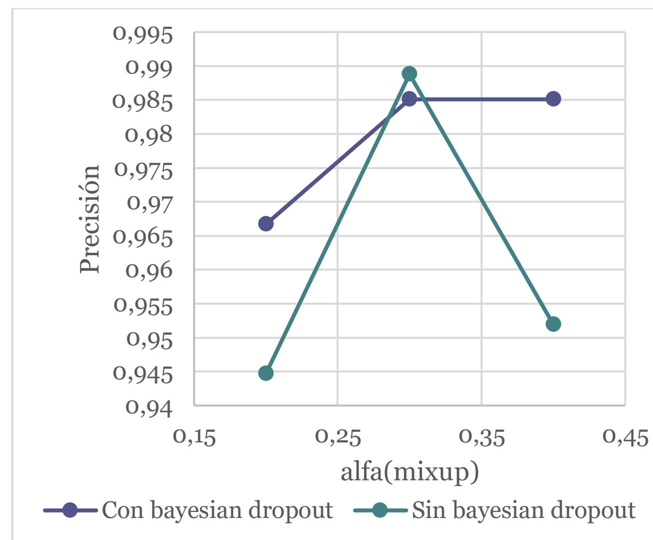
Como puede verse en la *Tabla 3* hay picos de precisión bastante altos, ya que pasamos de valores entre el 85 y 88 por ciento a valores de más del 90% incluso cercanos a 100%, todos estos valores son promedios de todos los folds durante test.

Se explicarán estos casos resaltados dado que alcanzaron las mayores precisiones tras varias pruebas. Empezando por los casos 3 y 4, los cuales son el mismo pero con el matiz que durante el testeo de estos casos, el caso número 3 dio una precisión promedio del 90.01%, pero se comprobaron nuevamente los datos y se vio que había datos que no correspondían con los de las muestras necesarias y tras depurar estos, reentrenar la red y utilizar *bayesian dropou*t, activando los dropout durante testo, el nuevo valor alcanzó un 93,71%, por lo que fue el valor más alto alcanzado hasta ese momento.

Antes del último caso se probaron otras combinaciones de dropouts como puede verse que no resultaron efectivas por lo que se volvió a los parámetros del caso 4 y se cambió el número de filtros lo que como se ve produjo otra vez una precisión promedio del 98.51% durante el testeo, siendo este el caso seleccionado como el mejor.

4.2.3. Resultado final

Una vez seleccionado el modelo final se realizan unas últimas experimentaciones sobre este cambiando el índice alfa de mixup.



Como se puede ver la gráfica al aumentar el valor de mixup de 0.2 a 0.3 la precisión con *bayesian dropout* aumentó a 98.5% mientras sin *bayesian dropout* aumentó a 98.88 y subirlo a 0.4 no afectó en la precisión con *bayesian dropout,* pero si al testeo sin bayesian dropout bajándolo a 95.19.

Estos datos demuestran por un lado que el bayesian dropout no siempre dan un resultado mayor que sin este método, dado que al parecer en el mixup con valor 0.3 la red alcanzó su valor máximo promedio del 98.88 % durante el entrenamiento y *bayesian dropout* no pudo mejorarlo.

Entonces nos quedamos con el modelo con precisión de 98.88% para el cual también se calcularon la sensibilidad y especificidad dando unos resultados de 98,51 y 99,92 respectivamente por lo que de esta forma comprobamos que para un número N de imágenes tan solo un 1,49% de las imágenes de buena calidad se clasificarán como mala calidad y solo un 0,08% de las imágenes de mala calidad se cómo buenas.

5. Conclusiones

5.1. Resultados

En este proyecto se ha implementado una red convolucional para el control de calidad de imágenes de RM. Se ha conseguido una red que obtuvo una precisión de 98.88% usando una red 3D. Si bien este resultado se puede considerar bastante bueno es cierto que el limitado número de imágenes hace que las conclusiones no sean todo lo robustas que me gustaría(aun usando k-fold). En el futuro se planea aumentar el tamaño del dataset para obtener un resultado masa confiable.

5.2. Valoración personal

La realización de este trabajo ha sido de gran satisfacción personal ha llevado más tiempo del esperado, pero finalmente se llegó a su conclusión.

Todo este tiempo ha servido como proceso de aprendizaje de tecnologías y conceptos que durante el curso se dan por encima, dado que no existe una asignatura sobre aprendizaje profundo, siendo esta una tecnología en auge con gran potencial.

Estas tecnologías son complejas y requieren de muchos recursos, lo que ha limitado en cierto modo la realización de este trabajo, que con más recursos puede llegar a mejorarse.

5.3. Posibles mejoras

El objetivo de este trabajo era el de clasificar estas imágenes en dos clases realizando un control de calidad muy básico, en un futuro se podría implementar una red neuronal convolucional que distinguiera no solo entre imágenes buenas y malas, sino que, además de esas imágenes malas pudiera distinguir entre los distintos errores que pueden aparecer como:

- Ruido
- Imágenes fantasmas(*ghosting*)
- Cortes mal realizados

Esto podría utilizarse para mejorar el sistema de clasificación de la red.

6. Bibliografía

[1] Justin Johnson, Andrej Kaparthy. CS23n. Consultado en: http://cs231n.github.io/ [Último acceso: Junio 2019]

[2] Toward Data Scince. A comprehensive guide to convolutional neural networks. Consultado en: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 [Último acceso: Junio 2019]

[3] Toward Data Scince. Activation functions in neural networks. Consultado en: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 [Último acceso: Junio 2019]

[4] Github, Kulbear. ReLU and Softmax Activation Functions. Consultado en: https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions [Último acceso: Junio 2019]

[5] Yarin gal y Zoubin Ghahramani. Bayesian convolutional neural networks with Bernoulli approximate variational inference. 2016.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A simple way to Prevent Neural networks from overfitting. 2014.

[7] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, David Lopez-Paz. mixup: Beyond Empirical Risk Minimization. 2018.

[8] Sergey Lofffe, Christian Szegedy. Batch Normalization Accelerating Deep Network Training by Reducing Internal Covariate Shift

[9] Super Data Science. Convolutional neural network(CNN): Step 3 - Flattening Consultado en: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening [Último acceso: Junio 2018]

[10] Toward Data Science. Coding neural network – Forwardpropagation and backpropagation. Consultado en: https://towardsdatascience.com/coding-neural-network-forward-propagation-and-backpropagtion-ccf8cf369f76 [Último acceso: Junio 2019]

[11] Medium. Holdout vs cross validation in machine learning Consultado en: https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f [Último acceso: Junio 2018]

[12] Encyclopedia Britannica. Nuclear magnetic resonance Consultado en: https://www.britannica.com/science/nuclear-magnetic-resonance [Último acceso: junio 2018]

[13] Chemestry libretextrs. Nuclear magnetic resonance (NMR) Spectroscopy Consultado en:

https://chem.libretexts.org/Courses/Purdue/Purdue%3A_Chem_26505%3A_Organic_Chemistry_I_(Lipton)/Chapter_5._Spectroscopy/5.3_Nuclear_Magnetic_Resonance_(NMR)_Spectroscopy [Ultimo acceso: Junio 2019]

[14]Sociedad Española de Imagen Cardiaca. ¿Qué es el T1 y T2?. Consultado en: https://ecocardio.com/documentos/biblioteca-preguntas-basicas/preguntas-al-radiologo/914-que-es-t1-y-t2.html [Ultimo acceso: Junio 2019]

[15] Javier Lafuente Martínez, Luis Hernández Moreno. Técnica de la imagen por resonancia magnética. [Último acceso: Junio 2019]

[16] ML Cheatsheet. Loss functions. https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html [Último acceso: Junio 2019]

[17] Algorithmia. Introduction to optimizers. Consultado en: https://blog.algorithmia.com/introduction-to-optimizers/ [Último acceso: Junio 2019]

[18] Sebastian Ruder. An overview of gradient descent optimization algorithms Consultado en: http://ruder.io/optimizing-gradient-descent/index.html#momentum [Último acceso: junio 2019]

[19] Michael Nielsen. Neural networks and deep learning. Determination Press, 2019