UNIVERSIDAD DE ZARAGOZA

MÁSTER EN MODELIZACIÓN E INVESTIGACIÓN MATEMÁTICA, ESTADÍSTICA Y COMPUTACIÓN.

Trabajo de fin de máster:

Redes Neuronales Convolucionales. Aspectos teóricos y aplicaciones en aprendizaje supervisado.

Inés Aldea Blasco

Directores del trabajo: José Tomás Alcalá Nalvaiz y Lidia Orellana Lozano

Septiembre 2019

Glosario

Accuracy: porcentaje de predicciones correctas respecto al total de clases posibles.

API (*application programming interface*): interfaz de programación de aplicaciones.

Backpropagation: algoritmo de retopropagación para el aprendizaje de una CNN.

Batch: lote.

Bias: sesgo.

Convolutional Neural Network (CNN): red neuronal convolucional.

Epoch: época, tiempo en el cual se procesa todo el conjunto de entrenamiento en la CNN.

Feature maps: capas de características.

Feedforward: nombre de las redes que usan la propagación hacia delante.

Flattening: proceso que agrupa y cambia la forma de las matrices de salida de la última capa de convolución por un único vector.

Forward o Forward propagation: proceso por el cual la red envía la información de entrada a través de sus capas en sentido de salida sin permitir retrocesos (propagación hacia delante).

Frame: fotograma.

Fully connected: Parte de la arquitectura de la CNN que configura el modelo de red neuronal normalmente llamada capas completamente conectadas en este ámbito.

Intranet: red informática privada interna de una institución, organización o empresa.

Loss function: función de pérdida.

Neural Network (NN): red neuronal.

One hot encoding: método utilizado en la codificación de las clases.

Padding: proceso que aumenta el tamaño de una matriz con la finalidad de que los términos del exterior influyan de igual modo que los centrales.

Pooling: etapa de la capa de convolución que reduce el tamaño de matriz.

Stride: medida de definición del avance del filtro por la matriz en el producto de convolución. También es usado para definir el avance del entorno rectangular en la etapa de pooling.

Abstract

Over the last few years, the increase in the databases and the volumen of these have made models proliferate naturally capable of processing and analyzing this data.

During this work, we will talk about one of these models: convolutional neural networks (CNNs), which are currently booming. The increase of the data mentioned above together with the technological improvements have facilitated and improved the performance of its construction, training and execution.

CNNs are a model of artificial intelligence corresponding to deep learning that is, they are models capable of extracting for themselves the information necessary to, in the case of CNNs, make a classification.

We will begin by detailing the architecture of the model defining the elements that configure it by differentiating two parts; the first in charge of extracting the characteristics of the data and, the second, more focused on the classification.

Throughout this part, we will specify the convolution layers that form the first part of the architecture, defining the convolution product for the CNN context, the activation functions and the pooling functions. In addition, we will see how the second part of the architecture is composed of a neural network model, which is responsible for classification, and the process of flattening responsible for linking both parts.

Once the model is defined, we will explain the learning process during which the parameters of the model are modified thanks to the optimization of a loss function that measures the error made by the network. This process will be carried out using the learning algorithm called backpropagation optimized thanks to the stochastic gradient descent technique.

To finish the theoretical part of the work we will make a small comment about the evaluation of the classification of these models focusing on the study of the confusion matrix.

With the purpose of greater knowledge of the model and reflecting the current state of the art, we will comment on a small collection of articles where we will show different applications as well as different data sets to which we can apply our networks.

Finally, we will use the knowledge acquired in the viability analysis of a project of automatic management of room occupancy in the Efor company. This company, belonging to the *integra* group and dedicated to providing services and technological solutions for the management, communication and marketing of companies, facilitated the logistics with which to carry out the study.

The study was developed using the Tensorflow library, using Jupyter notebook and will be presented following the real temporal evolution that has been undergoing as the work was carried out in order to better understand the changes that have been made in the model and how they have influenced the final result. During this part of the work, the preprocessing of the data set will be shown, as well as the adjustment and validation of the selection of a network architecture.

The analysis of the selected model come to the conclusion that carrying out the project for the automatic management of room occupancy by applying the convolutional neural network model is feasible because the network does yield good results in the evaluation of the model.

Resumen

En los últimos años el aumento de bases de datos y del volumen de estas han hecho que proliferen de forma natural modelos capaces de hacer frente al procesamiento y análisis de estos datos.

Durante el presente trabajo hablaremos de uno de estos modelos: las redes neuronales convolucionales (CNNs de ahora en adelante), que se encuentran actualmente en auge. El aumento de los datos mencionado más arriba junto a las mejoras tecnológicas han facilitado y mejorado el rendimiento de su construcción, entrenamiento y ejecución.

Las CNNs son un modelo de inteligencia artificial correspondiente al *deep learning* es decir, son modelos capaces de extraer por ellos mismos la información necesaria para, en el caso de las CNNs, realizar una clasificación.

Comenzaremos detallando la arquitectura del modelo definiendo los elementos que lo configuran diferenciando dos partes; la primera encargada en la extracción de las características de los datos y, la segunda, más enfocada en la clasificación.

A lo largo de esta parte, especificaremos las capas de convolución que forman la primera parte de la arquitectura, definiendo el producto de convolución para el contexto de las CNNs, las funciones de activación y las funciones de *pooling*. Además, veremos cómo la segunda parte de la arquitectura está formada por un modelo de red neuronal, que es el encargado de la clasificación, y el proceso de *flattening* encargado de enlazar ambas partes.

Una vez definido el modelo explicaremos el proceso de aprendizaje durante el cual se modifican los parámetros del modelo gracias a la optimización de una función de pérdida que mide el error cometido por la red. Este proceso se llevará a cabo mediante el algoritmo de aprendizaje llamado retropropagación optimizado gracias a la técnica del gradiente descendente estocástico.

Para finalizar la parte teórica del trabajo realizaremos un breve comentario acerca de la evaluación de la clasificación de estos modelos centrándonos en el estudio de la matriz de confusión.

Con la finalidad de un mayor conocimiento del modelo y de reflejar el estado actual del arte, comentaremos una pequeña recopilación de artículos donde mostraremos diferentes aplicaciones así como diferentes conjuntos de datos a los que podemos aplicar nuestras redes.

Finalmente, emplearemos los conocimientos adquiridos en el estudio de la fase de viabilidad de un proyecto de gestión autómata de la ocupación de salas en la empresa Efor. Esta empresa, perteneciente al grupo integra y que se dedicada a dar servicios y soluciones tecnológicas para la gestión, comunicación y marketing de las empresas, nos facilitó la logística con la que poder llevar a cabo el estudio.

El estudio se ha desarrollado utilizando la librería *Tensorflow* mediante *Jupyter notebook* y se presentará siguiendo la evolución temporal real que ha ido sufriendo conforme se iba realizando el trabajo con el fin de poder comprender mejor los cambios que se han llevado a cabo en el modelo y cómo han influido en el resultado final. Durante esta parte del trabajo se mostrará el preprocesamiento del conjunto de datos, así como el ajuste y la validación de la selección de una arquitectura de red.

El análisis del modelo seleccionado nos permite llegar a la conclusión de que realizar el proyecto de la gestión autómata de la ocupación de salas aplicando el modelo de redes neuronales convolucionales es factible ya que se alcanzan muy buenos resultados en la evaluación del modelo.

Índice general

Capítulo 1

Introducción

Hoy en día un término del que se suele hablar con frecuencia es el *Big Data*, que describe conjuntos de datos o combinaciones de ellos que, por su volumen, variedad o complejidad y velocidad de almacenamiento dificultan su procesamiento o análisis mediante técnicas convencionales. Por tanto, un problema de *Big Data* es aquel que cumple las 3 V mencionadas, volumen, variedad y velocidad. Este término ha generado una demanda en el procesamiento de estos datos de forma autómata e inteligente, cuestión que la inteligencia artificial (IA) es capaz de solventar.

La IA se puede interpretar como la incorporación de la inteligencia humana a las máquinas. Son sistemas capaces de entender, aprender o incluso razonar gracias a una serie de algoritmos o reglas estipuladas que nos lleva a pensar en ese comportamiento "inteligente".

La clave de la IA para solucionar un problema de las 3 V es el aprendizaje mediante algoritmos que lo permiten. Para entender lo que hacen estos algoritmos podemos compararlos con nuestros propios métodos de aprendizaje. En el aprendizaje supervisado, por ejemplo, premiaremos aquellas conductas que queremos que se repitan, lo que aumenta la probabilidad de que aparezcan y sancionamos las que no, para que tiendan a desaparecer.

Este aprendizaje supervisado es el que trataremos a lo largo de este trabajo, el cual requiere cierta intervención humana para hacer saber al algoritmo lo que está bien y lo que está mal. Para ello contaremos con un conjunto de entrenamiento con el que el algoritmo es capaz de obtener patrones y a través de nuestra participación, aprender. El término que hace referencia a desarrollar estas técnicas o algoritmos de aprendizaje de la IA es el *Machine Learning*.

En nuestro caso hablaremos de un tipo de técnicas de *Machine Learning* llamadas *Deep Learning*. El *Deep Learning* intenta incorporar la percepción humana a las máquinas siendo capaces de descubrir automáticamente las características que se usan para la clasificación. Técnicas como las redes neuronales o las redes neuronales convolucionales imitan el comportamiento del sistema nervioso humano mediante capas de unidades de procesamiento que llamaremos neuronas. Estas semejanzas permiten que dentro de un sistema global haya neuronas que se especialicen permitiendo una mejora en el aprendizaje.

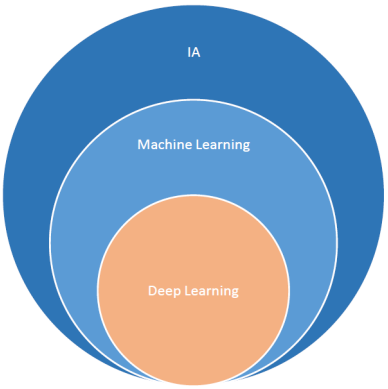


Figura 1.1: Diagrama que muestra la relación entre los términos de la inteligencia artificial.

Durante este trabajo hablaremos de las redes neuronales convolucionales (CNN). Actualmente, el uso más generalizado de este tipo de redes está en el tratamiento y clasificación de imágenes, por eso nos centraremos en él, sin embargo no es el único uso de estas redes como se mostrará en algún ejemplo más adelante.

Así como hemos adaptado el concepto de aprendizaje a la CNN, debemos adaptar el concepto de imagen ya que para un ordenador una imagen no es lo mismo que para nosotros.

Para una máquina, una imagen es una aplicación $I : C \subset \mathbb{R}^2 \to [0, 1]$ de manera que, a cada píxel de la imagen, le asignamos un número que corresponde con el color dentro de una escala de grises, donde el 0 es el negro y el 1 es el blanco, en el caso que tengamos una imagen en blanco y negro. Si la imagen es en color, la imagen de nuestra aplicación sería un vector de longitud tres, correspondiente al color rojo, verde y al azul de la imagen. Esto se debe a que las imágenes utilizan la codificación del color RGB (*red, green, blue*), ya que es posible representar un color mediante la mezcla de los tres colores primarios. En este último caso diremos que la imagen tiene tres canales correspondientes a esos tres colores.

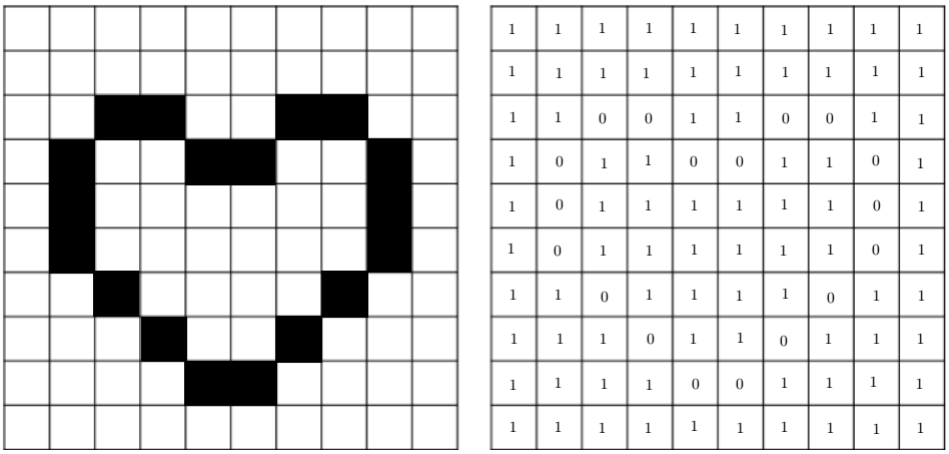


Figura 1.2: Diferencia entre la forma en la que nosotros vemos una imagen en blanco y negro (izquierda) y la forma en la que las máquinas "ven" la misma imagen (derecha).

De esta manera, al utilizar imágenes como entradas a nuestra red neuronal convolucional en realidad hablaremos y trataremos con matrices de píxeles o simplemente matrices. Si la matriz es en blanco y negro tendremos una matriz con las dimensiones de la imagen $m \times n$ y en el caso de que la matriz sea a color, tendremos una matriz para los tres canales, por tanto estaremos

tratando con una matriz de tamaño $m \times n \times 3$. Hablaremos entonces de tres dimensiones, alto, ancho y profundo. Como consecuencia, las imágenes en blanco y negro tendrán profundidad uno.

A continuación se muestra una pequeña linea de tiempo con los eventos más importantes para la evolución que han ido teniendo las CNN.

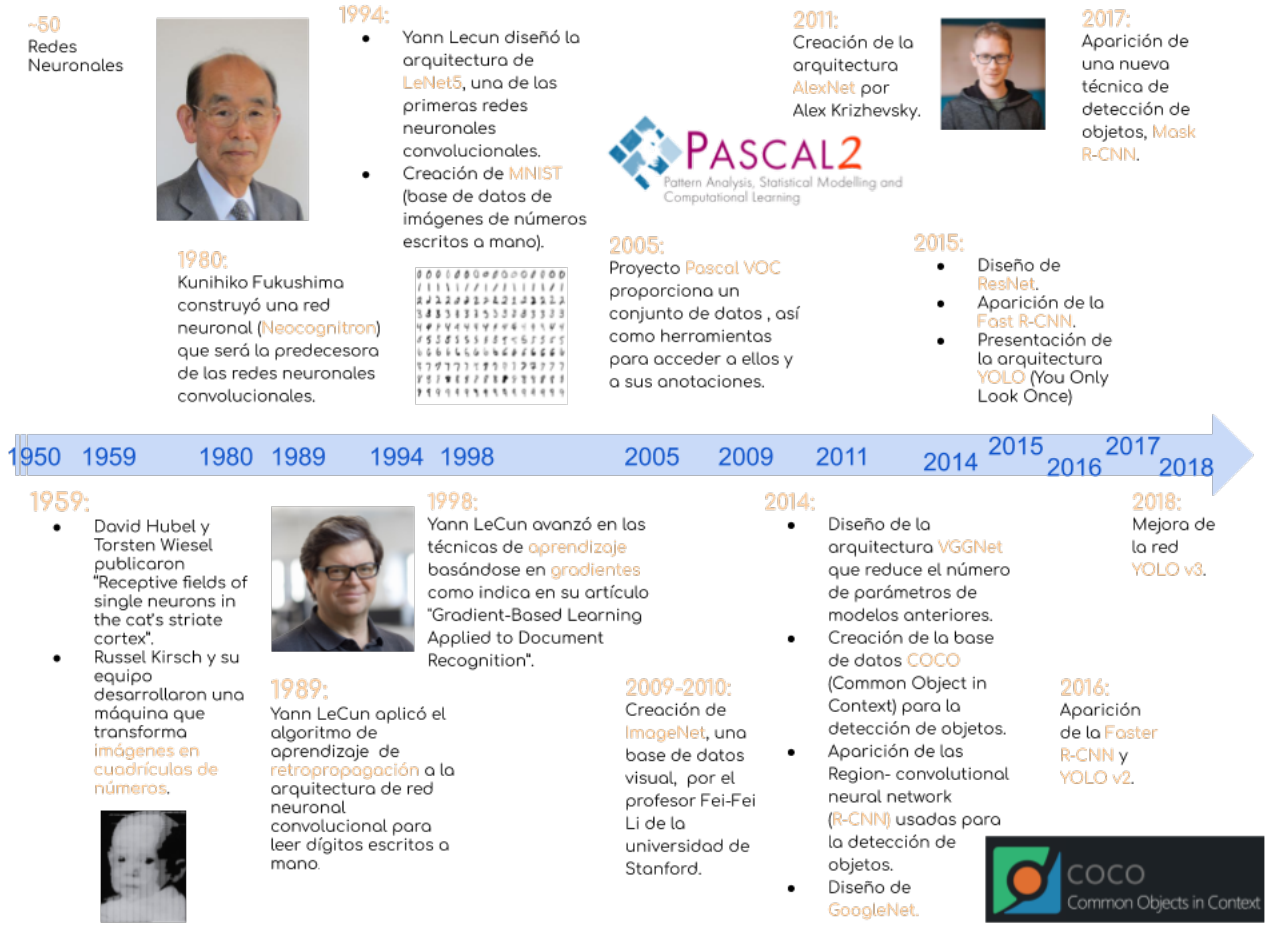


Figura 1.3: Linea de tiempo de la Evolución de las CNN.

1.1. Entorno tecnológico

Los principales entornos de trabajo más extendidos para la programación de las redes neuronales convolucionales son:

- Tensorflow (ver [23]): librería de *deep learning* desarrollada por Google basada en C++ con interfaces de programación en Python capaz de construir y entrenar redes.
- Theano (ver [24]): librería programada en Python y compilador de optimización para manipular y evaluar expresiones matemáticas, especialmente las de valor matricial. Este paquete ofrece valores de tiempo muy buenos a la hora del entrenamiento por su optimización.
- Keras (ver [13]): API de alto nivel escrito en Python capaz de ejecutarse tanto sobre Tensorflow como sobre Theano diseñado especialmente para experimentar de forma rápida y sencilla.

Para emplear estos paquetes utilizaremos *Jupyter Notebook* (ver [12]), aplicación web de código abierto que posibilita crear documentos o cuadernos que permiten incorporar elementos de código con texto narrativo con el que podremos facilitar la explicación y documentación de nuestras observaciones. Esta herramienta que permite visualizar datos, hacer simulación numérica, modelado estadístico, transformación de datos, aprendizaje automático,... es una de las más usadas en la actualidad. Grandes empresas como IBM o Microsoft utilizan esta herramienta como elemento base en sus aplicaciones o plataformas.

Jupyter Notebook cuenta con un núcleo que se encarga de ejecutar el código, en nuestro caso para código en Pyhton, aunque *Jupyter* también soporta otros lenguajes como R, scala o Julia. Otra característica de esta aplicación web es la capacidad de mostrar resultados utilizando medios enriquecidos como Latex, PNG, HTML, ... y adjuntos al código que los generó.

La facilidad de introducir notación matemática dentro de las celdas, la edición de texto enriquecido que permite no limitarnos al texto sin formato en los comentarios o el resaltado automático de sintaxis y sangría facilitan el uso y aportan limpieza a los cuadernos.

1.2. Democratización de las redes neuronales convolucionales

Grandes empresas de hoy en día han apostado por acercar estas técnicas de redes neuronales convolucionales a personas que no tienen grandes conocimientos sobre ellas ni sobre programación. Para ello han generado plataformas de servicios cognitivos a través de las cuales las máquinas son capaces de procesar información, aprender, resolver problemas,...

Dos ejemplos de esto son Watson de IBM (Ver [11]) y Azure de Microsoft (ver [17]) que se encargan de entrenar modelos, mantenerlos y actualizarlos. Unos de los servicios cognitivos que ofrecen es la visión por ordenador, donde tienen gran relevancia las redes convolucionales que vamos a estudiar. Con estas herramientas los únicos conocimientos que debemos saber son la manera de introducir los datos y la interpretación de los resultados.

Otra de las importantes ventajas que ofrecen estas plataformas son los corpus (bancos de imágenes, textos, conjuntos de datos estructurados) de los que disponen y que ponen al servicio de los usuarios. Estos grandes conjuntos de información, con los que hacen difícil la competencia, suponen grandes ventajas a la hora de entrenar los modelos como veremos más adelante.

Google es otra de las empresas que destaca por el desarrollo de TensorFlow y por la construcción de GoogLeNet (ver [22, págs 4-8]). Aunque es cierto que no es tan automático el disfrute de estas herramientas, aportan grandes ventajas en el uso de las redes neuronales convolucionales y en su investigación.

De esta manera, estos servicios cognitivos hacen de la IA un servicio que se ofrece para satisfacer ciertas necesidades capacitándola de un valor económico con el que poder comercializar.

1.3. Desarrollo del trabajo

El presente trabajo está dividido en tres grandes capítulos junto con una pequeña introducción al tema que hemos realizado a lo largo de este.

El primer capítulo tras la introducción consta de una recopilación y explicación detallada donde se recoge la teoría básica de las CNNs. En esta teoría encontramos fundamentalmente información sobre las partes de la arquitectura de las redes y, los algoritmos de aprendizaje y

su optimización. Asimismo, también se aportan una serie de referencias donde poder ampliar la información si se desea.

Durante el segundo capítulo, se presentan una serie de artículos en los cuales se exponen alguna de las posibles aplicaciones de las redes, así como una muestra de los diferentes tipos de datos de entrada y algún ejemplo del pretratamiento de estos.

Finalmente, en el último capítulo, se aplicarán los conocimientos adquiridos en la creación del trabajo para resolver un problema real sobre la gestión de salas en la empresa Efor. Veremos además, como esta técnica es realmente eficaz alcanzando muy buenos resultados.

Capítulo 2

Redes Neuronales Convolucionales

Como ya hemos mencionado en la introducción, las redes neuronales convolucionales son una técnica del *Deep Learning* que, además, incorpora redes neuronales en su estructura. Esta técnica es la que desarrollaremos a lo largo del presente trabajo desde una perspectiva tanto teórica como práctica.

Una red neuronal artificial es un modelo matemático que emula de manera simplificada el funcionamiento de las neuronas cuya función principal es recibir, procesar y transmitir información. Se denomina arquitectura de una red neuronal a la estructura o patrón de conexión de la red. Esta estructura tendrá un elemento esencial al que hemos llamado neurona la cuál se organizará en capas. Veremos como si nos abstraemos de la inspiración biológica, las redes neuronales se pueden ver como una serie de operaciones matemáticas sobre una lista de números que da como resultado otra lista de números.

Las redes neuronales convolucionales son redes neuronales que utilizan la convolución en lugar de la multiplicación usual de matrices en al menos una de sus capas. Su arquitectura se puede dividir en dos partes diferenciadas: las capas de convolución que son las encargadas de extraer los patrones y las características de los datos que introducimos a la red, y la capas formadas por la red neuronal que son las encargadas de utilizar la información obtenida para clasificar.

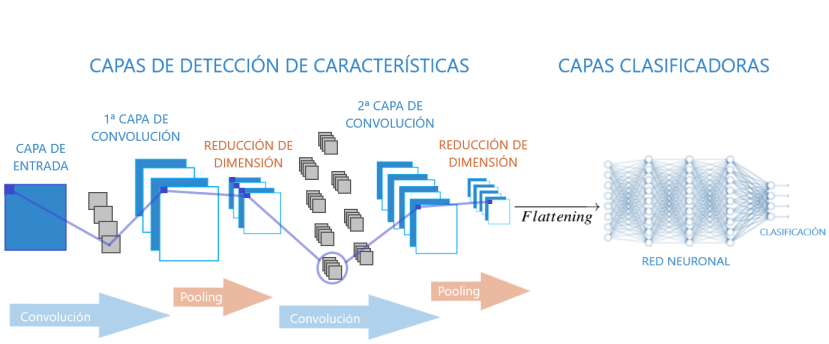


Figura 2.1: Ejemplo de una estructura de red neuronal convolucional.

De este modo, las CNN son clasificadores, su propósito es asignar a un elemento de entrada una categoría o clase conocida gracias a la información que ella misma adquiere mediante un algoritmo. Para alcanzar este objetivo, se han de llevar a cabo las siguientes fases.

Se comienza fijando las características del modelo, en nuestro caso, el diseño de la arquitectura de nuestra red. Durante la fase de entrenamiento, se contará con un conjunto de datos similares a los que se quieren predecir, clasificados de forma correcta, con el fin de ir modificando el valor de los parámetros del modelo para que pueda desarrollar correctamente su función. Terminado el entrenamiento, se procede a la validación del modelo con el conjunto de validación, un conjunto similar al de entrenamiento pero formado por datos diferentes y en menor cantidad que junto con el conjunto de entrenamiento formarán el conjunto de datos inicial. La finalidad de esta fase es estimar la precisión de la clasificación con datos que no hayan sido utilizados para el entrenamiento. Si aceptamos la precisión de la red, ya puede ser utilizada para realizar la tarea para la que fue definida, si por el contrario no lo hacemos, tendremos que cambiar las características del clasificador y volver a realizar todo el proceso o, realizar otro entrenamiento cambiando el conjunto de entrenamiento o aumentando el número de iteraciones.

Durante la parte teórica de este trabajo seguiremos este esquema: explicaremos la arquitectura, tanto el proceso que se realiza como las características de esta; el algoritmo por el cual la CNN va aprendiendo y, finalmente, el estudio de la precisión de la red.

2.1. Arquitectura de las redes neuronales convolucionales

El diseño de una red neuronal convolucional no es una tarea fácil. La elección del número de capas, los tipos o las conexiones no sigue un patrón definido, ni existe un proceso específico que ayude a la definición. Se utiliza la "*prueba y error*" como el método heurístico para ir probando alternativas y comprobando que funcionan.

Para abordar este problema lo que se suele hacer es utilizar aquellas redes que han demostrado una alta capacidad de aprendizaje llegando a buenas tasas de exactitud. Algunas de estas redes son AlexNet, GoogleNet, VGG o la ResNet pudiendo encontrar un resumen de sus características en [14].

La arquitectura de una red se estructura en la concatenación de una serie de capas, llamando capa de entrada a la capa que incorpora nuestros datos a la red, la capa de salida donde se obtienen los resultados y una serie de capas intermedias a las que llamaremos capas ocultas.

En estas capas podemos distinguir dos partes diferenciadas. Las primeras capas de la arquitectura son las capas de convolución, que son las encargadas de detectar e identificar características o patrones del conjunto de datos que introducimos en la red neuronal convolucional. Después seguirán las capas que forman una red neuronal que se encargan de clasificar los datos utilizando la información extraída en las capas anteriores.

A continuación vamos a explicar las características de estas capas donde he seguido los siguientes textos [2], [10] y [8] fundamentalmente.

2.1.1. Capas de convolución

Las capas de convolución son las que hacen convolucional a una red neuronal. Durante las primeras capas de convolución se detectan patrones simples como líneas, figuras geométricas simples como círculos o bordes. Al realizar la concatenación de varias de estas capas y, por tanto, aumentar la profundidad de la arquitectura, la red es capaz de mezclar esta información para ir aprendiendo conceptos cada vez más complejos.

Una capa de convolución está formada por tres etapas. La primera, donde se realiza la operación de la convolución, una segunda donde se aplica una función de activación y por último, la aplicación del *pooling*. No obstante las dos últimas son optativas en el diseño de la arquitectura.

Convolución

La convolución es un operador matemático que transforma dos funciones f y g en una tercera donde se representa la magnitud en la que se superponen g y una versión trasladada de f. Se denota $f * g$ y generalmente se define como:

$$(f * g)(\mathbf{x}) = \int_D f(\mathbf{x\text{-}t})g(\mathbf{t})\mathrm{d}t.$$

Sin embargo, el producto de convolución en el ámbito de las CNNs no sigue la definición mencionada anteriormente. Se utiliza la siguiente variación de la convolución que definiremos para dimensión dos:

$$(f \star g)(x_1, x_2) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x_1 + t_1, x_2 + t_2)g(t_1, t_2)\ \mathrm{d}t_1\mathrm{d}t_2.$$

Podemos comprobar como efectivamente la definición formulada es una convolución realizando el cambio de variable $\mathbf{t} = -\mathbf{y}$.

$$(f \star g)(x_1, x_2) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x_1 + t_1, x_2 + t_2)g(t_1, t_2)\ \mathrm{d}t_1\mathrm{d}t_2 =$$

$$= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x_1 - y_1, x_2 - y_2)g(-y_1, -y_2)\ \mathrm{d}y_1\mathrm{d}y_2 = (f * h)(x_1, x_2).$$

Considerando $h = g(-\mathbf{t})$.

Por lo general, las entradas a nuestras redes neuronales serán datos discretos, lo que nos llevará a usar la definición discreta de la convolución:

$$(f \star g)(x_1, x_2) = \sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} f(x_1 + t_1, x_2 + t_2)g(t_1, t_2).$$

Una vez definido el producto que usaremos en las capas de convolución, vamos a explicar el proceso que se sigue en ellas. Este proceso consiste en multiplicar una matriz de pequeñas dimensiones a la que llamaremos **filtro o Kernel** por submatrices de la que llamaremos matriz de entrada a la capa o simplemente matriz mediante la convolución discreta que acabamos de definir,

$$C_1(i, j) = (I \star K)(i, j) = \sum_{m=0}^{\overline{m}-1}\sum_{n=0}^{\overline{n}-1} I(i + m, j + n)K(m, n),$$

donde, adaptando la definición a nuestro entorno, I es una submatriz de la matriz de entrada y K es la matriz del filtro, ambas de dimensiones $\overline{m} \times \overline{n}$.

Durante el proceso de convolución, el filtro va recorriendo la matriz para hacer el producto de convolución que hemos definido. Para regular este movimiento se define el ***stride*** que es la medida que define el avance del filtro por la matriz.

Veamos un pequeño ejemplo para aclarar y visualizar como se transforma la entrada al convolucionar con el filtro. Supongamos que queremos convolucionar la matriz de letras con el filtro $\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}$ y *stride*=1, la operación se realiza como sigue:

$$\begin{pmatrix} \boxed{\begin{matrix} a & b & c \\ f & g & h \\ k & l & m \end{matrix}} & d & e \\ i & j & \\ n & o & \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & \\ & \end{pmatrix} \quad \left| \quad \begin{pmatrix} a & \boxed{\begin{matrix} b & c & d \\ g & h & i \\ l & m & n \end{matrix}} & e \\ f & j & \\ k & o & \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & b-d-l+n \\ & \end{pmatrix} \right.$$

$$\left| \ \dots \ \right| \ \dots \ \left| \quad \begin{pmatrix} a & b & c & d & e \\ f & g & \boxed{\begin{matrix} h & i & j \\ m & n & o \\ r & s & t \end{matrix}} \\ k & l & \\ p & q & \\ u & v & w & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & b-d-l+n & c-e-m+o \\ f-h-p+r & g-i-q+s & h-j-r+t \\ & & \end{pmatrix} \quad \right| \ \dots$$

$$\begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & b-d-l+n & c-e-m+o \\ f-h-p+r & g-i-q+s & h-j-r+t \\ k-m-u+w & l-n-v+x & m-o-w+y \end{pmatrix}$$

Haciendo el cálculo un poco más detallado de un elemento de la matriz resultante tenemos,

$$C_1(1,2) = \sum_{m=0}^{2}\sum_{n=0}^{2} I(1+m, 2+n)K(m,n) = I(1,2)K(0,0) + I(2,2)K(1,0) + I(3,2)K(2,0) +$$

$$+I(1,3)K(0,1)+I(2,3)K(1,1)+I(3,3)K(2,1)+I(1,4)K(0,2)+I(2,4)K(1,2)+I(3,4)K(2,2) =$$

$$= h\cdot 1 + m\cdot 0 + r\cdot(-1) + i\cdot 0 + n\cdot 0 + s\cdot 0 + j\cdot(-1) + o\cdot 0 + t\cdot 1.$$

Podemos aplicar el mismo filtro a la entrada pero esta vez modificando la forma en la que el filtro recorre la imagen con un *stride*=2 para ver como influye en el producto.

$$\begin{pmatrix} \boxed{\begin{matrix} a & b & c \\ f & g & h \\ k & l & m \end{matrix}} & d & e \\ i & j & \\ n & o & \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & \\ & \end{pmatrix} \qquad \begin{pmatrix} a & b & \boxed{\begin{matrix} c & d & e \\ h & i & j \\ m & n & o \end{matrix}} \\ f & g & \\ k & l & \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & c-e-m+o \\ & \end{pmatrix}$$

$$\begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ \boxed{\begin{matrix} k & l & m \\ p & q & r \\ u & v & w \end{matrix}} & n & o \\ & & s & t \\ & & x & y \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & c-e-m+o \\ k-m-u+w & m-o-w+y \end{pmatrix} \qquad \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & \boxed{\begin{matrix} m & n & o \\ r & s & t \\ w & x & y \end{matrix}} \\ p & q & \\ u & v & \end{pmatrix} \star \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a-c-k+m & c-e-m+o \\ k-m-u+w & m-o-w+y \end{pmatrix}$$

Observamos que el producto de convolución reduce la dimensión de la matriz que convolucionamos. Ese cambio está influido por el tamaño del filtro y por el *stride* que se aplica en la convolución. Supongamos que la dimensión de la matriz que convolucionamos es $I_1 \times I_2$ y la del filtro es $\overline{m} \times \overline{n}$, la dimensión resultante al hacer la multiplicación por el filtro es:

$$\frac{I_1 - \overline{m}}{stride} + 1 \times \ \frac{I_2 - \overline{n}}{stride} + 1. \tag{2.1}$$

En ocasiones, esta reducción del tamaño en la matriz de entrada puede generar desventajas como la limitación de la profundidad de la arquitectura en las capas convolucionales. Otra de

las desventajas está relacionada, a su vez, con la definición del producto de convolución porque los elementos de los bordes de las matrices influyen en menor medida que los situados en las zonas centrales, esto puede producir un efecto de perdida de información dependiendo de la estructura de los datos.

Si se desea solventar, se utiliza el método conocido como *padding*. Consiste en aumentar el tamaño de la matriz para que el Kernel pueda acceder a los elementos de los bordes de las matrices y mantener esa información. Estos elementos añadidos pueden tomar diferentes valores; cero, los valores del borde opuesto o la extensión de los valores del propio borde son algunos ejemplos.

Si evadimos la abstracción matemática de este concepto, la convolución tan solo es la aplicación de un filtro a la imagen como cualquier filtro determinado que aplicamos cuando las retocamos. Como ya hemos dicho, mediante estos filtros la CNN extrae la información sobre ciertas características de las matrices para luego aprender de estas.

Mediante la herramienta *GIMP* (ver [5]), que es un programa de edición de imágenes digitales, podemos ver las transformaciones de los filtros en las imágenes ya que nos permite aplicarlos introduciendo la matriz con la que queremos convolucionar.

Estos son algunos ejemplos de cómo cambia la imagen al convolucionar con los siguientes filtros:

- Filtro para la detección de bordes.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Figura 2.2: Cambio de una imagen al convolucionar por un filtro que detecta los bordes.

- Filtro para un mayor enfoque.

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Figura 2.3: Cambio de una imagen al convolucionar por un filtro que provoca un mayor enfoque.

Función de activación

Tras la etapa de convolución, a cada elemento de la matriz resultante al que llamaremos **neurona**, se le aplica una función de activación. Esta etapa sirve para detectar las características dando lugar a las matrices llamadas *feature maps* o mapas de características. Estas funciones deciden como modificar la información antes de seguir procesándola. De manera similar a nuestras neuronas que se activan si se debe seguir transmitiendo el impulso nervioso, las neuronas artificiales se activan o no gracias a estas funciones para seguir transmitiendo la información si se activan, o cero en caso contrario.

Una aplicación importante de estas funciones es aportar la no linealidad si nuestro conjunto de datos lo requiere ya que el producto de convolución es lineal. Una de las más utilizadas es la función de activación **ReLU** (*Rectified Linear Unit*) que puede definirse por,

$$\phi(x) = \text{máx}(0, x),$$

siendo x el valor de la neurona tras la convolución. Podemos observar como la función de activación solo hace que se activen las neuronas en el caso de salidas positivas.

Otra función bastante utilizada, para la capa de salida de la red neuronal, es la función **Softmax**. Esta función calcula la distribución de probabilidades del elemento de entrada sobre n clases diferentes,

$$\phi_i(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{j=1}^{n} e^{x_j}} \text{ para } i = 1, ..., n \text{ con } \mathbf{x} \in \mathbb{R}^n.$$

Como nos devuelve un vector de cantidades comprendidas en (0,1] se utiliza para estimar la probabilidad de pertenecer a las clases que queremos clasificar.

Estas son otras funciones que pueden ser utilizadas como funciones de activación:

- **Función sigmoide** (*Sigmoid Function*):

$$\phi(x) = \frac{1}{1 + e^{-x}}.$$

  Toma el valor real de entrada y genera otro entre 0 y 1.

- **Función tangente hiperbólica** (*Hiperbolic Tangent Function*):

$$\phi(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}.$$

  Esta función es similar a la función sigmoide con las diferencias de que los valores reales los convierte al rango [-1,1] y su salida está centrada en el 0.

- **Función ELU** (*Exponential Linear Unit*):

$$\phi(x) = \begin{cases} x & si\ x > 0, \\ a(e^{x} - 1) & en\ \ otro\ \ caso. \end{cases}$$

  Es similar a la función ReLU, pero en las entradas negativas se vuelve suave hasta que su salida es igual a $-a$ y puede producir salidas negativas.

Durante la sección de Aprendizaje explicaremos en profundidad la importancia de estas funciones y sus características particulares.

Pooling

La etapa de *pooling* consiste en reemplazar la salida de la capa de convolución por un resumen estadístico de las salidas cercanas reduciendo su dimensión.

Durante esta etapa se define un entorno rectangular que recorre la matriz de entrada de manera similar a la del filtro para luego aplicar el resumen estadístico deseado. Hay diferentes formas de realizar esta operación, aunque la más utilizada es la ***max pooling*** que consiste en calcular el máximo de los valores del entorno definido. Otras de las opciones utilizadas son *sum pooling* que consiste en la suma de los datos del entorno definido, *average pooling* con la media de los datos del entorno, una media ponderada basada en la distancia con el píxel central o la norma L^2.

Para definir el proceso se debe detallar las dimensiones del entorno rectangular y el *stride* con el que se mueve, en este caso, el entorno rectangular. Tras este proceso la dimensión de la salida sigue la fórmula (2.1) pero en vez de calcularse con las dimensiones del filtro serán las dimensiones del entorno definido.

Si queremos, por ejemplo, aplicar a la siguiente matriz la etapa de pooling utilizando max pooling de dimensiones 2×2 y *stride*=2, obtenemos como resultado:

$$\left(\begin{array}{cc|cc} 12 & 20 & 15 & 0 \\ 8 & 7 & 4 & 3 \\ \hline 15 & 35 & 34 & 7 \\ 100 & 59 & 75 & 13 \end{array}\right) \xrightarrow{Max\ Pooling\ 2\times 2\ stride=2} \left(\begin{array}{c|c} 20 & 15 \\ \hline 100 & 75 \end{array}\right)$$

Si nos fijamos, el pooling realizado con las características del ejemplo reduce las dimensiones de la entrada a la mitad.

Esta etapa mantiene las características obtenidas en las etapas anteriores aunque puede modificar su lugar, de manera que, gracias a este proceso, se captura la invarianza espacial de los datos, es decir, por ejemplo, la posición o un pequeño cambio en la forma de un objeto no confunden a la red o hace que pierda información importante.

Las capas de convolución son la concatenación de estas etapas, con la aclaración que en capa de convolución no solo se utiliza un único filtro sino que se aplican varios de ellos a la vez, ocasionando que en cada capa, paralelamente, se realicen las etapas con las mismas características a excepción de los elementos del filtro, haciendo que la arquitectura tenga apariencia de red.

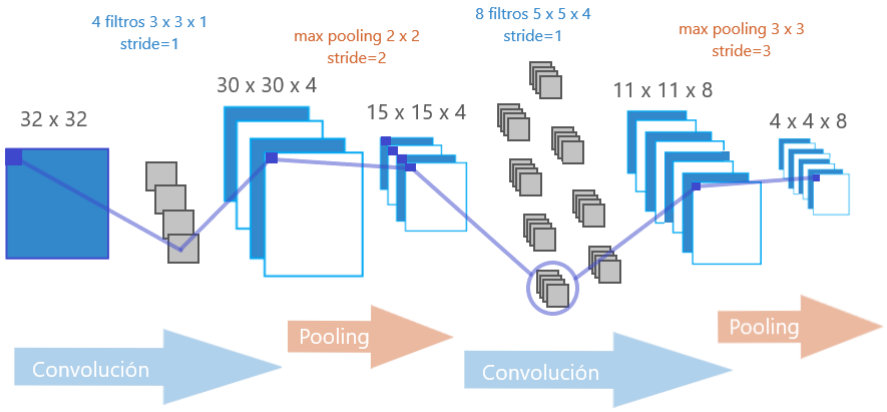


Figura 2.4: Transformación de una imagen en dos capas de convolución según las dimensiones y características de dichas capas.

Aparentemente podemos pensar que el producto de convolución funciona de manera diferente a la definición planteada si el filtro utilizado y la matriz con la que se convoluciona no tiene profundidad 1, sin embargo, lo que ocurre es que cada elemento de la matriz resultante es influido por la suma de las convoluciones de las matrices con las que se realiza la convolución en ese mismo lugar tal y como las hemos definido. Hay ocasiones en las que se añade a esa suma un término llamado *bias* o **sesgo** para poder modificar los valores de entrada a la función de activación. De esta manera podremos formular el producto de una matriz I con dimensión $I_1 \times I_2 \times p$ y un filtro K de $a \times b \times p$ como

$$C(i,j) = \sum_{q=1}^{p} C_q(i,j) + b.$$

Observamos que la matriz resultante del producto de convolución tiene profundidad uno. Para el resto de dimensiones se respetará la fórmula de las dimensiones (2.1).

Tras las capas convolucionales, se debe "aplanar" la salida de la última capa oculta de esta parte convolucional para que deje de ser tridimensional y así poder enlazarla con la red neuronal. Esta transformación se realiza gracias al proceso llamado ***flattening*** el cual extrae cada una de las líneas de píxeles de cada matriz y las alinea.

$$\begin{pmatrix} 20 & 15 \\ 100 & 75 \end{pmatrix} \xrightarrow{Flattening} \begin{pmatrix} 20 \\ 15 \\ 100 \\ 75 \end{pmatrix}$$

2.1.2. Capas de la red neuronal

Después de las capas de convolución se utilizan capas completamente conectadas (*fully connected*) que forman una red neuronal convencional. Estas capas serán las encargadas de utilizar la información adquirida durante las capas de convolución para clasificar la información de nuestro conjunto de datos.

Cada elemento del vector obtenido con el proceso de *flattening* formará la entrada a nuestra red neuronal. Como consecuencia, tendremos tantas entradas como longitud de ese vector.

La labor de la neurona consiste en, dado un conjunto de entradas $\{x_i\}_{i=1}^{n}$, calcular una suma ponderada con una serie de pesos $\{\omega_i\}_{i=1}^{n}$ que reflejan las conexiones entre las neuronas. Estos pesos determinan el efecto de la entrada en la neurona. Del mismo modo que habíamos añadido un sesgo tras el proceso de convolución, podemos añadirlo en esta suma ponderada de manera que este valor ω_{n+1} no estará influido por ninguna neurona.

Otra forma de ver esta suma ponderada es verla como un producto de matrices donde W es una matriz de dimensión $1 \times n$ con los pesos y $\mathbf{x}$ de dimensión $n \times 1$ con las entradas a la red neuronal.

Finalmente, se aplica una función de activación ϕ que, del mismo modo que en las capas de convolución, se encarga de transmitir la información si se activa o cero en caso contrario. Estas funciones de activación pueden ser las mismas que los ejemplos mostrados en la explicación de las capas de convolución y su resultado establecerá el valor de salida de la neurona.

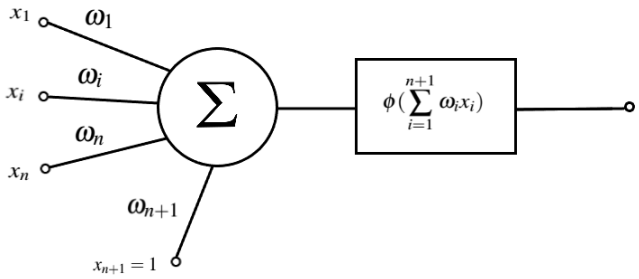


Figura 2.5: Estructura de una neurona

Cada capa oculta de la red neuronal está formada por un conjunto de neuronas. Cada una de estas tiene conexiones con todas las neuronas de la capa anterior por las cuales le llega la información ($\{x_i\}_{i=1}^{n}$) y conexiones con todas las neuronas de la capa posterior a las que enviará la salida producida de manera que no permitiremos conexiones hacia atrás.

La capa de salida, la última capa de la red neuronal, está formada por neuronas que reciben la información procesada y la devuelven al exterior. La finalidad de la red neuronal de clasificar es finalmente ejecutada en esta capa, de tal forma que, con los procesos realizados, se selecciona como candidata una de todas las clases para clasificar.

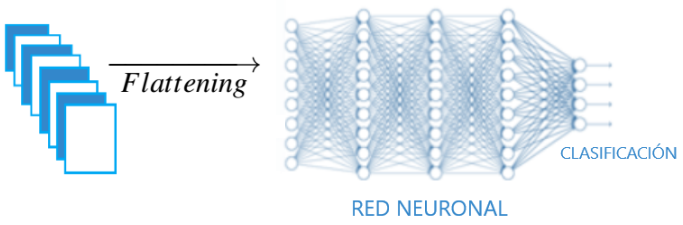


Figura 2.6: Unión de la red neuronal con la red neuronal convolucional

En pocas palabras, la CNN va transformando en cada capa la matriz de entrada y propagando los resultados hacia las capas posteriores a lo largo de nuestra red para ir obteniendo más información hasta llegar a la capa de salida donde se realizará una predicción. Esta forma de propagación en la que solo se transmite desde la capa de entrada a la de salida, sin conexiones hacia atrás, se llama *forward*.

Si se desea hacer una idea visual, tanto de la propagación como de todo el desarrollo de una CNN, se puede visitar `http://scs.ryerson.ca/~aharley/vis/conv/flat.html`. En esta página web se muestra un red neuronal convolucional que toma como entrada una imagen con un número que nosotros mismos le dibujamos y cuya finalidad es predecir el número dando además no solo la primera opción de predicción sino también la segunda con mayor probabilidad.

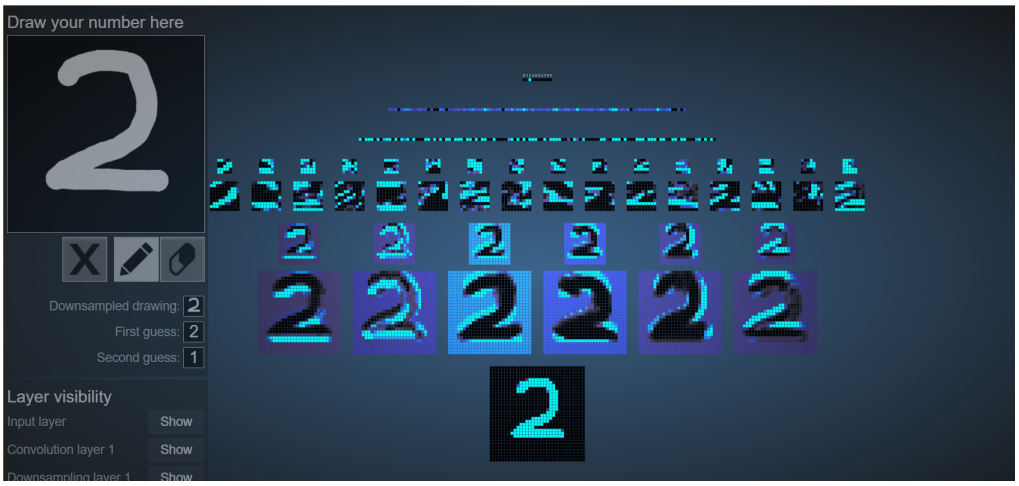


Figura 2.7: Captura de pantalla de la CNN de la página web.

Como vemos, esta CNN tiene una arquitectura de 2 capas de convolución, 2 ocultas de *fully connected* y una capa de salida, donde se iluminan con mayor intensidad los números que tienen más probabilidades de ser el dibujado. Además, una de las ventajas de la interactividad de la web es la visualización de los elementos que van influyendo en el siguiente paso si ponemos nuestro cursor encima.

Para que la predicción de la CNN sea correcta deberemos enseñarle la relación entre la entrada y la salida en función de la predicción que ella misma hace. Este proceso de aprendizaje se realiza durante la ya mencionada fase de entrenamiento.

2.2. Aprendizaje

La forma de enseñar a las CNNs es modificar sus parámetros hasta alcanzar la precisión deseada o, aceptar la precisión que se alcanza. Los parámetros de este tipo de redes son: los filtros de las capas de convolución, los pesos de la red neuronal integrada y los sesgos utilizados en los productos; así pues, modificándolos, iremos cambiando la información que la red extrae de los datos de entrada hasta fijar los parámetros que le permitan distinguir los datos de entrada entre las clases deseadas.

Como hemos visto, los pesos de la parte de red neuronal se pueden agrupar y ser tratados como una matriz, asimismo cada elemento del filtro se puede ver como un elemento independiente. De esta manera, todos los parámetros de la red pueden ser tratados de forma similar

aunque organizados de diferente manera en la arquitectura. De ahora en adelante, si hablamos de pesos en forma genérica haremos referencia a todos los parámetros de la red como elementos independientes, es decir, a cada elemento del filtro, a los pesos de la parte neuronal convencional y a los sesgos.

El razonamiento anterior nos permite igualar en cierta manera las dos partes diferenciadas en la arquitectura. Igualmente, estas similitudes residen en la forma de definir las operaciones en las diferentes capas. Tanto la convolución como la suma ponderada son productos de matrices con la adición de un sesgo, pero la organización en las matrices de filtros (de menor tamaño que las matrices de entrada) de los pesos de la parte convolucional supone una reducción en el número de parámetros. Además, cada elemento del filtro es usado en todas las posiciones de la matriz de entrada (a excepción de algunos elementos del borde dependiendo de las decisiones del diseño de la convolución) dando como resultado que, en lugar de aprender un conjunto aislado de parámetros para cada ubicación, aprendamos un conjunto.

El proceso de aprendizaje, es decir del proceso de actualización de pesos que generalmente se inicializan de forma aleatoria viene dada, en líneas generales, por la optimización de la función de pérdida $\mathcal{L}$. Esta función $\mathcal{L}$ mide el error cometido por la red durante su proceso de aprendizaje y nos muestra el progreso de aprendizaje o retroceso. Durante la fase de entrenamiento, se minimizará esta función con la ayuda de optimizadores mientras la red va reajustando los pesos. Reiteraremos este proceso para un determinado número de iteraciones de forma que en cada una de ellas se minimiza la función de pérdida.

Durante este trabajo nos centraremos en el algoritmo de aprendizaje de retropropagación o *backpropagation* optimizado gracias a la técnica del gradiente descendente estocástico o *Stochastic Gradient Descent*.

A lo largo del proceso de aprendizaje utilizaremos el conjunto de entrenamiento $\{z_1, z_2, ..., z_t\}$, con t el número de observaciones que tendremos en cuenta. Este conjunto está formado por parejas de elementos, $z_i = (I_i, y_i)$, con I_i la matriz de entrada a la CNN e y_i el valor real de la clase a la que pertenece I_i para $i \in \{1, ..., t\}$.

2.2.1. Función de pérdida

Las funciones de pérdida $\mathcal{L}$ son funciones que miden el error que comete una red durante el algoritmo de aprendizaje; miden la distancia entre el valor real y_i y la predicción de la red $\widehat{y}_i$. Su evaluación depende de los pesos asignados ya que influyen en la predicción de la red $\mathcal{L} = \mathcal{L}(\theta; \{z_i\}_{i=1}^t)$ donde θ es el conjunto de valores de los pesos de la red en esa evaluación.

Si $\mathcal{L}$ disminuye después de cada iteración podemos decir que la red está aprendiendo gradualmente de los errores anteriores. Una red con función de pérdida mínima debería devolver una predicción cercana al valor real.

Ejemplos de diferentes funciones de pérdida son:

- **Error cuadrado medio:**

$$\mathcal{L}(\theta; \{z_i\}_{i=1}^t) = \frac{1}{t}\sum_{i=1}^{t}(y_i - \widehat{y}_i)^2$$

- **Error medio absoluto:**

$$\mathcal{L}(\theta; \{z_i\}_{i=1}^t) = \frac{1}{t}\sum_{i=1}^{t}|y_i - \widehat{y}_i|$$

- **Norma L_1:**

$$\mathcal{L}(\theta; \{z_i\}_{i=1}^t) = \sum_{i=1}^{t} |y_i - \widehat{y}_i|$$

- **Norma L_2:**

$$\mathcal{L}(\theta; \{z_i\}_{i=1}^t) = \sum_{i=1}^{t} (y_i - \widehat{y}_i)^2$$

- **Entropía Cruzada o *Cross Entropy*:**

$$\mathcal{L}(\theta; \{z_i\}_{i=1}^t) = \frac{1}{t} \sum_{i=1}^{t} y_i \text{Log}(\widehat{y}_i) + (1 - y_i) Log(1 - \widehat{y}_i)$$

Notar que el verdadero valor de la predicción de una entrada y_i es una clase, es decir una etiqueta, pero estas se codifican numéricamente para poder realizar estas evaluaciones y calcular el error obtenido.

2.2.2. Retropropagación o Backpropagation

El algoritmo de retropropagación estima la contribución de cada peso al valor del error tomando la derivada de la función de pérdida. Es el método utilizado para minimizar la función de pérdida y a su vez el cálculo de la actualización de los pesos.

Con el fin de modelar el algoritmo de la retropropagación introduciremos la siguiente notación.

- n_c: Número de capas de convolución.
- n_n: Número de capas de la red neuronal.
- $N = n_c + n_n$: Número de capas totales de la cnn.
- ϕ_n: Función de activación en la capa n, con $n \in N$.
- ψ_n: Función de pooling en la capa n, para $n \in \{1, ..., n_c\}$.
- $\widehat{y}^n$: Salida de la capa n de la red neuronal ($n \in \{n_c + 1, ..., n_c + n_n\}$).
- C_r^n: Matriz de salida tras el proceso de convolución con el filtro r en la capa n con dimensiones $\overline{m}^n \times \overline{n}_c^n$ y aplicación de la función de activación a todos sus elementos.
- S_r^n: Matriz de salida tras el proceso de pool en la capa n con dimensiones $\overline{m}_s^n \times \overline{n}_s^n$.
- F: Función que realiza el proceso de flattening.

<u>*Parámetros de la red θ*</u>

- $K_{\overline{m},\overline{n},p}^{n,r}$: r filtros de la capa n de tamaño $\overline{m} \times \overline{n} \times p$ con $n \in \{1, ..., n_c\}$.
- W^n: Matriz de pesos de la capa n cuyo número de neuronas es α_n con $n \in \{n_c + 1, ..., n_c + n_n\}$.
- b_r^n: Sesgo r utilizado en la capa n, con $n \in \{1, ..., n_c\}$.
- b^n: Vector de sesgos utilizados en la capa n de la red neuronal ($n \in \{n_c + 1, ..., n_c + n_n\}$) cuya longitud es α^n.

Esta notación está inspirada en [28], donde se muestra el algoritmo de la retropropagación para una arquitectura específica de CNN. En el presente trabajo haremos una generalización del mismo.

Antes de comenzar a explicar el método de la retropropagación, comenzaremos haciendo una pequeña modelización del proceso de predicción de las CNN de *forward* del que hemos estado hablando a lo largo de este trabajo con la notación anteriormente formulada.

Pseudocódigo de feedforward (CNN)

for 1:n_c **do**
  $C_r^n = \phi_n(\sum_{q=1}^{p} S_q^{n-1} \star K_{\overline{m},\overline{n},q}^{n,r} + b_r^n)$ *Etapa de convolución y función de activación*
  $S_r^n = \psi_n(C_r^n)$ *Etapa de pooling*
end for
$F(\{S_r^{n_c}\}_r) = \widehat{y}^{n_c}$ *Flattening*
for $n_c + 1 : N$ **do**
  $\widehat{y}^n = \phi_n(W^n \widehat{y}^{n-1} + b^n)$ *Capas completamente conectadas*
end for
$\widehat{y}^N$ *Predicción de la red neuronal convolucional*

Notar que la capa 0 corresponde a la matriz de entrada a la CNN de dimensiones $\overline{m}_0 \times \overline{n}_0 \times r_0$.

El objetivo del algoritmo de retropropagación es minimizar la función de pérdida, para ello se utiliza el cálculo diferencial por medio de la regla de la cadena con el motivo de percibir el cambio de la función de pérdida respecto a los parámetros de la red. De esta forma iremos transmitiendo la señal desde la capa de salida de la CNN a la de entrada, al contrario que se trasmite cuando la CNN hace una predicción.

De forma genérica obtendremos,

Pesos de las nn

$$\nabla_{W^n(i,j)}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial W^n(i,j)} = \frac{\partial \mathcal{L}}{\partial \widehat{y}^n(i)} \cdot \frac{\partial \widehat{y}^n(i)}{\partial (W^n \widehat{y}^{n-1} + b_r^n)(i)} \cdot \frac{\partial (W^n \widehat{y}^{n-1} + b_r^n)(i)}{\partial W^n(i,j)}$$
$$\forall i \in \{1, ..., \alpha^n\},\ \forall j \in \{1, ..., \alpha^{n-1}\}$$

Sesgos de las nn

$$\nabla_{b^n(i)}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial b^n(i)} = \frac{\partial \mathcal{L}}{\partial \widehat{y}^n(i)} \cdot \frac{\partial \widehat{y}^n(i)}{\partial (W^n \widehat{y}^{n-1} + b^n)(i)} \cdot \frac{\partial (W^n \widehat{y}^{n-1} + b^n)(i)}{\partial b^n(i)}$$
$$\forall i \in \{1, ..., \alpha^n\}$$

Filtros de la cnn

$$\nabla_{K_{\overline{m},\overline{n},p}^{n,r}(u,v)}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial K_{\overline{m},\overline{n},p}^{n,r}(u,v)} =$$
$$= \sum_{i=0}^{\overline{m}_s - 1} \sum_{j=0}^{\overline{n}_s - 1} \frac{\partial \mathcal{L}}{\partial S_r^n(i,j)} \sum_{i'=0}^{\overline{m}_c - 1} \sum_{j'=0}^{\overline{n}_c - 1} \frac{\partial S_r^n(i,j)}{\partial C_r^n(i',j')} \cdot \frac{\partial C_r^n(i',j')}{\partial in_c(i',j')} \cdot \frac{\partial in_c(i',j')}{\partial K_{\overline{m},\overline{n},p}^{n,r}(u,v)}$$
$$\forall u \in \{1, ..., \overline{m}\}, \forall v \in \{1, ..., \overline{n}\}$$

Sesgos de la cnn

$$\nabla_{b_r^n}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial b_r^n} = \sum_{i=0}^{\overline{m}_s - 1} \sum_{j=0}^{\overline{n}_s - 1} \frac{\partial \mathcal{L}}{\partial S_r^n(i,j)} \sum_{i'=0}^{\overline{m}_c - 1} \sum_{j'=0}^{\overline{n}_c - 1} \frac{\partial S_r^n(i,j)}{\partial C_r^n(i',j')} \cdot \frac{\partial C_r^n(i',j')}{\partial in_c(i',j')} \cdot \frac{\partial in_c(i',j')}{\partial b_r^n}$$

Con $in_c(i', j') = \sum_{q=1}^{p} S_q^{n-1} \star K_{\overline{m},\overline{n},q}^{n,r} + b_r^n$

El método de optimizar buscando los puntos críticos y, entre ellos, obtener el mínimo, es muy costoso por la cantidad de parámetros de la red. Por este motivo, el algoritmo de propagación se ayuda de optimizadores como el Gradiente Estocástico Descendente (*Stochastic Gradient Descent*).

2.2.3. Gradiente Estocástico Descendente (SGD)

Aplicar optimizadores al método de retropropagación ayuda a encontrar el mínimo de una función de pérdida de forma más rápida ya que resuelven de forma numérica el sistema de ecuaciones no lineales de la sección anterior. El optimizador SGD pertenece a la familia de los optimizadores que utilizan la técnica del gradiente descendente, buscan el mínimo actualizando los parámetros en la dirección opuesta al gradiente.

El motivo anterior conlleva a la necesidad de reiteraciones, ya que es muy probable que no se alcance el mínimo a la primera. Como consecuencia, se hace necesaria la definición de varios conceptos correlados que pueden inducir a error:

- Una **época** o *epoch* es el tiempo en el que se procesa todo el conjunto de entrenamiento.
- El **tamaño del lote** o *batch size* es el número de elementos del conjunto de entrenamiento utilizado en cada proceso de aprendizaje.
- Una **iteración** es el tiempo en el que se procesa un lote tras el cual se actualizarán los parámetros.

La relación entre estos tres conceptos es la siguiente. Para cada una de las épocas definidas en la CNN se ha de cumplir,

$$\text{tamaño del lote} \times \text{iteraciones} \geq \text{tamaño del conjunto de entrenamiento.}$$

Por este motivo tan solo es necesaria la definición de un elemento más a parte del número de épocas porque el otro queda unívocamente determinado.

El método del gradiente estocástico descendente utiliza un único elemento escogido al azar de entre todo el conjunto de entrenamiento para actualizar los pesos en la iteración $k+1$ como sigue,

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta \mathcal{L}(\theta_k; z_i), \text{ para un } i \in \{1, ..., t\},$$

con η el parámetro de aprendizaje, que nos indica la magnitud de cambio en la dirección opuesta al gradiente. Dicho parámetro no tiene porque ser constante a lo largo de todas las iteraciones. Este elemento escogido al azar indicará el orden en el cual los elementos serán predichos por la CNN.

No se puede calcular analíticamente la tasa de aprendizaje óptima para un modelo dado un conjunto de entrenamiento. Se sabe que si el parámetro es muy pequeño podemos tener una convergencia muy lenta y si es muy grande puede no converger o diverger como se muestra en la siguiente figura.

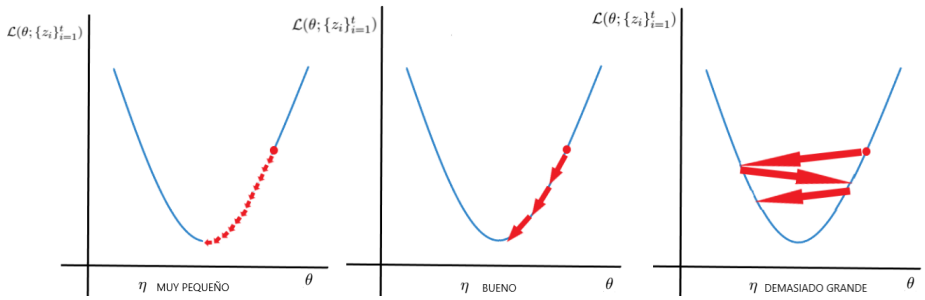


Figura 2.8: Cambio en la convergencia respecto al tamaño el parámetro de aprendizaje.

Las condiciones necesarias para que el SGD converja siendo η_k el parámetro de aprendizaje de la iteración k son:

$$\sum_{k=1}^{\infty} \eta_k = \infty \quad \text{y} \quad \sum_{k=1}^{\infty} \eta_k^2 < \infty.$$

En la práctica se suele usar el valor predeterminado 0.1 o 0.01.

Como podemos ver en [26], la idea del SGD se puede relacionar con la teoría de *Herbert Robbins* y *Sutton Monro*, quienes hicieron un estudio detallado del método de búsqueda de raíces en el algoritmo de Robbins-Monro en lo que forma parte de la teoría "*A stochastic approximation method*". Esta relación surge de que se puede considerar equivalente el problema de obtener el mínimo de la función de pérdida con el de encontrar el cero del gradiente de dicha función. De esta forma podemos aplicar al SGD resultados como las condiciones de convergencia presentadas anteriormente. Si se desea ampliar la información sobre esto se puede consultar la referencia [25].

Dentro de la familia de los optimizadores que emplean la técnica del gradiente descendente también se incluyen por ejemplo el *mini-batch gradient descent*, que actualiza los parámetros tras procesar un lote completo del conjunto de entrenamiento pero no cada elemento como el SGD o el *Batch gradient descent* que los actualiza tras procesar todo el conjunto de entrenamiento, lo que aporta más varianza en el gradiente. Otros ejemplos junto con su formulación los encontramos en [20].

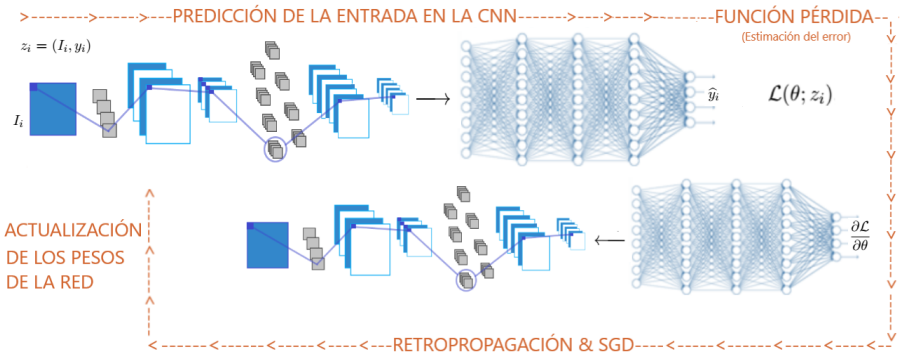


Figura 2.9: Esquema del proceso de aprendizaje de retropropagación durante una iteración para la observación i escogida al azar con el optimizador SGD.

2.2.4. Importancia de las funciones de activación en el aprendizaje

A simple vista puede parecer que la elección de las funciones de activación es superflua pero no es así. Como hemos visto, durante el proceso de retropropagación debido al uso de la regla de la cadena, se calculan los gradientes de las funciones de activación. Estos gradientes influyen mucho en el proceso de aprendizaje dado que pueden provocar que una red aprenda de forma lenta o incluso deje de aprender.

De igual modo que en [16], vamos a realizar un estudio de las ventajas y desventajas de las funciones de activación presentadas anteriormente y su intervención en el proceso de aprendizaje.

- Función sigmoide, $\phi(x) = \dfrac{1}{1+e^{-x}}$.

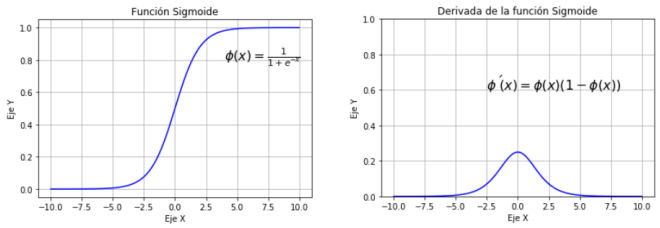


Figura 2.10: Función sigmoide y su derivada.

Una de las ventajas que presenta esta función es la facilidad de su derivada, ya que se puede escribir en términos de ella misma, $\phi'(x) = \phi(x)(1-\phi(x))$.

La función sigmoide toma un valor real y genera un valor en el intervalo $(0, 1)$. Debido a su forma y al dominio de definición tan grande, se puede considerar que, para valores muy grandes o muy pequeños de los pesos la salida es casi binaria, lo que implica que el gradiente en estos puntos es prácticamente nulo. De esta forma se puede inicializar una neurona de tal manera que no se active o que nunca aprenda. En estos extremos, se cumple también que, grandes cambios en la entrada corresponden a cambios pequeños en la salida.

- Función tangente hiperbólica, $\phi(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$.

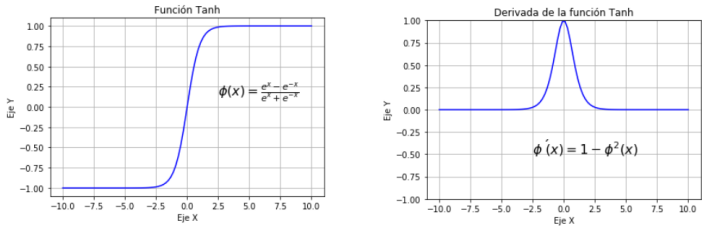


Figura 2.11: Función Tanh y su derivada.

La tangente hiperbólica es similar a la función sigmoide, toma un número real transformándolo esta vez en el rango $[-1, 1]$, centrando la salida en el 0 y con una derivada $\phi'(x) = 1 - \phi^2(x)$ más pronunciada.

La función sigmoidea y la tangente hiperbólica tienen el problema conocido por *"vanishing gradient problem"* o "gradiente de fuga". Este problema es causado porque los gradientes se encuentran en el intervalo (0,0.25) para la función sigmoidea y (0,1) para la tangente hiperbólica. El efecto de multiplicar n veces en el proceso de aprendizaje, siendo n el número de capas que usan estas funciones de activación, produce un decrecimiento exponencial del gradiente lo que provoca que la red aprenda de forma lenta. Ir alternando estas funciones con otras de activación hace que este problema pueda desaparecer.

- Función ReLU, $\phi(x) = max(0, x)$.

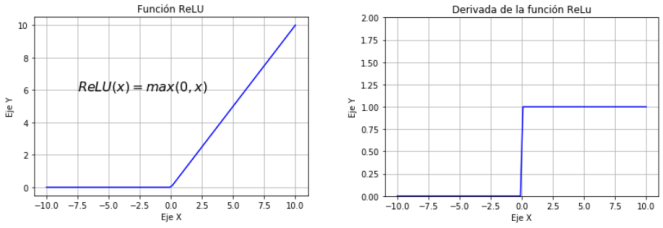


Figura 2.12: Función ReLU y su derivada.

La función ReLU ofrece las ventajas de la no linealidad de las funciones anteriores con un mayor rendimiento y menos coste computacional por la forma de su derivada. Presenta el inconveniente de restringir su uso a las capas intermedias de la red por su rango de definición en la salida $[0, \infty)$.

Esta función de activación hace que la información se propague a través de la red si la entrada es mayor o igual a cero. Para el resto de entradas el valor del gradiente será 0 así que los pesos puede que no se ajusten durante la fase de aprendizaje. Esto significa que estas neuronas pueden dejar de responder a las variaciones en el error porque el gradiente es cero, problema conocido como *"dying ReLU problem"*.

Notar que esta función no es derivable aunque definiremos su derivada para el cero obteniendo,

$$\phi'(x) = \begin{cases} 1 & si\ x > 0, \\ 0 & si\ x \leq 0. \end{cases}$$

De esta manera podemos aplicar la regla de la cadena sin ofrecer problema ni oponernos al trasfondo del aprendizaje. Si la entrada de la función de activación es cero, esta función no se activa y no envía información (envía cero), luego no influirá en el error esta neurona así que la propagación hacia atrás se puede tomar como cero.

- Función ELU, $\phi(x) = \begin{cases} x & si\ x > 0, \\ a(e^x - 1) & en\ \ otro\ \ caso. \end{cases}$

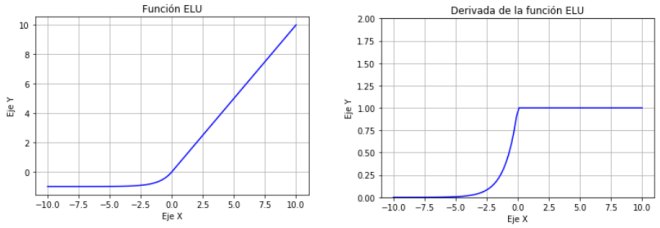


Figura 2.13: Función ELU y su derivada para a=1.

Esta función es bastante usada ya que tiende a converger el coste a cero más rápido y con resultados más precisos. Puede producir salidas negativas para entradas negativas y está indicada para capas ocultas por su rango de salida. Su principal diferencia con la ReLU es la suavidad hasta alcanzar el valor -a.

En el caso de las capas convolucionales, por su configuración, estas apreciaciones pueden cambiar ligeramente. La razón es el algoritmo de retropropagación, ya que se modifican los parámetros a través de varias neuronas, al contrario que las capas de la red neuronal que se hace mediante una única neurona. Es decir, en las capas convolucionales, que una neurona no se active no implica necesariamente que esos pesos no se modifiquen.

2.3. Evaluación de la clasificación de las CNNs

El conjunto de validación tiene la finalidad de testear la CNN con un conjunto que no ha sido utilizado para el aprendizaje. De esta manera tendremos una representación más fiel de como clasifica nuestra red y evitar el sobreajuste.

Como hemos mencionado anteriormente, el conjunto de validación tiene las mismas características que el de entrenamiento ya que ambos componen la base de datos con la que construiremos nuestro modelo. Esta partición se suele hacer entorno a un 80 % de los datos para entrenar y el 20 % para validar.

Haremos una predicción de este conjunto de activación con la red entrenada para aplicar técnicas que nos miden la buena clasificación del modelo, haciendo un estudio de los resultados de esas predicciones. En este trabajo le daremos mayor importancia a las matrices de confusión.

2.3.1. Matriz de confusión

La **matriz de confusión** permite visualizar el rendimiento del modelo. Aunque normalmente en forma de tabla, la matriz de predicción representa el número de predicciones de cada clase (filas) en función de las instancias en la clase real (columnas) o viceversa.

		CLASE REAL			
		Sujetos clase 1	Sujetos clase 2	...	Sujetos clase n
CLASE PREDICHA	Predichos clase 1	B_{11}	M_{12}	...	M_{1n}
	Predichos clase 2	M_{21}	B_{22}	...	M_{2n}
	⋮	⋮	⋮	⋱	⋮
	Predichos clase n	M_{n1}	M_{n2}	...	B_{nn}

Figura 2.14: Esquema de una matriz de confusión para n clases.

Con la información ordenada de esta manera, se ve de forma sencilla que el número de individuos bien clasificados se sitúan en la diagonal de la matriz y fuera los que se han clasificado de forma errónea.

La matriz de confusión tiene especial relevancia si el número de clases a clasificar es dos ya que nos permite hace un análisis más detallado sobre la calidad de la clasificación de la red. Comúnmente se usa la terminología de renombrar las clases como positiva y negativa si la matriz de confusión solo tiene dos clases y así hablar de falsos positivos o falsos negativos para los individuos mal clasificados.

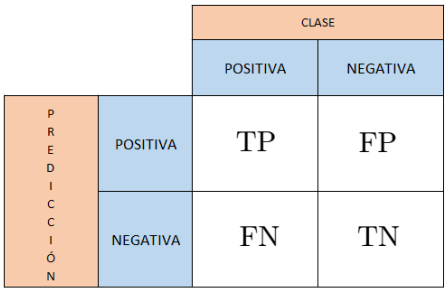


Figura 2.15: Esquema de una matriz de confusión para 2 clases.

El análisis consiste en el cálculo de los siguientes conceptos.

- Tasa de bien clasificados (*accuracy*): porcentaje de los individuos bien clasificados respecto al total de predicciones.

$$ACC = \frac{TP + TN}{TP + FP + FN + TN}$$

- Tasa de mal clasificados (TMC): porcentaje de los individuos mal clasificados respecto al total de predicciones.

$$TMC = \frac{FP + FN}{TP + FP + FN + TN}$$

- Sensibilidad (TPR): porcentaje de los individuos correctamente clasificados como positivos respecto al total de individuos positivos (tasa de verdaderos positivos).

$$TPR = \frac{TP}{TP + FN}$$

- Especificidad (TNR): porcentaje de los individuos correctamente clasificados como negativos respecto al total de individuos negativos (tasa de verdaderos negativos).

$$TNR = \frac{TN}{FP + TN}$$

- Precisión (PPV): porcentaje de los individuos correctamente clasificados como positivos respecto al total de individuos clasificados como positivos.

$$PPV = \frac{TP}{TP + FP}$$

- Tasa de falsos positivos (FPR): porcentaje de individuos negativos que se han clasificado como positivos.

$$FPR = \frac{FP}{FP + TN}$$

- Tasa de falsos negativos (FNR): porcentaje de individuos positivos que han sido clasificados como negativos.

$$FNR = \frac{FN}{FN + TP}$$

Un buen clasificador debería tener los valores de la tasa de bien clasificados, sensibilidad, especificidad y precisión próximos a 1 o lo mayor posible y la tasa de mal clasificados, de falsos positivos y falsos negativos lo más cercana a 0 o lo más pequeña posible. No siempre se podrán obtener los valores de lo que sería un perfecto clasificador porque a nuestros datos no lo permiten o no queremos complicar tanto el modelo por algún motivo como puede ser la rapidez de ejecución.

El análisis de nuestro clasificador consistirá en observar aquellos porcentajes de los conceptos mencionados y ver si nuestro modelo está en un ratio de resultados que aceptemos, como por ejemplo que la tasa de bien clasificados sea por lo menos del 80 %, si no es así deberemos cambiar la arquitectura del modelo y volver a entrenar el modelo para volver a hacer la validación.

2.3.2. Otras técnicas de validación

Otras técnicas que se pueden emplear para la evaluación de la clasificación de nuestra red son:

- Tasa de error nula (*null error rate*): frecuencia con la que se equivocaría si siempre se predijera la clase mayoritaria.
- Kappa de Cohen: Es una media que compara lo bien que cataloga el clasificador en comparación de una asignación de las clases al azar.
- Curva ROC: Gráfico utilizado solo en el caso de tener dos clases donde se representa la tasa de falsos positivos (eje x) frente a la tasa de verdaderos positivos (eje y) en función de un valor de decisión creciente. Calculando el área bajo la curva (AUC) nos podemos hacer idea de lo buen clasificador que es nuestro modelo teniendo en cuenta que un clasificador que no se equivoca nunca tendría un AUC=1.

En función del resultado de estas técnicas, se finalizarán la construcción y aprendizaje del modelo o se modificarán con el fin de conseguir resultados satisfactorios, tal y como se refleja en el siguiente esquema:

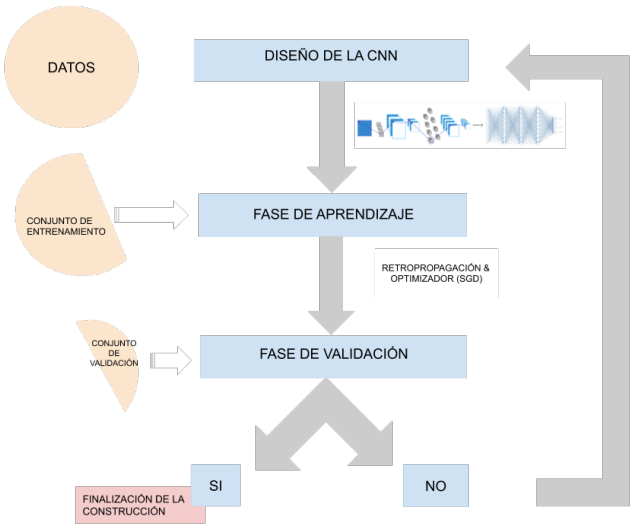


Figura 2.16: Esquema del proceso de construcción de una CNN.

2.4. Predicción de la clase

Tras la validación de nuestro modelo, la CNN estará lista para usarse y poder realizar la finalidad para la que se construyó.

Tanto las predicciones que se realizan a lo largo del proceso de construcción de la CNN como las predicciones de los datos nuevos se resumen en una serie de operaciones dentro de la red, como hemos visto, gracias a las definiciones de los algoritmos utilizados. En nuestro caso particular, la entrada a la red será una imagen codificada en forma de matriz con la que realizaremos operaciones (sumas y productos) y obtendremos un valor que corresponderá a una clase.

Como ya mencionamos sin detenimiento, estos valores que obtenemos son codificaciones numéricas de las etiquetas de las clases de nuestro problema. Codificaciones habituales son el uso del 0 y 1 en el caso de tener dos clases o la codificación *one hot encoding*. Esta última codificación es la más utilizada para problemas multiclase, es decir, problemas con más de dos clases en la predicción. Consiste en la asignación de un vector de longitud igual al número total de clases cuyo único elemento distinto de cero tiene valor 1 y cuyo lugar corresponde con una de las clases a clasificar. Estas codificaciones basadas en 0s y 1s independientemente del número de clases nos permiten definir el algoritmo del modelo, fundamentalmente las funciones de activación, como lo hemos hecho.

Capítulo 3

Estado del arte

La aplicación más conocida de las CNNs es la visión por ordenador, donde el conjunto de entrada son una serie de imágenes y la red realiza una tarea tras la extracción de patrones de estas. Sin embargo, las imágenes no son el único formato de datos con los que las CNNs pueden trabajar, los textos o las series temporales son ejemplos de esto. Pero no solo eso, si nuestros datos se pueden codificar de manera que concuerden con la forma de entrada, es muy probable que las redes puedan sacar patrones y aprender de ellos para poder llevar a cabo una labor. Aunque esta tarea resulta complicada porque se añade esa complejidad de codificar los datos de manera eficiente para poder sacar información de estos. Como curiosidad, en relación a esto último, se puede ver el artículo [3] donde se codifican los ficheros ejecutables binarios en imágenes en escala de grises para poder introducirlos a una red y con esto detectar códigos maliciosos.

Vamos a ver algunos ejemplos de forma detallada de las aplicaciones de las CNNs a imagen, texto y series temporales con la exposición de varios artículos:

- Ayuda en el diagnóstico de enfermedades.

  Una de las aplicaciones importantes de las CNNs es la ayuda en el diagnóstico de enfermedades. Este tipo de redes extraen patrones de los datos recogidos por las pruebas médicas que en algún caso, por su volumen, hace difícil su análisis manual.

  Un ejemplo de la obtención de gran cantidad de datos es la resonancia magnética, que proporciona imágenes tridimensionales que posibilitan la detección de las lesiones de materia blanca. Estas lesiones son células desmielinizadas que se encuentran en la materia blanca del cerebro. Este ejemplo es el que se recoge en el artículo [4], donde se aplican las CNNs para la detección de estas lesiones.

  Aunque en este caso los datos que se han de estudiar están en forma de imagen, y por tanto ya tenemos nuestros datos en forma de matriz, se realiza un preprocesamiento por dos motivos principales: mejorar la imagen y adaptarla a la entrada de la CNN construida. Para este preprocesamiento se realizan varios procesos.

  Primero se extrae el cráneo de las imágenes obtenidas en la resonancia magnética, proceso que está explicado en [1] en el apartado *C Skull Stripping*. Una vez tenemos solo la imagen del cerebro sin el recubrimiento del cráneo la mejoramos. Esta mejora principalmente se aplica para aumentar el contraste en la imagen y que así sea más fácil la clasificación de las células para saber si están desmielinizadas. En este artículo que estamos presentando se aplica el algoritmo de comparación de histogramas que se puede consultar en

[7, págs 88-108]. Finalmente para adaptar la imagen a la entrada de la CNN entrenada se utiliza el algoritmo SLICO que se encarga de hacer una segmentación del cerebro agrupando los píxeles en función de algún requisito, que en nuestro caso es el color. Esta agrupación se llamará superpíxel. Para más información del algoritmo SLICO se puede consultar [18]. Finalmente, la entrada a nuestra red serán estos superpíxeles y sus vecinos.

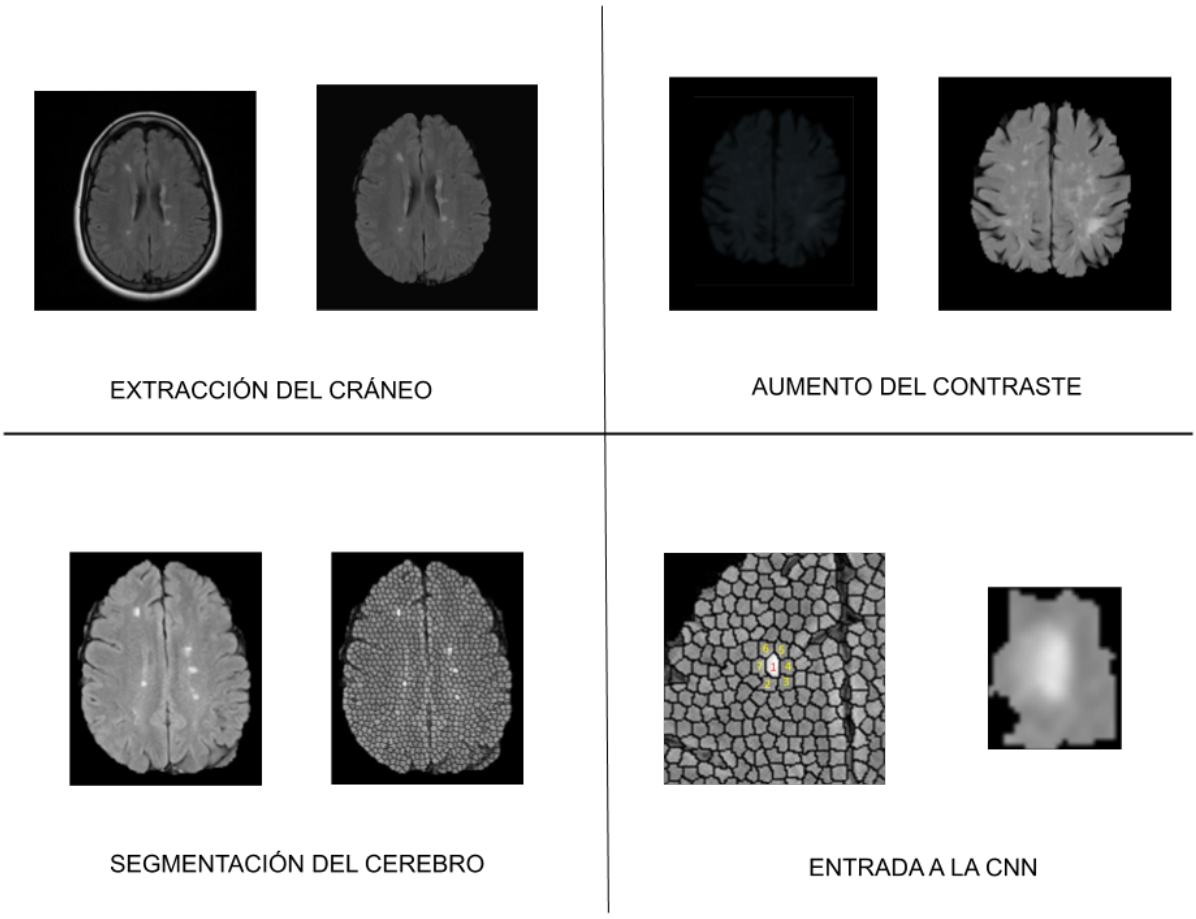


Figura 3.1: Antes y después de los procesos indicados en el tratamiento de la imagen. Imágenes tomadas del artículo [4].

He elegido este artículo a parte de su aplicación con el fin de mostrar que el uso de imágenes en las CNNs también puede requerir un tratamiento de estas previo. Aunque la imagen para el ordenador sea como una matriz de números y no necesite esa adaptación hay veces que es necesario recortarla o mejorarla. Estos tratamientos previos de las imágenes pueden mejorar y simplificar la tarea de clasificación para resultar más sencilla.

El artículo presenta la red que mejor resultados da para su finalidad, clasificar los superpíxeles en dañados y no dañados entendiendo por dañados aquellos que representarían lesiones blancas, después de realizar experimentos para su diseño. La arquitectura del modelo seleccionado, presentada en el cuadro 3.1, ha sido la que mejor resultado obtuvo comprando los resultados de sensibilidad, especificidad, tasa de bien clasificados y falsos positivos por imagen. La red destacada se compone de las siguientes características:

Arquitectura de la CNN
Convolución con 20 filtros de 5×5 Función de activación ReLu Maxpooling 2×2
Convolución con 50 filtros de 5×5 Función de activación ReLu Maxpooling 2×2
Flattening
Capa completamente conexa de 500 neuronas & ReLu
Capa de salida 2 neuronas & Softmax

Cuadro 3.1: Arquitectura de la CNN utilizada en el experimento.

Destacamos que después de la segunda capa de convolución se aplica *Dropout* del 50 %. Esta técnica consiste en bloquear un porcentaje de neuronas de una capa que se indica al azar, en este caso el 50 % de ellas, para que no actualicen sus pesos y como consecuencia no aprendan. Esta técnica evita el sobreajuste de los datos previniendo que la red se adapte mucho a los datos del conjunto de entrenamiento.

El conjunto de datos con el que se cuenta tiene la información de 91 pacientes de los cuales, una vez determinada la arquitectura de la CNN, se han usado 76 para el conjunto de entrenamiento y 15 para el conjunto de validación seleccionados al azar. La red resultante detectó el 78.79 % de las lesiones de materia blanca con solo un 0.005 de falsos positivos por imagen y con una especificidad de 98.77 % y tasa de bien clasificados de 98.73 %.

- Análisis de texto.

Como ya hemos mencionado las CNNs no se aplican únicamente a imágenes. En el artículo [15] se aplican para análisis de texto como clasificadores de la satisfacción o sentimiento recogido en el texto. Es decir, en nuestro caso, clasifica si en el texto que se le introduce se expresa un sentimiento positivo o uno negativo.

Este tipo de redes se propone para estudios de este tipo ya que, gracias a sus capas de convolución, son capaces de agrupar y extraer información global. Esta característica pretende mejorar los métodos convencionales como la máquina de vector soporte (SVM) o Naïve Bayes que analizan el texto palabra por palabra perdiendo información al clasificar la palabra principal sin tener en cuenta al resto.

Para hacer una clasificación con las redes estudiadas necesitaremos un preprocesamiento de los datos que nos transforme el texto en una matriz con su información en la que cada fila será un vector que representa a cada una de las palabras. Si el texto cuenta con s palabras y el vector que codifica las palabras tiene dimensión d, el texto se sustituirá por una matriz de dimensiones $s \times d$. Uno de los métodos más conocidos para este preprocesamiento es el método Word2vec, que se explica de forma detallada en [6].

Tras este proceso, tendremos una matriz de palabras que formará la entrada a la red. Una vez tenemos los datos en forma de matriz se aplicarán los procesos que hemos descrito a lo largo de este trabajo aunque con una pequeña modificación ya que en este caso se utiliza la convolución y pooling conocidos como convolución 1D (1-dimensional) o pooling 1D. La diferencia entre la 1D y la 2D (que es la presentada en el capítulo 2 de este trabajo) es la dimensionalidad de los datos de la entrada junto con su adaptación del producto de convolución, como se desliza el filtro a través de la matriz resultante y el entorno en el que hacemos el *pooling*.

Figura 3.2: Forma en la que el filtro recorre la matriz en la que hemos codificado el texto.

Podemos observar en los casos 1-dimensionales como el filtro solo se mueve en una dirección a diferencia de los filtros en el caso 2D. Por tanto, para definir estos filtros 1D, nos bastará con definir su altura ya que el número de columnas será igual al número de columnas del conjunto de entrada. Esta construcción nos hace poder agrupar n palabras (que será la dimensión del filtro) para obtener patrones de los conjuntos de palabras mediante la convolución. No tendría sentido el movimiento del filtro en sentido horizontal, ya que no nos podemos olvidar que las filas representan la codificación de palabras completas y por tanto no obtendríamos la relación entre ellas. Como resultado de este proceso de convolución obtenemos una matriz cuyas dimensiones son:

$$(n^{o}\ filas\ de\ la\ matriz - n^{o}\ de\ filas\ del\ filtro + 1) \times n^{o}\ filtros\ aplicados.$$

La convolución genera en la matriz resultante, para cada columna, los resultados de convolucionar con el mismo filtro. El proceso de *pooling* se aplica en estas columnas, definiéndose ahora el entorno como un número de elementos contiguos a los que aplicarle los resúmenes estadísticos que mencionamos en la parte teórica y así reducir dimensiones.

Un ejemplo de arquitectura y funcionamiento de red para texto sería:

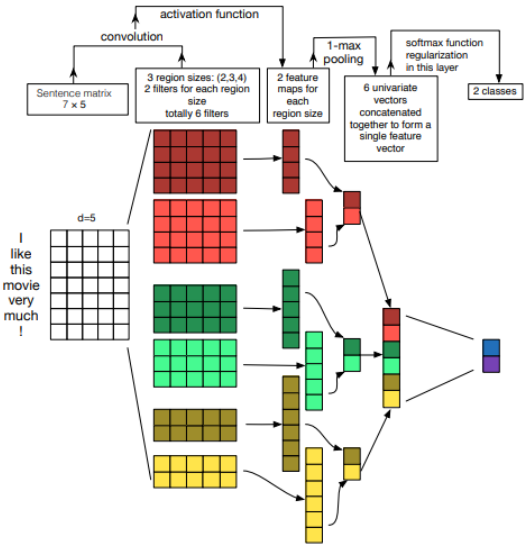


Figura 3.3: Ejemplo de una estructura de red neuronal convolucional aplicada a textos.

En el artículo se hace un experimento con dos conjuntos de datos: el conjunto MR, que consta de una colección de reseñas de películas y el conjunto STS, una colección de tweets reales. Se consigue para los conjuntos de validación de los datos mencionados una tasa de bien clasificados de 74.5 % y 68 % respectivamente, obteniendo un porcentaje mayor que para los clasificadores convencionales.

Si se desea tener mayor información de este proceso a parte de [15] se puede consultar [27] que cuenta con un estudio más extenso y es donde se ha obtenido la figura 3.3.

- Predicción en series temporales.

En el artículo [9] se proponen, para predecir la velocidad y la dirección del viento, las técnicas de clasificación de las 1D CNNs con un conjunto de datos del viento en forma de series temporales, es decir, secuencias de datos medidos en el tiempo (en intervalos regulares) y ordenados cronológicamente.

El conjunto de datos utilizado para el experimento está formado por series temporales para la dirección y la velocidad del viento, medidas en dos lugares diferentes, Stuttgart (Alemania) y Holanda, con intervalos de tiempo mensuales. Las muestras de este conjunto tendrán múltiples entradas, en concreto una para cada elemento que queremos predecir. Es decir, para un valor de tiempo t_i tendremos dos mediciones, la respectiva a la velocidad y la de la dirección en cada entrada.

Para definir la longitud de la entrada así como el valor de la clase de esta se definen dos escalares, W_s y W_B. Por un lado, W_s que define las $W_s + 1$ medidas consecutivas que formarán la entrada a la red a partir de un cierto tiempo t_j que se irá moviendo para obtener diferentes muestras. Como consecuencia la entrada de la red estará formada por las mediciones de la velocidad y la dirección del viento en los valores del tiempo $\{t_j, t_{j+1}, ..., t_{j+W_s}\}$.

Por otra parte, W_B definen las $W_B + 1$ medidas consecutivas tras el último valor de la entrada, es decir tras t_{j+W_s}, de la misma manera que se definen los valores la entrada. El conjunto definido por W_B sirve para calcular el valor real de la predicción (su clase) de la entrada como veremos a continuación y para definir el intervalo de tiempo para el que se hará la predicción.

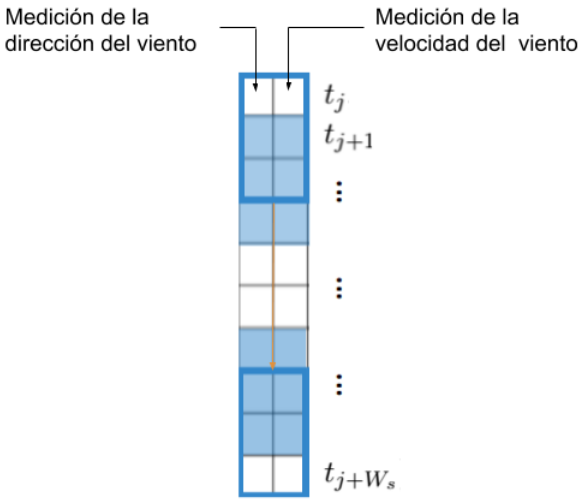


Figura 3.4: Matriz de entrada a la CNN de las series temporales y forma de recorrer el filtro en dicha matriz.

La forma de predecir de estas redes consiste en fragmentar $\mathbb{R}$ en varios segmentos o intervalos donde incluiremos el extremo inferior y excluiremos el superior. Esta fragmentación se basa en la media μ y la desviación estándar σ del conjunto completo de los datos calculado para la velocidad y la dirección por separado. En este estudio se definen los límites de los intervalos de la misma forma para ambas características, dando lugar a las 11 clases con las que clasificará la CNN. Como resultado, la CNN predice clasificando la entrada en una de las siguientes clases:

Clase	Rango inferior	Rango superior
1	$\mu - k_1\sigma$	$\mu + k_1\sigma$
2	$\mu + k_1\sigma$	$\mu + k_2\sigma$
3	$\mu + k_2\sigma$	$\mu + k_3\sigma$
4	$\mu + k_3\sigma$	$\mu + k_4\sigma$
5	$\mu + k_4\sigma$	$\mu + k_5\sigma$
6	$\mu + k_5\sigma$	∞
7	$\mu - k_2\sigma$	$\mu - k_1\sigma$
8	$\mu - k_3\sigma$	$\mu - k_2\sigma$
9	$\mu - k_4\sigma$	$\mu - k_3\sigma$
10	$\mu - k_5\sigma$	$\mu - k_4\sigma$
11	$-\infty$	$\mu - k_5\sigma$

Cuadro 3.2: Definición de las clases utilizadas.

Donde se toman los valores de k_1, k_2, k_3, k_4 y k_5 como $0'15, 0'45, 0'65, 0'95$ y $1'25$ respectivamente. Notar que aunque se definan de la misma forma para clasificar la velocidad y la dirección, los valores de μ y σ cambian y por tanto los valores de los intervalos serán distintos.

Una vez definidas las clases, la forma de calcular la etiqueta de las muestras es ordenar los $W_B + 1$ valores en estos intervalos y calcular el intervalo con mayor número de ellos. Este intervalo será la clase o etiqueta que le asignemos a la muestra. Notar que en cada arquitectura que se presentan se entrena y se estudia los resultados para la velocidad y la dirección por separado cambiando únicamente los valores de las etiquetas, las entradas serán exactamente las mismas para ambos casos.

En este artículo se proponen dos arquitecturas de 1D CNN, una simple (1DS) donde la entrada es la que hemos mencionado anteriormente y una múltiple (1DM) donde la entrada se duplica de varias formas para obtener 5 entradas diferentes y con ello se extrae información diferente en 5 grupos de convolución aislados hasta que se conectan tras el proceso de *flattening*. La duplicación de la información consiste en, para un grupo de convolución, introducir los datos como en 1DS, para el segunda comenzando con el primer valor de la muestra e incrementando el valor de t_i en dos, para el tercero la duplicación se hace de forma análoga a la segunda pero comenzado con el segundo valor. La entrada al cuarto y al quinto grupo es definido de forma similar al del segundo y tercero pero realizando un incremento de tres. El esquema resultante de las arquitecturas utilizadas es el siguiente.

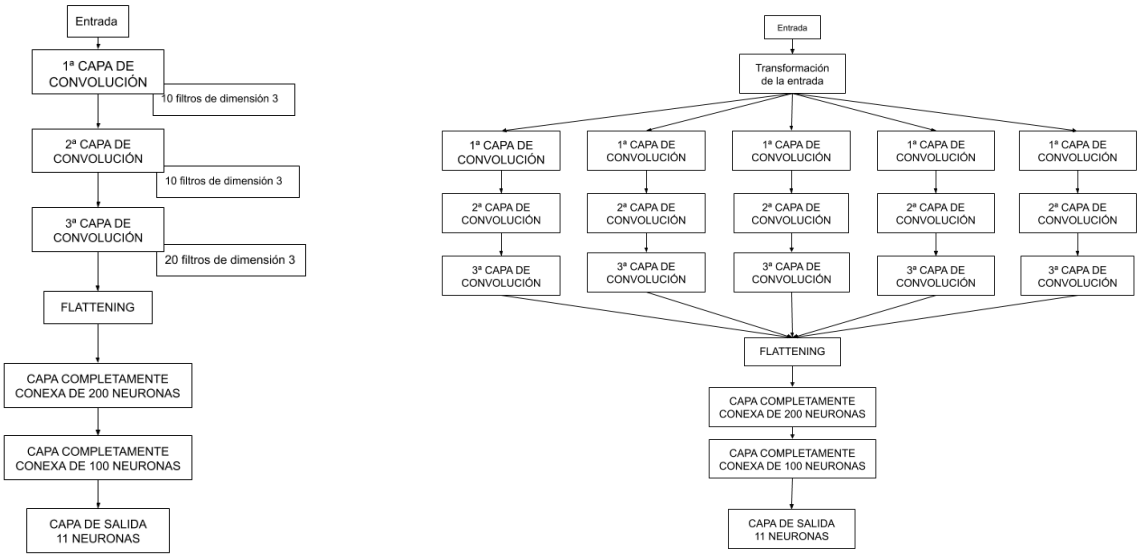


Figura 3.5: Estructuras propuestas de las CNN (a la izquierda la estructura simple y a la derecha la estructura múltiple).

Las características de las partes convolucionales son las mismas que en el caso simple y la función de activación utilizada durante todo momento es la ELU con $a = 30$. Además, después de cada una de las capas de convolución se realiza un *dropout* del 20 % de las neuronas y la técnica de *Batch Normalization* que reduce la covarianza de los datos normalizándolos de manera que optimiza y acelera el entrenamiento de la red.

Algunos de los resultados obtenidos en el estudio son los siguientes, no obstante, se recalca en él que los resultados podrían mejorar añadiendo más capas en las arquitecturas aunque requiere de recursos de *hardware* y unidades de procesamiento gráfico (GPU) mejores a los utilizados. Para mayor detalle del proceso y más información de los resultados, consultar el propio artículo [9].

	Stuttgart		Holanda	
Característica	peor *accuracy*	mejor *accuracy*	peor *accuracy*	mejor *accuracy*
velocidad	85.4 %	90.2 %	90 %	95.2 %
dirección	89.8 %	95.1 %	91.3 %	94.7 %

Cuadro 3.3: Resultados para la arquitectura 1DS CNN.

	Stuttgart		Holanda	
Característica	peor *accuracy*	mejor *accuracy*	peor *accuracy*	mejor *accuracy*
velocidad	92.0 %	96.8 %	93.6 %	99.7 %
dirección	97.5 %	98.8 %	97.6 %	99.4 %

Cuadro 3.4: Resultados para la arquitectura 1DM CNN.

Estudios de este tipo son importantes, por ejemplo como se comenta en el artículo, para la instalación de aerogeneradores cuya potencia depende de la velocidad y dirección del tiempo.

Estos son algunos ejemplos en las aplicaciones de las CNNs, pero no son los únicos. Se ha elegido un ejemplo que resulte representativo para los principales tipos de datos con los que se trabaja hoy en día en las CNNs y que, igualmente, contuviese alguna característica especial como el tratamiento de las imágenes, la codificación de los datos iniciales o arquitecturas diferentes a las simples que se han mostrado en la teoría permitiendo varios grupos de capas de convolución aislados.

Los estudios relacionados con las técnicas de las CNNs han experimentado en estos últimos años un aumento, bien por los resultados que proporcionan, o bien por las mejoras en tecnología que hacen que disminuya el tiempo computacional de las múltiples operaciones que se realizan en estas “cajas negras”, proliferando sus aplicaciones.

Capítulo 4

Gestión autómata de la ocupación de salas

Durante el desarrollo de este capítulo se va a llevar a cabo el estudio del problema de gestión de salas en la empresa Efor mediante la implementación de CNNs poniendo en práctica la teoría mencionada a lo largo del trabajo. Efor es una empresa aragonesa que forma parte del grupo integra dedicada a dar servicios y soluciones tecnológicas para la gestión, comunicación y marketing de las empresas. El tratamiento de este problema de una manera autómata y con herramientas actualmente emergentes y en crecimiento hace efectivo su eslogan "la innovación es necesaria".

4.1. Descripción y metodología del proyecto

Efor cuenta con una serie de salas para la atención de clientes, reuniones, formación,... Estas salas se pueden reservar mediante una intranet en la cual aparecen tanto los horarios disponibles como las franjas de horas en la que están las salas ocupadas. La problemática a resolver comienza aquí.

En alguna ocasión, la reserva de salas se programa de manera periódica, de modo que, si por algún motivo no se puede llevar a cabo la finalidad por la que fue reservada, la sala figura como ocupada pero no se usa. Salvo que la sala se deje libre de manera análoga a la que se reservó, aparecerá como ocupada y nadie podrá emplearla. La anulación de una visita de clientes o el cambio del horario y reserva de otra sala si anteriormente se disponía de una, también pueden dar lugar al problema anterior.

El principal motivo del presente trabajo es la automatización del conocimiento de la ocupación de las salas sin la necesidad de molestar ni desplazarnos al asegurarnos nosotros de manera presencial si la sala está vacía y la podemos emplear. Para llevarlo a cabo, contamos con una serie de vídeos, de los cuales sacaremos *frames* para poder trabajar con estas imágenes y aplicarles las CNNs para obtener de la imagen la información de si la sala está vacía o está ocupada. Estos controles se podrían realizar de manera reiterada en intervalos de tiempo homogéneos de cuarto de hora empezando a las 8:00, horario en el que la empresa abre, o implementar una aplicación para que el control solo se realizase si es necesario.

El proyecto se realiza siguiendo el siguiente diagrama de fases. Como se muestra, el proyecto distinguirá principalmente dos fases, la primera donde se realiza un estudio de viabilidad con las herramientas propuestas y una segunda donde se pondrá en marcha las conclusiones obtenidas para la resolución del problema.

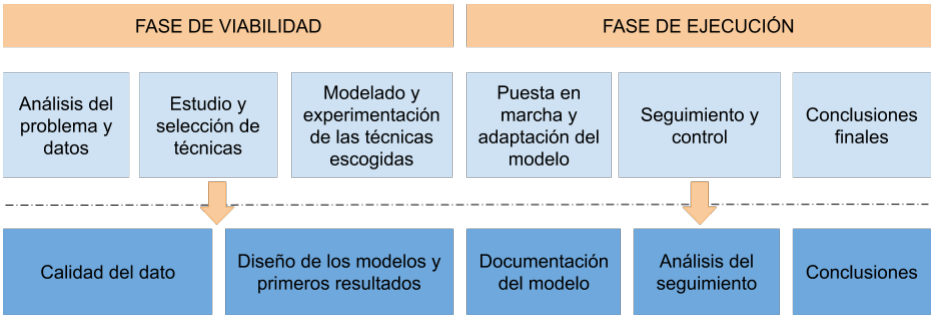


Figura 4.1: Diagrama usual de fases de un proyecto de empresa.

En el estudio de viabilidad afrontaremos el estudio de la aplicación de CNNs a este proyecto. Aplicaremos de manera directa una arquitectura diseñada de CNN con las características que hemos estado mencionando a lo largo de la parte teórica del trabajo. Para ello se realizará un preprocesamiento de la imagen para fragmentarla con el fin de que la entrada de la red definida sean fragmentos de la propia imagen que queremos predecir. La parte práctica del presente trabajo fin de máster (TFM) corresponde a esta primera fase.

Una vez se ha experimentado con la técnica anterior y se ha visto que es factible abordar este experimento con ella, se pondría en marcha la ejecución. Durante esta fase se adaptaría el modelo seleccionado a las diferentes salas de la empresa, entrenando, para ello, el modelo seleccionado en las diferentes salas, generando así un modelo entrenado por sala.

Con la finalidad de enlazar estos modelos, se haría otra red la cual fuese capaz de adjudicar el modelo correspondiente en función de la imagen de la sala. Esta red tomaría el fragmento de imagen donde aparece el número de cámara de vigilancia, asignando a cada uno de esos números el modelo de la sala correspondiente teniendo en cuenta que contamos con una cámara diferente por sala.

Tras un pequeño seguimiento sobre la implantación de estas redes, el proyecto global de la gestión autómata llegaría a su fin.

4.2. Presentación del conjunto de datos y obtención de frames

Para este estudio se han usado una serie de vídeos de la red de seguridad de Efor proporcionados por la propia empresa con el formato de tipo .asf (*Advanced Streaming Format*). Los vídeos extraídos son 3 en diferentes horas y días para tener muestras con diferentes alteraciones de luz en la sala:

Sala	Día	Hora de grabación	Duración (min)	Ocupación	Uso
Boole	4/07/2019	15:30	30	vacía	Entrenamiento
Boole	5/07/2019	9:30	30	ocupada	Entrenamiento
Boole	23/07/2019	13:00	20	ocupada	Validación

Cuadro 4.1: Información sobre los vídeos.

Para obtener los *frames* con los que trabajaremos de estos vídeos se ha utilizado el código mostrado en el cuaderno "Obtención de *frames*" adjuntado en el anexo B. Notar que la ocupación del vídeo no hace referencia a los *frames* que podemos obtener, ya que los vídeos ocupados tienen fragmentos grabados en los que no tenemos personas, bien porque se han grabado los fragmentos de entrada y salida a la sala o bien, porque hemos salido de la sala para tener muestras de la sala sin personas en otro contexto.

Los *frames* extraídos son imágenes (en formato .jpg) de tamaño 1920×1080 obtenidas aproximadamente con una tasa de 15 *frames* por segundo en cada vídeo. Supone un volumen grande del conjunto de datos con los que desarrollaremos el proyecto, pero no debemos olvidar la peculiaridad de que muchos de los *frames* son muy similares por obtenerlos en intervalos tan próximos lo que supone llevar especial cuidado a la hora de tomar el subconjunto de *frames* con los que entrenaremos el modelo.

En alguna ocasión necesitaremos hacer un tratamiento previo en las imágenes para reducir tamaño y resolución para que las imágenes sean más ligeras y los entrenamientos de los modelos sean más rápidos. Esto no supone un impedimento en las predicciones ya que introduciremos las imágenes de una manera análoga a lo que nuestros modelos de redes harán con las imágenes.

4.3. Fase de viabilidad

La fase de viabilidad es la que toma mayor relevancia por ser un punto de inflexión. Gracias a los estudios realizados durante esta fase se obtienen una serie de resultados y conclusiones con los que se conocerá si se pone en marcha el proyecto o por el contrario, este no es factible o en nuestro caso, la técnica propuesta no es capaz de solventar el problema, por resultados o por eficiencia de los modelos.

En esta fase se diseñará una red personalizada para nuestro problema experimentando con diferentes modelos y documentando tanto los experimentos como las conclusiones que nos llevan a seleccionar o descartar las redes. Por consiguiente, diseñaremos redes que sean capaces de clasificar dos clases, ocupada y vacía, que hacen referencia al estado de la sala, afrontando de esta manera el problema como uno de clasificación biclase.

Asimismo se realizará un preprocesamiento de los datos ya que el método empleado para introducir la información al modelo será fragmentando las imágenes de la salas generando ventajas que comentaremos más adelante.

De esta manera, la argumentación a seguir será la siguiente; si la red predice todos los trozos de la imagen como la clase vacía, la sala estará vacía y, si hay al menos un fragmento que pertenezca a la clase ocupada, esté será el estado que tendrá la sala.

El esquema de esta fase experimental sigue la evolución temporal real realizada. Comienza con un análisis y pretratamiento de los datos con los que se prueban varias arquitecturas modificando elementos que varían el número de parámetros del modelo como son el número de capas, filtros y neuronas hasta encontrar una arquitectura sencilla que aportase buenos resultados con los que comenzar a ajustar ese modelo. Durante este proceso de ajuste se pone especial relevancia en el conjunto de entrenamiento con el que aprende la red, el número de épocas, la mejora del modelo con nuevas inicializaciones y el ajuste de la predicción final a través de la modificación del valor de corte en la función de activación de la última capa. Una vez seleccionado el modelo final, realizaremos un estudio de la evaluación de su clasificación más apropiado según la casuística del proyecto y la argumentación seguida para clasificar.

Señalar que los experimentos mostrados a continuación se han realizado con un ordenador con procesador Intel(R) Core (TM) i5-7200U CPU @ 2.50GHz 2.71 GHz, una memoria RAM de 8.00 GB y un sistema operativo de 64 bits.

4.3.1. Estudio de la sala

Para acotar el experimento, aplicaremos esta técnica a una de las salas, la sala Boole. Comenzamos estudiando la sala tanto con los *frames* obtenidos con los vídeos como de forma presencial.

La sala Boole tiene la siguiente distribución donde nos encontramos igualmente los objetos con los que cuenta la sala que con la posición de la cámara de donde hemos obtenido los datos.

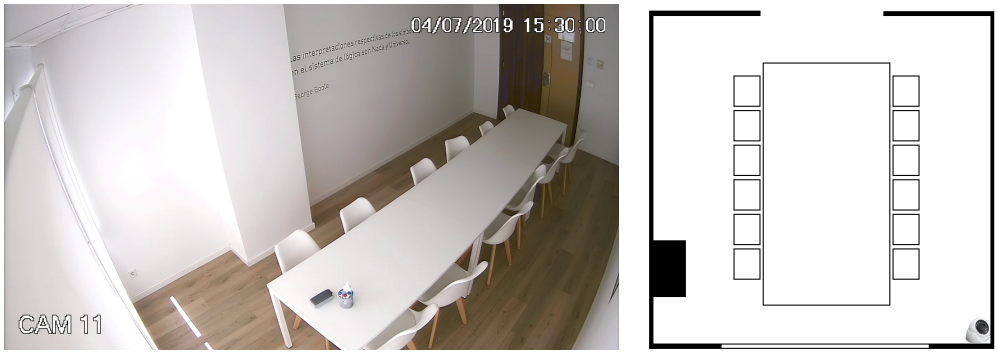


Figura 4.2: Distribución de la sala Boole.

Debido al tamaño de las imágenes adquiridas en los vídeos, así como lograr una pequeña optimización en los tiempos de ejecución y prevenir falsos negativos cuando la sala esté vacía, se ha optado porque la entrada de la red definida sean fragmentos de los *frames* y no estos completos. De esta manera, se logra optimizar los tiempos de ejecución ya que el tamaño de las imágenes se estará reducido significativamente y se eliminan fragmentos que contienen espacios donde no vamos a encontrar personas, como son los fragmentos del techo, consiguiendo además, evitar falsos negativos.

El estudio de la sala en la que se aplicará la red es importante puesto que, dependiendo de la perspectiva, localización de la cámara de seguridad, luz o tamaño de la sala puede producir la necesidad de aplicar alguna mejora a la imagen. En nuestro caso, el motivo esencial del estudio de la sala radica en aplicar uno u otro mallado a la imagen para segmentarla evitando la perdida de información y ayudando, en la medida de lo posible, a mejorar la predicción.

Para la realización de este estudio se ha de tener en cuenta la forma en la que las imágenes se leen con *Python* en los cuadernos de *jupyter*, ya que la función que segmenta la imagen se ha programado de manera que los fragmentos de imagen que queden fuera de un mallado regular se pierdan. Este mallado comenzará definiéndose en la parte superior izquierda de la lectura de la imagen pudiendo provocar pérdida de información en la parte derecha e inferior de esta.

Se comenzó proponiendo un mallado compuesto por cuadrados llevando cuidado con el tamaño de estos porque si son muy pequeños nos pueden llevar a confusión. Un ejemplo de un mallado demasiado pequeño es cortar la siguiente imagen que forma parte de los *frames*

obtenidos con un mallado de 65 × 65. El mallado nos representa los trozos extraídos donde los rectángulos pintados nos indican la pérdida de información.

Figura 4.3: Mallado de la sala Boole de dimensiones 65 × 65.

Como vemos la pérdida de información es mínima, sin embargo se plantea otro problema, ya que los siguientes fragmentos pertenecientes al mallado anterior serían difíciles de clasificar, incluso para el ojo humano, en relación a si pertenecen a fragmentos de sala vacía o trozos de las personas de la imagen.

Figura 4.4: Fragmentos del mallado que pueden provocar error.

Con esta información se propuso un mallado de dimensiones 120 × 120. Este mallado era lo suficientemente grande como para solventar el efecto anterior y además nos ofrece la ventaja de no tener pérdida de información en los laterales.

Figura 4.5: Mallado de la sala Boole de dimensiones 120 × 120.

Al realizar experimentos con este mallado se observaba que para mejorar las predicciones se

debía complicar más la arquitectura, lo que suponía un aumento en el tiempo de entrenamiento del modelo pero no mejoraba significativamente las predicciones en cada paso.

El siguiente paso fue ver como la red estaba prediciendo para mejorar el mallado. Ejemplos de imágenes de las predicciones que obteníamos eran:

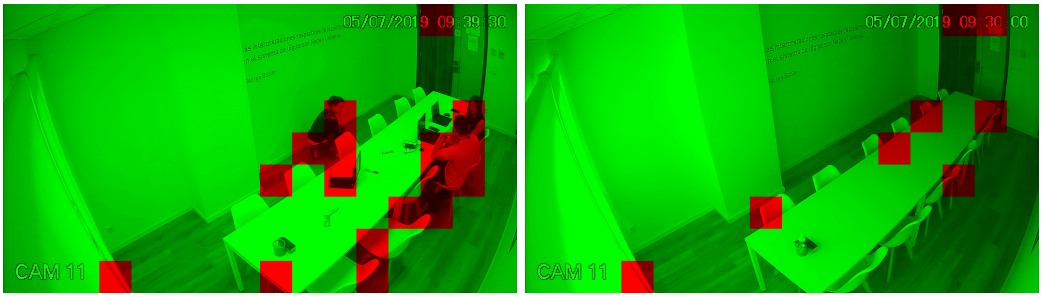


Figura 4.6: Imágenes de predicciones de una red con dos capas convolucionales y dos capas completamente conectadas.

Como veremos en profundidad más adelante, las zonas verdes son aquellas que la red predice como fragmentos de sala vacía y los rojos de sala ocupada. Se puede ver en las imágenes anteriores como hay zonas de la sala vacía que nos la predice como ocupada, produciéndose un falso negativo y zonas de la sala ocupada, como son trozos de la mesa y sillas, que predice como si estuviese ocupada.

Para mejorar estas predicciones, se observó que la figura de las personas se asemeja más a un rectángulo, introduciendo de esta manera más información en cada iteración de entrenamiento resultando más fácil la distinción de los fragmentos y reduciendo los falsos negativos. Finalmente, haciendo pruebas con varios mallados rectangulares, se opto por uno de dimensiones 120 $\times$ 240. De esta forma, los fragmentos de las imágenes serían los resultantes de romper la imagen por el siguiente mallado:

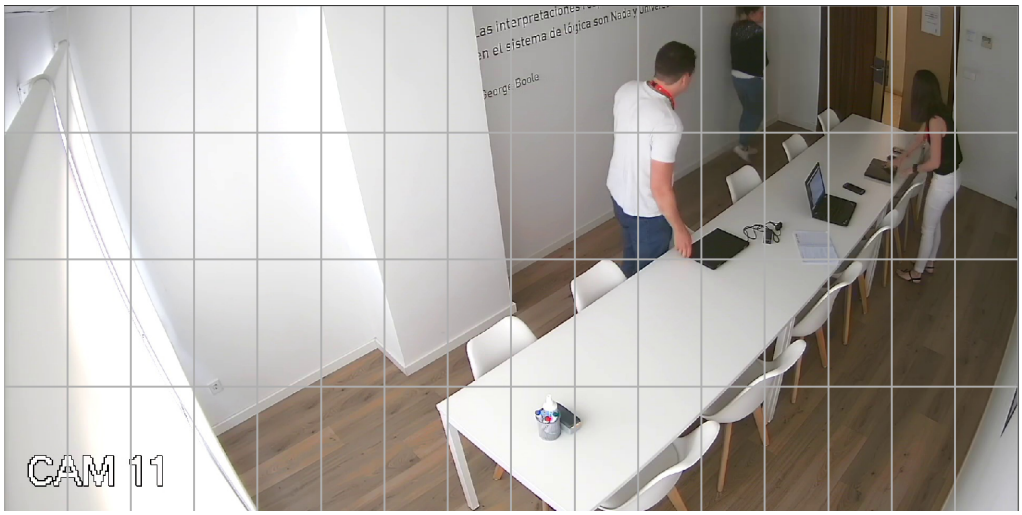


Figura 4.7: Imágenes del mallado finalmente seleccionado de dimensiones 120 $\times$ 240.

El código empleado para la realización de este estudio se puede ver en el cuaderno “Estudio de la sala” del anexo B.

4.3.2. Estudio de los datos y obtención de los conjuntos del experimento

Tras escoger el mallado con el que cortar la imagen, tenemos que formar los conjuntos con los que el modelo de la red trabajará. En nuestro caso, se realizará cada experimento con dos conjuntos de datos, el conjunto de entrenamiento y el conjunto de validación. Por la forma en la que la red está programada junto con la lectura de los datos, cada uno de estos conjuntos estará dividido en dos carpetas, una por cada clase de clasificación, que en nuestro caso serán dos (vacía y ocupada).

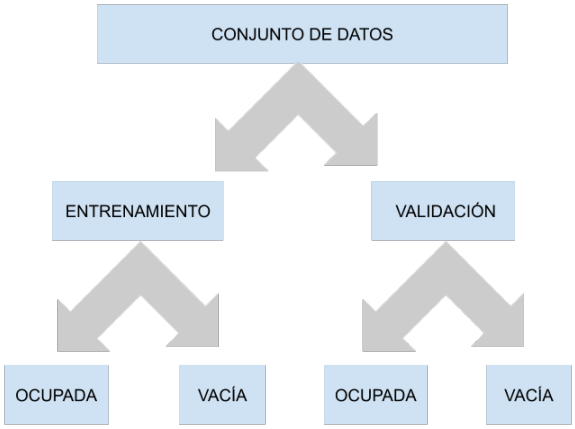


Figura 4.8: Estructura de los datos utilizada.

Para obtener estos conjuntos, se comenzó con la elección manual del los *frames* para cada uno de ellos. Esta selección manual condicionó las imágenes que posteriormente formaban los conjuntos. Para solventar esto, se opto por seleccionar de manera aleatoria las imágenes para cada uno de los conjuntos.

Tras obtener el número de los *frames* que pertenecen a los diversos conjuntos, se procedió a guardarlos en distintas carpetas para proceder a la inspección visual y selección manual de cada uno de los fragmentos de esas imágenes que finalmente sí pertenecerían a los conjuntos finales. Se puede consultar el código utilizado para este proceso en el cuaderno "Preparación de los conjuntos de entrenamiento y validación" del anexo B.

La selección manual fue variando a lo largo de las diferentes experimentaciones. Se comenzó seleccionando sin mucho detalle pero se observó modificando los conjuntos del modelo, que conforme se estudiaba un poco la selección de estos conjuntos, la red daba mejores resultados. Por tanto, se hizo teniendo en cuenta que aproximadamente más del 70 % del fragmento de la imagen tuviese elementos representativos de la clase a la que iba a representar (en el caso de entrenamiento) siendo un poco menos estrictos en el caso de los conjuntos de validación. La selección realizada es debida a que nos interesa que la red aprenda con un buen conjunto de entrenamiento, pero a la hora de validar el modelo, con el conjunto de validación, queremos poner al límite la red para ver si la clasificación de esta se adapta mucho al conjunto de entrenamiento.

(a) ocupada (b) ocupada (c) ocupada (d) vacía (e) vacía (f) vacía

Figura 4.9: Ejemplos de imágenes del conjunto de entrenamiento y sus etiquetas.

Finalmente se seleccionó un conjunto de entrenamiento con 1000 elementos para cada una de las dos clases que configuran nuestro proyecto y un conjunto de validación de 200 por clase, lo que forma un total de 2000 imágenes de entrenamiento y 400 para una primera validación inicial.

4.3.3. Creación, aplicación y evaluación de las CNNs

En esta parte correspondiente de la fase de viabilidad se diseñan los modelos que queremos aplicar para resolver nuestro problema, de los cuales seleccionaremos el modelo que nos dé mejores resultados. Con este fin, se han ido diseñando diferentes arquitecturas, entrenando y validando los modelos ayudándonos de los cuadernos "Diseño, entrenamiento y primeros resultas de la CNN" y "carga del modelo y clasificación" adjuntos en el anexo C.

Notar que el primer criterio para seleccionar un modelo es tener una tasa de bien clasificados superior al 70 % de manera que aquellos modelos que no lo superen los descartaremos de ser nuestro modelo final automáticamente sin la necesidad de estudiar ninguna otra métrica del modelo. Otra de las características de la arquitectura que fijamos al comienzo de los experimentos, por ser una clasificación biclase, es usar como capa de salida una única neurona con la función de activación sigmoide. De esta manera, se predice la clase ocupada si el valor resultante de la red es menor que 0.5 y vacía si es mayor.

Se comenzó diseñando arquitecturas de red para el caso del mallado compuesto por cuadrados. Durante estos experimentos la principal característica de la arquitectura que obtuvimos es la dimensión de los filtros. Estos no debían ser muy grandes ya que el tamaño de nuestras imágenes es pequeño porque la entrada son fragmentos de la imagen original obtenida. Otra información que pudimos deducir de los experimentos es que las etapas de *pooling* no suponían una gran mejora en la clasificación por tener imágenes tan pequeñas, lo que hizo agilizar nuestros modelos al eliminar estas fases.

Enseguida se pudo ver, como comentamos durante el estudio de la sala, que las predicciones se podían mejorar no solo cambiando la arquitectura de nuestros modelos, sino cambiando la forma en la que introducimos la información, es decir cambiando el mallado pasando a estar compuesto por rectángulos. Para probar nuevos modelos en esta línea se aplicó la arquitectura del mejor modelo conseguido hasta el momento a un nuevo conjunto de datos con fragmentos de imágenes en secciones rectangulares obtenidos de los mismos *frames* con los que se habían obtenido los buenos resultados de la red. La arquitectura usada para esta comparación fue:

Arquitectura de la CNN
Convolución con 6 filtros de 3×3 Función de activación ELU
Convolución con 6 filtros de 3×3 Función de activación ELU
Flattening
Capa completamente conexa de 512 neuronas & Sigmoide
Capa completamente conexa de 256 neuronas & Sigmoide
Capa de salida 1 neurona & Sigmoide

Cuadro 4.2: Arquitectura de la CNN usada para comparar la entrada en cuadrados y en rectángulos.

Al usar los mismos *frames* en ambos casos para la obtención de los fragmentos de imagen con los que la red iba a trabajar, las características del experimento quedan ligeramente modificadas para las distintas entradas viniendo resumidas en el siguiente cuadro. La reducción del volumen de elementos en la entrada rectangular es debida al aumento del tamaño de los fragmentos de la imagen.

Tipo mallado	dimensiones	nº de elementos de entrenamiento	nº de elementos de validación
Cuadrado	120×120	2002	178
Rectángulo	120×240	1000	102

Cuadro 4.3: Cuadro resumen de la adaptación del mismo modelo para dos mallados distintos.

Tras hacer la adaptación y el entrenamiento para 20 épocas en estos dos conjuntos de datos, los primeros resultados obtenidos para cada experimento vienen resumidos en el cuadro 4.4. Destacamos el aumento de parámetros del modelo al cambiar la entrada, es decir, una mayor complejidad del modelo empleado pero un tiempo de entrenamiento muy similar.

Tipo mallado	nº parámetros	Tiempo de entrenamiento	ACC de la última iteración (entrenamiento)	ACC de la última iteración (validación)
Cuadrado	41469427	37 min 50 seg	0.8328	0.7472
Rectángulo	84231667	37 min 3 seg	0.8093	0.7647

Cuadro 4.4: Cuadro resumen comparativo de los resultados con la misma arquitectura.

Realizamos un estudio sobre la evolución de ambos modelos para cada experimento en función de la función de pérdida y de la tasa de bien clasificados o *accuracy* con respecto a las épocas. Para el modelo cuyas entradas son fragmentos cuadrados de los *frames* obtenemos las siguientes variaciones:

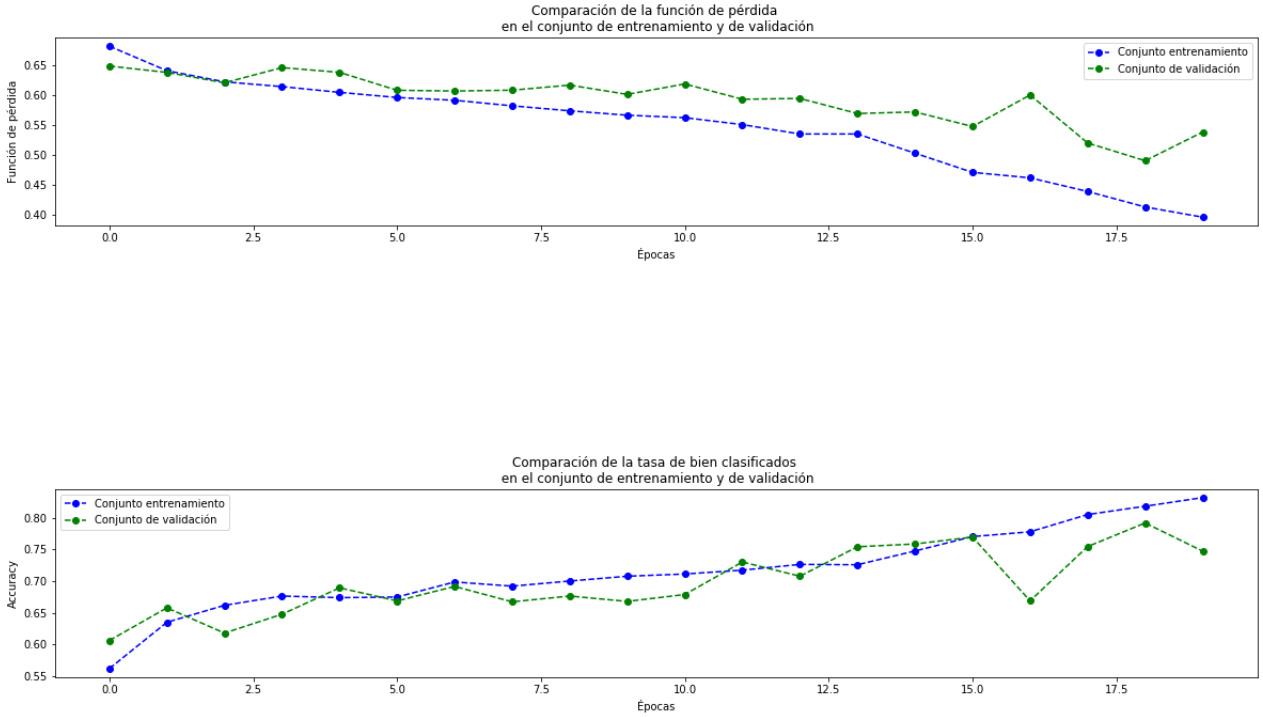


Figura 4.10: Variación de error y tasa de bien clasificados del modelo con entrada en cuadrados.

La evolución del modelo es bastante lineal salvo un pequeño pico en la época 16. Debemos destacar, además, lo ocurrido en la última época. Parece que la tendencia del conjunto de validación es aumentar el error de clasificación (aumenta la función de pérdida) y disminuir la tasa de bien clasificados al contrario de lo ocurrido con la tendencia del conjunto de entrenamiento; se debe a que el aumento en las épocas de entrenamiento puede producir un sobreajuste respecto al conjunto de entrenamiento, es decir, se adapta muy bien a ese conjunto y como consecuencia, si las nuevas imágenes no son muy parecidas al conjunto de entrenamiento, el modelo las clasificará mal. Hecho muy importante en nuestro caso particular ya que un pequeño cambio en la posición de las sillas al dejar la sala vacía nos daría una mala clasificación.

Con estos resultados deducimos que, para este caso particular del mallado de la imagen en cuadrados, no se necesitan más épocas ya que no va a aprender más el modelo sino que, se producirán sobreajustes. Para solucionarlo, se podría modificar el conjunto de datos con el que entrenar para hacerlo más heterogéneo, seguir modificando la arquitectura de la red (nuevos pesos, funciones de activación,...) o, como en nuestro caso, estudiar otro mallado para las imágenes. Seguiremos la evolución del experimento en esta última línea (por la mejoría de esos modelos en los resultados que iremos mostrando a continuación).

Haciendo el mismo estudio gráfico en el caso del mallado rectangular, los resultados de la variación en el modelo son:

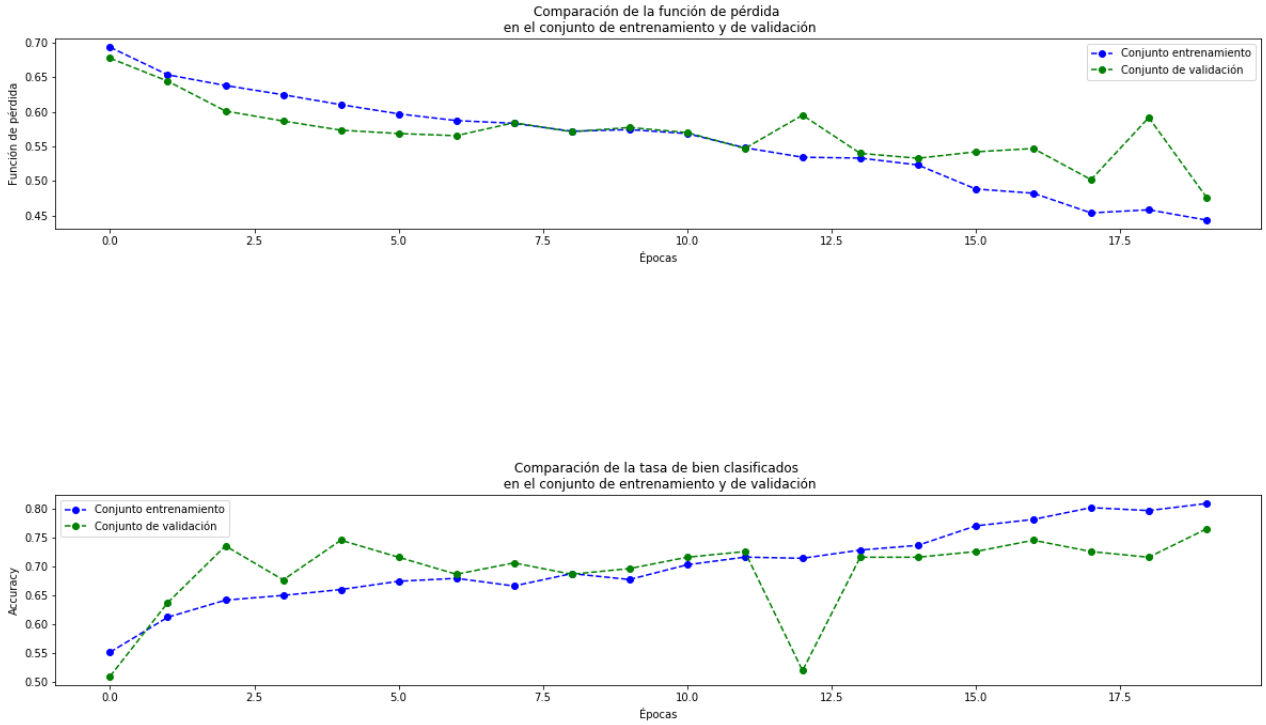


Figura 4.11: Variación de error y tasa de bien clasificados del modelo con entrada rectangular.

Al contrario que en el caso anterior, la evolución es más inestable lo que nos lleva a pensar en aumentar el número de épocas para los siguientes entrenamientos y así dejar que el modelo se estabilice. Destacar, en la época 12, un pequeño aumento en la función de error produce una disminución en la tasa de bien clasificados notable por lo que nos interesa esa estabilidad del modelo.

Observamos además, como en las últimas épocas, la tendencia de la función de error es decreciente y la tasa de bien clasificados es creciente lo que lleva a pensar que estos modelos pueden seguir entrenando sin producir sobreajustes. Este modelo de red podrá entrenar durante más épocas consiguiendo mejores resultados en las clasificaciones.

Tras hacer las analíticas de los modelos, también podemos hacer un estudio visual de la predicción de las clases en las imágenes. Con este pequeño análisis podemos ver las zonas problemáticas donde las redes fallan pudiendo así cambiar el conjunto de entrenamiento reforzando estas zonas. Las imágenes de este estudio se obtienen modificando los canales de color de la imagen a estudiar, dejando solo el canal verde o rojo haciendo cero al resto de valores. El color dependerá de la clase con la que la red clasifica el fragmento de imagen, siendo rojo si la clase predicha es ocupada y verde en caso contrario originándonos una idea rápida de la clasificación del modelo.

Tomamos dos *frames* del vídeo ocupado que no forman parte del conjunto de entrenamiento ni validación para hacer este estudio visual en ambos modelos.

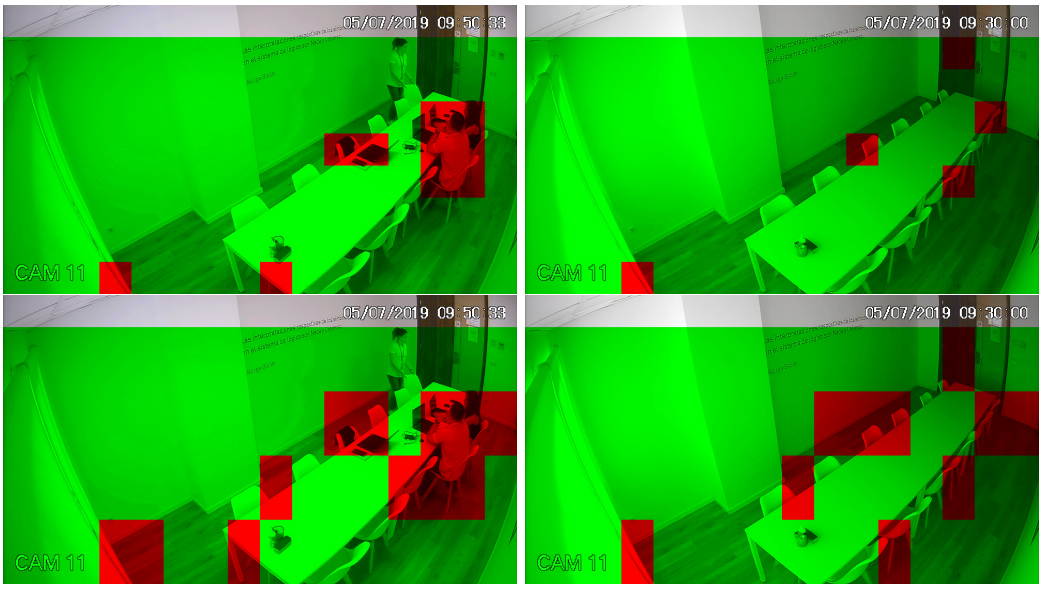


Figura 4.12: Estudio visual de las clases predichas por los modelos. Las imágenes superiores corresponden al modelo con entrada en forma de cuadrados y las inferiores al modelo con entrada rectangular.

Puede parecer que las primeras imágenes son mejores que las segundas, pero no debemos olvidar que es un modelo que se había hecho para mejorar la clasificación para trozos de imágenes cuadradas y que en el caso rectangular admite más épocas de entrenamiento, lo que nos permite mejorar mucho las clasificaciones adaptando el modelo al nuevo tipo de datos. Esta percepción es debida a que en la zona rectangular hay más área pintada de rojo y asociamos el rojo a elementos erróneos. La dificultad del diseño y ajuste de este tipo de redes reside en este punto; hay veces que malos resultados se puede mejorar notablemente produciendo un pequeño cambio.

Las apreciaciones que no debemos pasar por alto para comparar las predicciones en estas imágenes son:

- Hay una persona que pasa desapercibida en la clasificación de ambos modelos.
- Las zonas de error aproximadamente son las mismas; las zonas de borde de la mesa junto con las zonas de sillas. En esta comparación debemos considerara que estas zonas se evitan más fácilmente con el mallado rectangular ya que se puede aportar más información en cada iteración sin correr el peligro de tener un sobreajuste.

Con estos resultados no podemos distinguir sin incertidumbre cual es mejor aunque, parece que la clasificación del modelo del mallado rectangular es preferible ya que, la clasificación para nuevos elementos (tasa de bien clasificados en el conjunto de validación) es mejor en sentido analítico.

Debemos recalcar que la asignación de los pesos iniciales es aleatoria, lo que puede mejorar notablemente el modelo. Para ello, podemos entrenar varias veces los modelo y observar los resultados para poder ver en media si efectivamente se avanzaría y se mejoraría con una arquitectura o con otra, obteniendo cierta información sobre la dependencia de la inicialización de los pesos en nuestros modelos.

Para observar este fenómeno, se ha evaluado varias veces el modelo anterior para el mallado formado por rectángulos con el mismo conjunto de datos. Una muestra de los resultados de estas evaluaciones son:

Identificador de inicialización	nº parámetros	Tiempo de entrenamiento	ACC de la última iteración (entrenamiento)	ACC de la última iteración (validación)
i_1	84231667	37 min 3 seg	0.8093	0.7647
i_2	84231667	38 min 47 seg	0.7737	0.7255
i_3	84231667	38 min 30 seg	0.7871	0.6961
i_4	84231667	1h 14 min 8 seg	0.8153	0.7451

Cuadro 4.5: Cuadro resumen del efecto de la aleatoriedad de los pesos iniciales.

En la cuarta ejecución, el aumento del tiempo de entrenamiento es debido a que se ejecutó con el portátil sin conectar a la luz, lo que conlleva un menor rendimiento al depender de la batería. La agilidad de este tipo de modelos dependen en gran medida de la potencia y el rendimiento del ordenador.

La variabilidad de los valores de la tasa de bien clasificados no es alta. Por este motivo, se puede pensar que un modelo con estas características alcanzaría aproximadamente una tasa de bien clasificados del 79.6 % para el conjunto de entrenamiento y un 73.2 % para el de validación.

Con la ayuda de estas conclusiones, se optó por fijar esta arquitectura para el mallado rectangular por el buen comportamiento que parecía tener. Para lograr una mejora en la clasificación se pasó a adaptar el conjunto de entrenamiento a la entrada de la red, como se mostró en la parte final de la obtención de los conjuntos del experimento, siendo más estrictos a la hora de coger los fragmentos de la red etiquetados como clase ocupada y generando al azar los *frames* utilizados. De esta manera se probó la arquitectura anterior con los nuevos conjuntos de datos formados.

Para proceder a este estudio con una arquitectura de red ya fijada se pasó a fijar semillas para poder reproducir los resultados e ir ajustando el modelo. Hasta el momento, este hecho no era relevante ya que la asignación aleatoria de los pesos del modelo nos daba un conocimiento general del comportamiento del modelo y podíamos observar, como en el estudio del cuadro 4.5, si la tasa de bien clasificados varía significativamente o no. Esta aleatoriedad nos aportaba información de la robustez modelo y, además, los modelos podían guardarse si se descargaban los pesos y se era conocedor de la arquitectura. Ahora, necesitamos poder reproducir el entrenamiento de la red con el fin de ajustar el modelo. Por esta razón las semillas toman importancia en este punto del experimento.

Con este pequeño cambio en el conjunto de entrenamiento podemos ver como los resultados tienen mejoras bastante significativas. Para el conjunto de entrenamiento se clasifica bien un 90 % de los datos y para el de validación un 87 % en comparación con 80.93 % y 76.47 % obtenidos anteriormente.

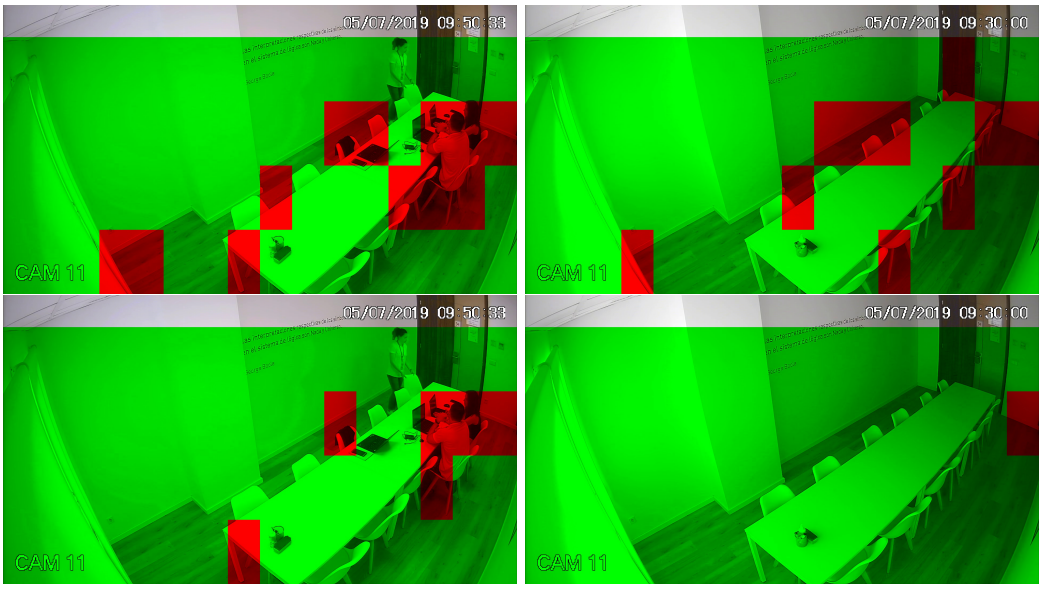


Figura 4.13: Comparación del estudio visual de las clases predichas por el mismo modelo cambiando únicamente los conjuntos con los que trabaja la red. Las imágenes superiores corresponden a las predicciones hechas anteriormente y las inferiores al modelo con los nuevos conjuntos.

Hacemos un estudio más en profundidad de los resultados obtenidos. Por un lado, podemos calcular la matriz de confusión del modelo sobre el conjunto de validación. Esta matriz presentada no toma exactamente la misma forma que la nombrada en la teoría porque ha sido normalizada de manera que todos sus elementos suman uno. Teniendo en cuenta que cada fila representa el 0.5 de los datos por la normalización observamos que, en los elementos etiquetados como vacíos, la predicción de clasificación acertada por la red es casi del 100 % de los elementos (representaría alrededor del 98.5 %). Sin embargo, para el caso de los elementos etiquetados como ocupados la precisión no es tan buena. Según se muestra en el estudio visual anteriormente mostrado de este modelo, aún sigue clasificando mal el trozo de imagen de la persona donde ya fallaba, error bastante importante.

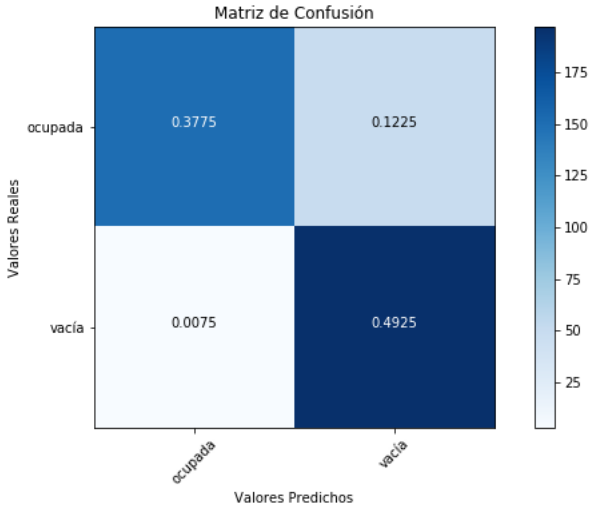


Figura 4.14: Matriz de confusión normalizada para el modelo de 20 épocas.

Por otro lado, podemos calcular la evolución del modelo respecto a las épocas. Se observa el mismo efecto que ya hemos comentado con anterioridad, a partir de este momento parece que la tendencia de las variaciones va a producir un sobreajuste.

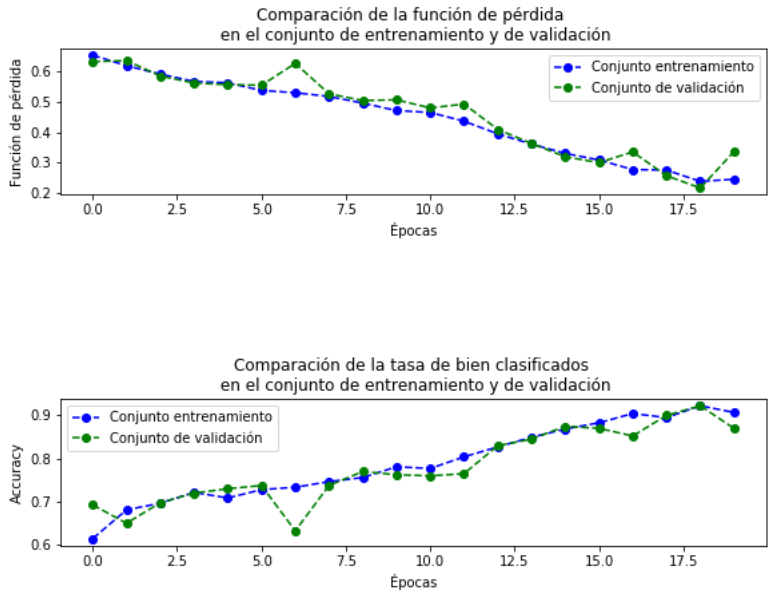


Figura 4.15: Variación de error y tasa de bien clasificados del modelo fijado con 20 épocas.

Como teníamos fijada la semilla, podemos reproducir el resultado para un número de épocas mayor y ver si se produce ese fenómeno o se estabilizan las variaciones de la función de pérdida y de *accuracy* con la finalidad de fijar el número de épocas óptimo para el modelo. Repetimos el experimento con 30 épocas.

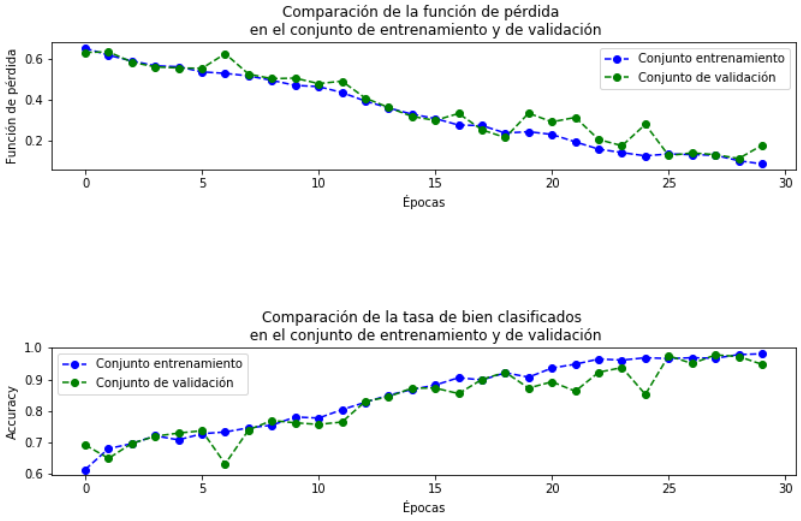


Figura 4.16: Variación de error y tasa de bien clasificados del modelo fijado con 30 épocas.

En este último gráfico se observa como esa variación de la última época del modelo entrenado en 20 épocas se acaba estabilizando produciendo, en varias épocas más, una mejora en la *accuracy* y reduciendo la función de pérdida. A partir de la época 26 (contando desde 1) el modelo estabiliza los resultados pudiendo producir otro desajuste pasada la época 30.

Aprovechamos esta estabilidad para estudiar el modelo en uno de estos puntos. El número de épocas elegido para realizar un estudio sobre la mejora del modelo es de 27 ya que queda en un punto central de la estabilidad donde no tiene grandes desequilibrios en su entorno.

Tras fijar este parámetro y entrenar nuevamente el modelo, calculamos la matriz de confusión para el conjunto de validación.

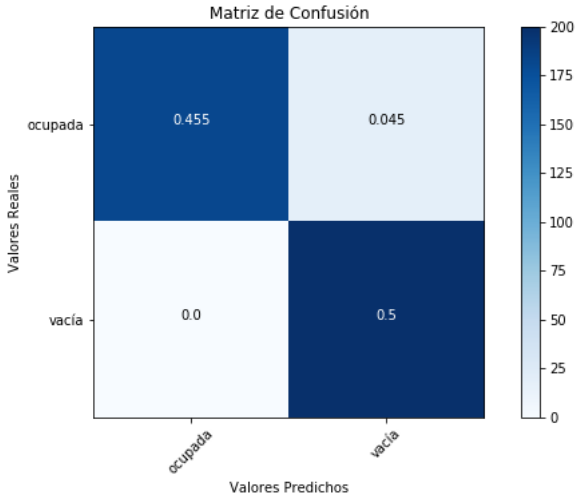


Figura 4.17: Matriz de confusión normalizada para el modelo de 27 épocas.

La matriz de confusión para los datos del conjunto de validación es realmente buena. Se observa que la predicción de la clasificación de los elementos etiquetados como vacía es correcta en el 100 % de los casos y para los datos etiquetados como ocupada del 91 %.

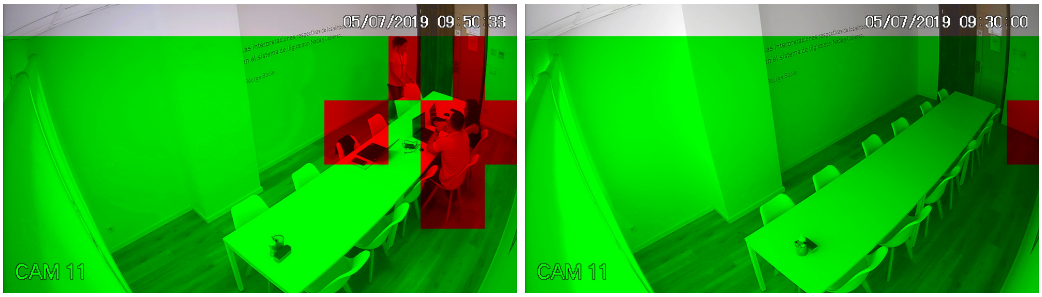


Figura 4.18: Estudio visual de las clases predichas por el modelo de 27 épocas.

El estudio del modelo con 27 épocas dio muy buenos resultados para el conjunto de validación sin embrago, si hacemos un estudio visual de un frame que no pertenece a ninguno de los dos conjuntos (entrenamiento y validación), como hemos visto en la figura 4.18, no se obtiene una buena clasificación ya que en la imagen de la sala vacía hay un fragmento clasificado como ocupado, lo que conlleva una mala predicción el estado de la sala. Como consecuencia, debemos seguir ajustando un modelo que solucione nuestro problema.

Seguimos estudiamos el modelo para otras de las épocas donde el modelo estabilizaba su variación, obteniendo peores resultados al modelo anterior salvo para 29 épocas. Para este modelo, la matriz de confusión para el conjunto de validación mejoró ya que había un mayor porcentaje de aciertos en la clase ocupada manteniéndose los buenos resultados para la clase vacía.

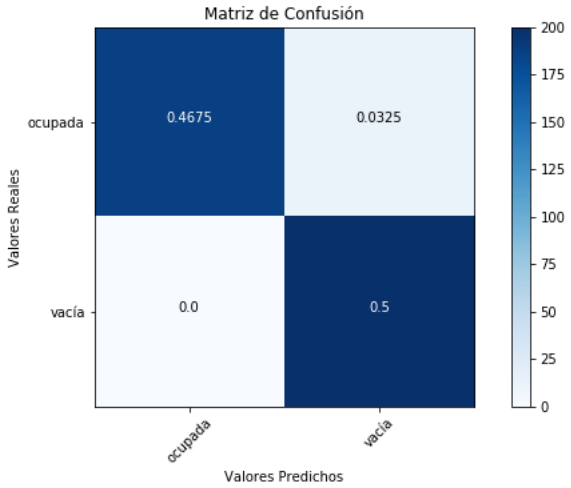


Figura 4.19: Matriz de confusión para el modelo de 29 épocas.

Con este modelo conseguimos el objetivo que queremos, todos los fragmentos de un frame que no ha sido utilizado en el modelo de sala vacía son clasificados bien. Pero este modelo presenta un problema, si predecimos un frame del vídeo empleado para la validación del modelo en profundidad vemos como la predicción con la sala vacía no funciona bien.

Figura 4.20: Estudio visual de las clases predichas por el modelo de 29 épocas. En el caso de la derecha para un frame del vídeo utilizado para el diseño y entrenamiento del modelo y en el de la izquierda un frame del vídeo utilizado solo para la validación y estudio del comportamiento del modelo.

Llegados a este punto, como la función de perdida alcanzada en la última época del modelo es de 0.1057 para el conjunto de entrenamiento y 0.1202 para el conjunto de validación decidimos reducir este error para encontrar un modelo mejor. Para ello cambiamos la semilla y volvemos a inicializar los pesos con la intención de que el modelo mejore globalmente fijando las épocas en 30.

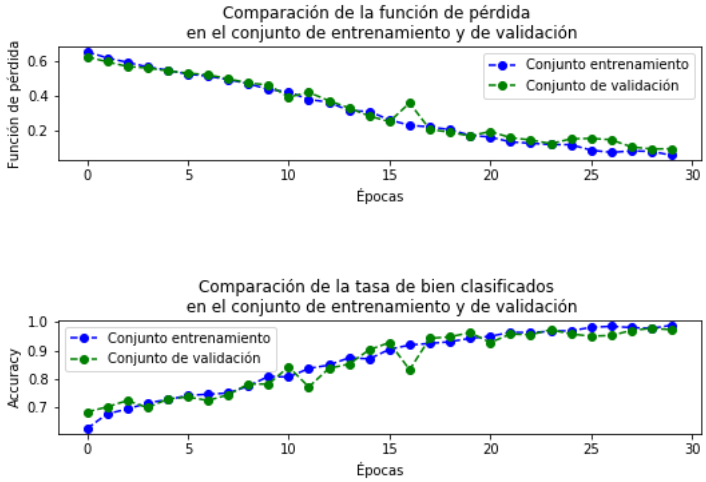


Figura 4.21: Evolución del modelo con 30 épocas con la nueva inicialización de los pesos.

La variación a lo largo de las épocas de este modelo es casi inexistente alcanzando el valor de 0.0516 para la función de pérdida del conjunto de entrenamiento y 0.0898 para el conjunto de validación. De este modo conseguimos reducir la función de pérdida obteniendo además una tasa de bien clasificados del 98.85 % para el conjunto de entrenamiento y 97.25 % para el conjunto de validación.

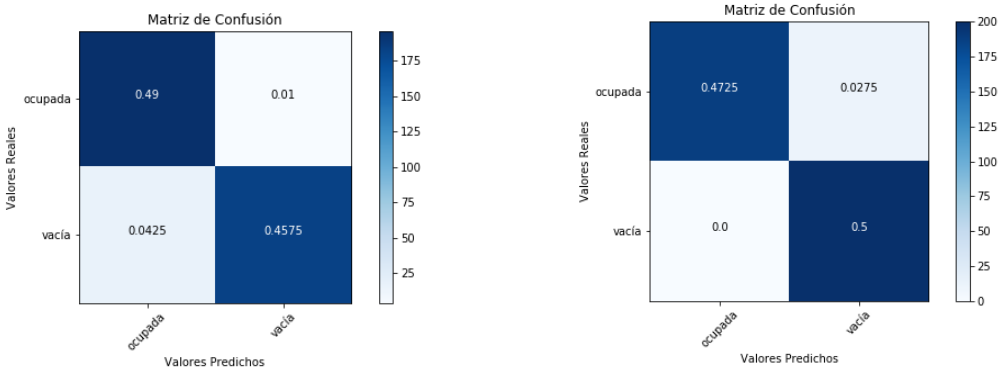


Figura 4.22: Matrices de confusión del modelo inicial con 30 épocas (izquierda) y el de 30 épocas con la nueva inicialización de los pesos (derecha).

Al aumentar la complejidad del modelo con una época más y cambiando los pesos iniciales, mejoramos el modelo respecto al de 29 épocas aumentando la tasa de bien clasificados de la clase ocupada y manteniendo los buenos resultados de la vacía. Si comparamos los modelos de 30 épocas con las dos inicializaciones de los pesos mediante la matriz de confusión observamos como el modelo con los pesos antiguos mejora la clasificación de la etiqueta ocupada pero no mantiene los buenos resultados de la vacía. Podríamos mejorar el modelo de 30 épocas antiguo para obtener una mejor clasificación en ocupada pero por la casuística de nuestro proyecto es más fácil ajustar la clase vacía ya que a excepción del cambio de posición de las sillas, la clase es más homogénea y tiene menos variabilidad en los fragmentos. Con estos buenos resultados obtenidos, el modelo de 30 épocas con la nueva inicialización de los pesos es con el que nos quedaremos.

4.3.4. Validación del modelo seleccionado

Tras hacer una pequeña primera validación del modelo con la matriz de confusión sobre el conjunto de validación que representaba aproximadamente el 20 % de los datos usados para el diseño del modelo, se procedió a hacer un estudio más a fondo para analizar los buenos resultados del modelo.

Se procedió ha hacer un estudio visual de algunos *frames* como habíamos hecho con algunos de los modelos del apartado anterior para valorar las zonas de error en la predicción de la clasificación.

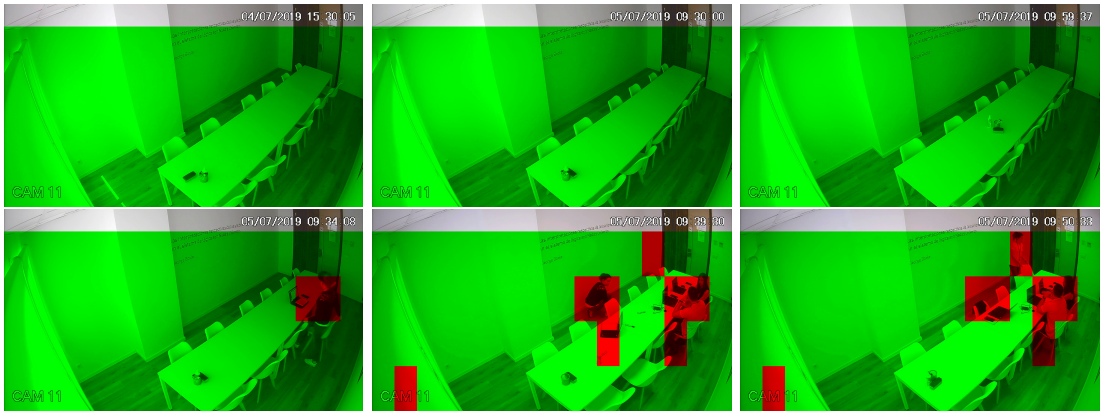


Figura 4.23: Estudio visual de las predicciones de la clasificación del modelo seleccionado.

Las predicciones realizadas en *frames* con la sala vacía de la clasificación es correcta en todos los casos. En *frames* de la sala ocupada observamos como la red predice bastante bien la clase ocupada a excepción de fragmentos realizados con el letrero de la cámara de vigilancia, que no debe preocuparnos porque podríamos recortar la imagen como hemos hecho con la zona del techo, y un fragmento en la imagen central por la posición de las sillas.

Una vez realizado este pequeño estudio, realizamos otro estudio visual, pero esta vez con los *frames* del vídeo utilizado para la validación. Este vídeo presenta otra disposición de las sillas diferente a los usados para el entrenamiento y validación, así como nuevas personas que hasta este momento la red no ha visto. Si comenzamos con algún *frame* de la sala vacía veremos como la predicción es errónea.

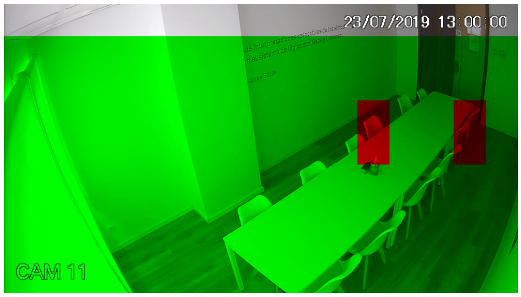


Figura 4.24: Estudio visual de las predicciones de un *frame* de la sala vacía del vídeo de validación.

Este efecto es producido por los datos con los que contamos. Aunque tengamos muchas imágenes, las posiciones, por ejemplo del vídeo de la sala vacía, no varían ya que la sala está

completamente vacía durante todo el vídeo. Eso implica que el modelo pueda dar pequeños fallos en este aspecto. Para evitarlo, cambiamos la forma en la que la red predice. En vez de utilizar el razonamiento de si es menor que 0.5, la clasificación será ocupada y en caso contrario vacía, disminuiremos ese punto de corte al valor de 0.2. De esta manera conseguimos que las predicciones realizadas como ocupada sean reales evitando el efecto de las sillas ya que las predicciones de la red en estos casos oscilaba el valor de 0.4.

La selección de este nuevo punto de corte se ha realizado ayudándonos del cálculo de la curva ROC para valores a partir de 0.1 hasta 0.6 con una distancia entre ellos de 0.1 sobre las predicciones de un conjunto de 929 elementos seleccionados aleatoriamente sobre los fragmentos de los *frames* extraídos del vídeo utilizado para la validación. En la selección de este conjunto se ha optado por no lograr un equilibrio en el número de clases para que el estudio sea más realista ya que en las predicciones de las imágenes, por lo general, hay mayor número de elementos clasificados como "vacía" que como "ocupada".

Notar que se debe calcular la curva ROC sobre un valor de decisión creciente, por ese motivo los valores representados se invierten para su cálculo. Para ver el código de su construcción se puede consultar el cuaderno "Análisis de la curva ROC" del anexo C.

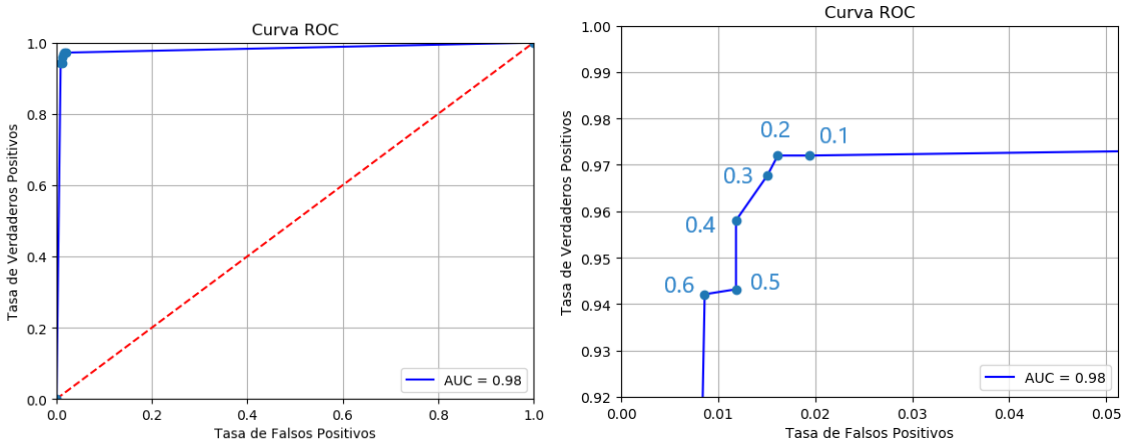


Figura 4.25: Curva ROC y ampliación del extremos superior izquierdo.

Se puede observar que el mejor valor de corte, que corresponde con el punto de menor distancia a la esquina superior izquierda del gráfico, es el que toma un valor de 0.2. Estimación que, como hemos mencionado anteriormente, hemos escogido para cambiar la predicción de nuestro modelo.

Volvemos a hacer el mismo estudio visual de la figura 4.23 para ver el efecto que tiene este cambio en las predicciones realizadas.

Como vemos en la siguiente figura, estas predicciones quedan ligeramente modificadas mejorando la clasificación de los fragmentos problemáticos que habíamos mencionado anteriormente. Al ser más estrictos a la hora de clasificar un fragmento como "ocupada" se necesitará mayor porcentaje de elementos representativos que no hagan a la red clasificarlo como "vacía". Este efecto se puede ver en la imagen inferior central donde ahora el área pintada de rojo es menor.

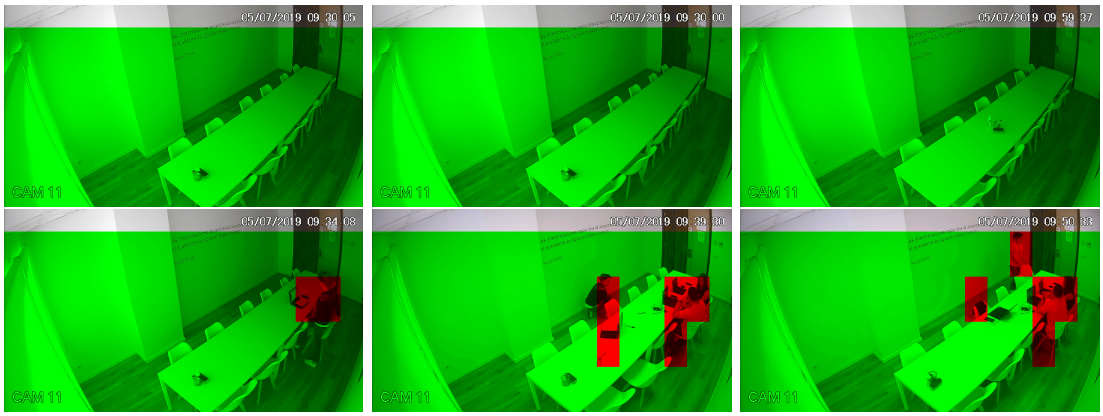


Figura 4.26: Estudio visual de las predicciones de la clasificación del modelo seleccionado.

Este impacto también se puede ver en las matrices de confusión (con mayor facilidad en las matrices sin normalizar). Como en el conjunto de validación habíamos sido menos estrictos a la hora de etiquetar los fragmentos de la clase ocupada, la tasa de bien clasificados disminuirá en el caso de utilizar el valor de corte de 0.2. Se deduce de este estudio que el número de fragmentos etiquetados como ocupados que perderíamos por el cambio del valor de corte como vemos no es muy relevante.

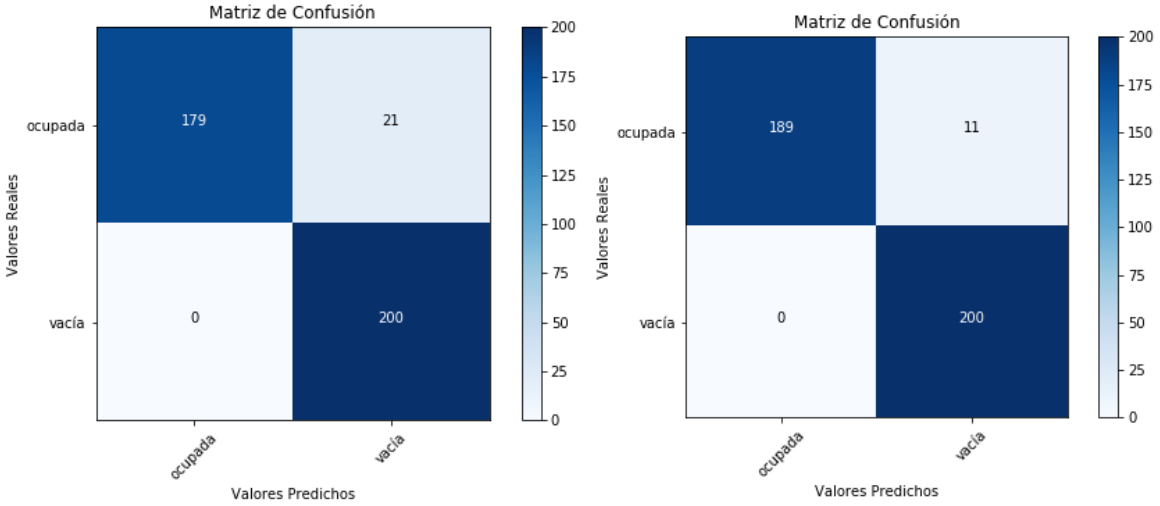


Figura 4.27: Matrices de confusión sin normalizar para el modelo seleccionado. La matriz de la izquierda se consigue con un valor de corte de 0.2 y la derecha de 0.5.

A continuación realizamos un estudio visual para un pequeño conjunto de *frames* del vídeo de validación. Estos serán los que nos sirvan para evaluar de una forma más realista la calidad del modelo.

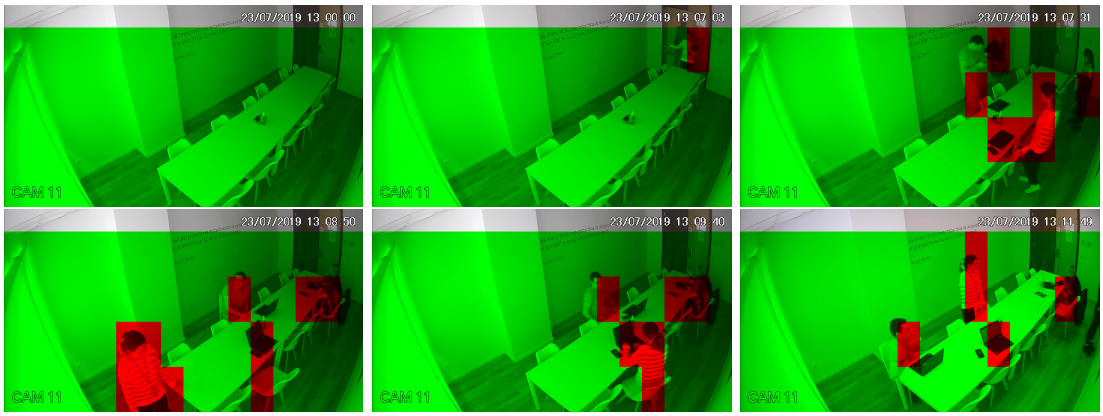


Figura 4.28: Estudio visual de un pequeño conjunto de *frames* del vídeo de validación.

La clasificación del modelo elegido es muy buena. El *frame* de la sala vacía está correctamente clasificado y, en el caso de los *frames* que corresponden con la sala ocupada, las predicciones de dichas zonas ocupadas son bastante exactas incluso para momentos como la entrada a la sala que suele conducir a error.

Gracias a la finalidad de construcción de nuestro modelo nos interesa la clasificación de la imagen completa del *frame*, por tanto evaluaremos el modelo desde este punto de vista. Por este motivo, calcularemos la matriz de confusión para la clasificación de las imágenes completas y no de sus fragmentos.

La construcción de esta nueva matriz se basa en la idea de clasificación que hemos llevado a lo largo del experimento. Recordamos que el proceso que sigue una imagen para ser clasificada es el siguiente:

1. La imagen es troceada eliminando la cabecera de esta.

2. Los fragmentos con clasificados por el modelo de red seleccionado.

3. Se realiza la clasificación de la imagen global. Si hay al menos un fragmento que el modelo haya predicho como ocupada, el estado de la sala será ocupado. En caso contrario la sala estará vacía.

Este último punto del proceso será el principal para crear la nueva matriz de confusión que aplicaremos al vídeo de validación.

Igualmente, nos aprovecharemos de la estructura del vídeo de validación. En este vídeo, la sala comienza vacía hasta el momento que un grupo de personas entran y se mantienen en ella hasta el final; lo que nos hace poder trasladar esta información a los *frames*. Podemos seleccionar un número de *frame* que nos sirva de referencia de tal manera que los *frames* que ocurran antes tengan la etiqueta vacía y a los *frames* de después les corresponda la ocupada. Debemos tener en cuenta que durante el proceso de extracción de los *frames* del vídeo se guardaron mediante una enumeración.

Por la cantidad de *frames* extraída por segundo de los vídeos se hace difícil esa elección de un único *frame*, por tanto cogemos una sucesión que sirvan de transición entre una y otra clase. Esta sucesión consta de 15 *frames* que corresponderían al intervalo de un segundo de vídeo.

Tras la clasificación de la imagen global y comparándola de este modo con su etiqueta real, podemos generar la matriz de confusión que buscábamos aplicada a un segundo conjunto de validación extraído de los *frames* del vídeo de validación. Esta matriz se ha creado con 1800 de esos *frames* generados aleatoriamente de entre todos los del vídeo, imponiendo que si hay algún *frames* que pertenezca a la sucesión de transición se clasifique como erróneo.

Esta imposición, aunque nos empeore los resultados de la clasificación del modelo, nos permitirá hacer un estudio distinguiendo los *frames* erróneos por clasificación de aquellos que muestran momentos problemáticos como entradas y salidas de la sala. Para mayor detalle de la construcción y del estudio de esta matriz de confusión se puede ver el cuaderno "Estudio del modelo seleccionado" del anexo C.

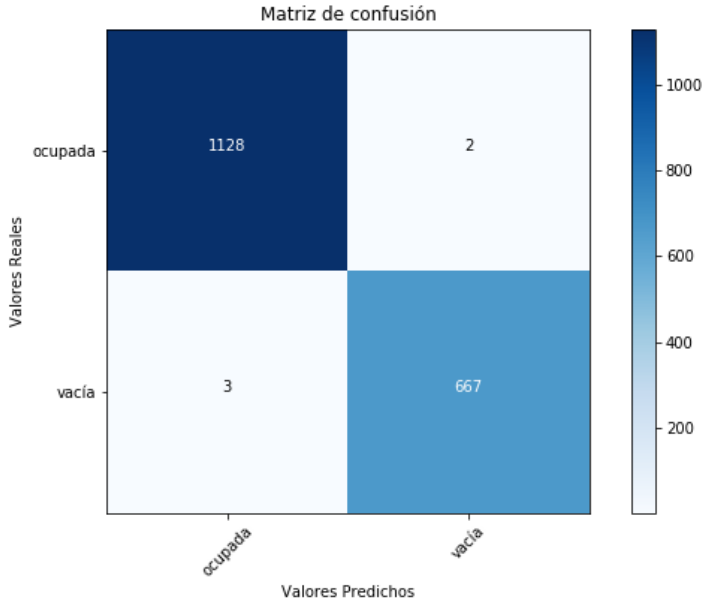


Figura 4.29: Matriz de confusión aplicado a un conjunto aleatorio de 1800 *frames* del vídeo de validación.

Podemos ver que los resultados de la evaluación del modelo en la matriz de confusión son muy buenos, solo se confunde en 5 de los 1800 *frames*. Para terminar de obtener información del rendimiento del modelo podemos analizar los conceptos definidos en la parte teórica extraídos de la matriz. Estos resultados son:

- Tasa de bien clasificados (*accuracy*), 99.72 %
- Tasa de mal clasificados, 0.28 %
- Sensibilidad, 99.55 %
- Especificidad, 99.82 %
- Precisión, 99.70 %

- Tasa de falsos positivos, 0.18 %
- Tasa de falsos negativos, 0.45 %

Son muy buenos resultados ya que, como dijimos en teoría, están próximos a los de un modelo óptimo. La *accuracy*, sensibilidad, especificidad y precisión están muy próximas al 100 % y el resto al 0 %. Destacamos también que la tasa de falsos negativos es más alta que la de falsos positivos como cabría esperar porque, como vimos a lo largo del experimento, las imágenes con cambios en las posiciones de las sillas podía producir errores en la predicción de la clasificación.

Gracias a la forma del código podemos hacer un estudio de los 5 *frames* mal clasificados. Estos son:

nº *frame*	clase de clasificación errónea
5251	ocupada
5382	ocupada
6325	vacía
6337	vacía
6341	vacía

Cuadro 4.6: Cuadro de los 5 *frames* mal clasificados durante la prueba de la matriz de confusión del estudio de validación del modelo global.

Procedemos a hacer un estudio visual de estos elementos para ver el porqué de la mala clasificación. De esta manera podremos saber cuántos de estos *frames* se han clasificado mal por la red y cuántos lo han hecho por formar parte de esos elementos de transición entre una clase y otra en el vídeo.

Comenzamos con los *frames* clasificados erróneamente como “ocupada”.

(a) *frame* 5251 (b) *frame* 5381

Figura 4.30: Imágenes mal clasificadas como ocupadas

El intervalo de tiempo transcurrido entre ambas imágenes es de 8 segundos. Del mismo modo, si nos fijamos en la figura 4.28, contemplamos como se clasificó un *frame* de un entorno de tiempo similar al de estos por la red de manera correcta. Este efecto puede ser consecuencia de los pequeños cambios de luz de momentos puntuales que hacen que varíe la clasificación.

En el caso de los *frames* mal clasificados como vacíos podemos repetir este mismo estudio.

(a) *frame* 6325

(b) *frame* 6337

(c) *frame* 6341

Figura 4.31: Imágenes mal clasificadas como vacías

Tan solo uno de estos *frames*, el número 6325 de todos los erróneamente clasificados, pertenece a la sucesión de transición. A este *frame* se le asignó una mala clasificación por la imposición que hicimos al construir la matriz de confusión. No obstante, podemos observar como la puerta no está del todo cerrada, efecto que requiere de detenimiento. Es por esto, que se tomaron este tipo de *frames* como elementos de transición. Podríamos pensar, en esta situación, que hubiese clasificado bien la sala ya que no hay nadie dentro y por tanto está vacía.

Las dos siguientes clasificaciones erróneas ya no pertenecen a los elementos de transición pero son situaciones difíciles de clasificar para la red, los momentos de entrada y salida. Hasta que la persona no está totalmente dentro de la sala, la red no es capaz de diferenciar que la sala está ocupada. Para comprobar este efecto podemos comparar estas dos imágenes con la imagen central superior de la figura 4.28, donde la red sí que es capaz de clasificar bien el fragmento correspondiente a entrar en la sala. Para solucionar este efecto bastaría con tener más elementos de entrada y salida en los vídeos de las salas y entrenar con ellos la red del modelo.

En resumen, estas clasificaciones erróneas obtenidas son bastante comprensibles ya que, los errores se pueden atribuir en gran medida a la sucesión de transición que hemos fijado para decir que esos fragmentos se consideran como sala ocupada.

4.4. Conclusiones

Para concluir la fase de viabilidad, que corresponde a este TFM, podemos afirmar tras los experimentos realizados que, es posible llevar a cabo el proyecto bajo la aplicación de las técnicas de las CNNs expuestas en las notas teóricas.

Esta afirmación se deriva de que los principales problemas encontrados han sido las pocas configuraciones distintas de las sillas en los vídeos de la sala donde se realizaron las pruebas y los momentos de entrada y salida. Ambos problemas remediables si extraemos de la actual fuente de datos, es decir, del sistema de seguridad que dispone la empresa en las salas, una

mayor muestra para reforzar el entrenamiento en estos aspectos.

Además el modelo finalmente seleccionado, ha obtenido resultados muy buenos con los conjuntos de datos con lo que contábamos. Para hacernos una idea, según estos resultados, si se quisiese poner un piloto en la puerta de la sala el cual se iluminase de verde si la sala está vacía o de rojo si la sala está ocupada automáticamente, apenas se equivocaría en 20 segundos al seleccionar el color de la luz para un intervalo de tiempo de aproximadamente 2 horas. Durante estos segundos (no necesariamente seguidos) se podría ver el error en el cambio de color, ya que se puede ser consciente si alguien está entrando o no en la sala. De esta manera se lograría un gran grado de fiabilidad del estado de la sala.

Finalmente, este estudio no solo sirve para gestionar las salas sino que, también, se pueden extraer datos con los que poder hacer un estudio estadístico sobre el buen o mal uso de las salas en la empresa.

Como resultado de esta prueba de viabilidad, se ha pasado a planificar la puesta en producción del modelo dentro de Efor.

Anexos

Anexo A

Funciones definidas

Los códigos mostrados a continuación son las funciones necesarias que se han programado para que resultase más fácil la programación del proyecto omitiendo de esta manera código repetitivo. Estas funciones están programadas en ficheros de *Python*, que serán cargados en los cuadernos de *Python* del proyecto que se mostrarán en los siguientes anexos. De esta manera podremos hacer uso de estas funciones a lo largo del código programado.

Las funciones son:

1. Función mallado.
2. Función mallado numerado.
3. Función corte de imágenes.
4. Función de cambio de color.
5. Función de lista a imagen.
6. Función de guardado.
7. Función cálculo de la matriz de confusión.

Notar que para las funciones mallado y mallado numerado se ha utilizado la estructura de una respuesta encontrada en [19] y la función para dibujar la matriz de confusión se inspira en el código de la documentación que podemos encontrar en [21].

Función mallado

mallado.py

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:

get_ipython().run_line_magic('pylab', '')
from PIL import Image
import matplotlib
import matplotlib.pyplot as plt
import pickle
import matplotlib.ticker as plticker

#**************Función mallado**************
#
#Función que superpone a la imagen introducida un mallado.
#
#Argumentos:
# img ---> imagen.
# myIntervalx ---> dimensión del rectángulo del mallado (Inicializado por defecto a 100)
    en el eje x.
# myIntervaly ---> dimensión del rectángulo del mallado (Inicializado por defecto a 100)
    en el eje y.
# my_dpi ---> resolución de la figura (Inicializado por defecto a 100).
#
#Devuelve una figura con el mallado pintado sobre la imagen.


def mallado(img,myIntervalx=100.,myIntervaly=100.,my_dpi=100.):
    fig=plt.figure(figsize=(float(img.size[0])/my_dpi,float(img.size[1])/my_dpi),dpi=
    my_dpi)
    ax=fig.add_subplot(111)
    fig.subplots_adjust(left=0,right=1,bottom=0,top=1)
    loc = plticker.MultipleLocator(base=myIntervalx)
    loc1 = plticker.MultipleLocator(base=myIntervaly)
    ax.xaxis.set_major_locator(loc)
    ax.yaxis.set_major_locator(loc1)
    ax.grid(which='both',b=bool, axis='both', linestyle='-',linewidth=2)
    ax.imshow(img)
    return fig
```

Función mallado numerado

mallado_num.py

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:

get_ipython().run_line_magic('pylab', '')
from PIL import Image
import matplotlib
import matplotlib.pyplot as plt
import pickle
import matplotlib.ticker as plticker

#**************Función mallado_num**************
#
#Función que superpone a la imagen introducida un mallado que además enumera.
#
#Argumentos:
# img ---> imagen.
# myIntervalx ---> dimensión del rectángulo del mallado (Inicializado por defecto a 100)
    en el eje x.
# myIntervaly ---> dimensión del rectángulo del mallado (Inicializado por defecto a 100)
    en el eje y.
# my_dpi ---> resolución de la figura (Inicializado por defecto a 100).
#
#Devuelve una figura con el mallado pintado sobre la imagen y enumerado.


def mallado_num (img,myIntervalx=100.,myIntervaly=100.,my_dpi=100.):
    fig=plt.figure(figsize=(float(img.size[0])/my_dpi,float(img.size[1])/my_dpi),dpi=
    my_dpi)
    ax=fig.add_subplot(111)
    fig.subplots_adjust(left=0,right=1,bottom=0,top=1)
    loc = plticker.MultipleLocator(base=myIntervalx)
    loc1 = plticker.MultipleLocator(base=myIntervaly)
    ax.xaxis.set_major_locator(loc)
    ax.yaxis.set_major_locator(loc1)
    ax.grid(which='both',b=bool, axis='both', linestyle='-',linewidth=2)
    ax.imshow(img)
    nx=abs(int(float(ax.get_xlim()[1]-ax.get_xlim()[0])/float(myIntervalx)))
    ny=abs(int(float(ax.get_ylim()[1]-ax.get_ylim()[0])/float(myIntervaly)))
    for j in range(ny):
        y=myIntervaly/2+j*myIntervaly
        for i in range(nx):
            x=myIntervalx/2.+float(i)*myIntervalx
            ax.text(x,y,'{:d}'.format(i+j*nx),color='g',ha='center',va='center')
    return fig
```

Función corte de imágenes

gridcrop.py

```
1
2
3  #!/usr/bin/env python
4  # coding: utf-8
5
6  # In[2]:
7
8  get_ipython().run_line_magic('pylab', '')
9  import numpy as np
10 from PIL import Image
11 import matplotlib
12 import matplotlib.pyplot as plt
13 import pickle
14 import matplotlib.ticker as plticker
15
16 #**************Función gridcrop**************
17 #
18 #Función que corta la imagen según el mallado introducido.
19 #
20 #Argumentos:
21 # image ---> imagen.
22 # dimx ---> dimensión del rectángulo en el eje x.
23 # dimy ---> dimensión del rectángulo en el eje y.
24 #
25 #Devuelve una lista con todos los trozos de la imagen.
26
27
28 def gridcrop(image,dimx,dimy):
29     nx=int(image.size[0]/dimx)
30     ny=int(image.size[1]/dimy)
31     trocitos=[]
32     for j in range (0,ny):
33         for i in range (0,nx):
34             trocitos=trocitos+[image.crop((i*dimx,j*dimy,dimx+i*dimx,dimy+j*dimy))]
35     return trocitos
```

Función de cambio de color

colors.py

```
1
2  #!/usr/bin/env python
3  # coding: utf-8
4
5  # In[2]:
6
7  get_ipython().run_line_magic('pylab', '')
8  import numpy as np
9  from PIL import Image
10 import matplotlib
11 import matplotlib.pyplot as plt
12 import pickle
13 import matplotlib.ticker as plticker
14
15 #**************Funciones change_color**************
16 #
17 #Funciones que anulan dos de los tres canales de una imagen en formato rgb.
18 #
19 #Argumentos:
20 # change_list ---> índices de los elementos que queremos cambiar .
21 # S ---> lista con imágenes.
22 #
23 #Devuelve la misma lista con el color de las imágenes modificado.
24
25 def change_colorred(change_list,S):
26     for i in change_list:
27         foto=S[i].copy()
28         data=foto.getdata()
29         r = [(d[0], 0, 0) for d in data] #se cambian los colores salvo el verde
30         S[i].putdata(r)
31     return S
32
33 def change_colorgreen(change_list,S):
34     for i in change_list:
35         data=S[i].getdata()
36         r = [(0, d[0], 0) for d in data] #se cambian los colores salvo el verde
37         S[i].putdata(r)
38     return S
39
40 def  change_colorblue (change_list,S):
41     for i in change_list:
42         foto=S[i].copy()
43         data=foto.getdata()
44         r = [(0, 0, d[0]) for d in data] #se cambian los colores salvo el verde
45         S[i].putdata(r)
46     return S
```

Función de lista a imagen

list2img.py

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:

import cv2
get_ipython().run_line_magic('pylab', '')
import numpy as np
from PIL import Image
import matplotlib
import matplotlib.pyplot as plt
import pickle
import matplotlib.ticker as plticker

#**************Función list2img**************
#
#Función que reconstruye una imagen desde un array donde están sus fragmentos.
#
#Argumentos:
# S ---> lista con imágenes.
# dim_x ---> número de fragmentos horizontales.
# dim_y ---> número de fragmentos verticales.
#
#Devuelve la imagen reconstruida.

def list2img(S,dim_x,dim_y):
    foo=S[0]
    for i in range (1,dim_x):
        foo=np.concatenate((foo,S[i]),axis=1)
    bar=foo
    for i in range (1,dim_y):
        foo=S[dim_x*i]
        for j in range (1,dim_x):
            foo=np.concatenate((foo,S[dim_x*i+j]),axis=1)
        bar=np.concatenate((bar,foo),axis=0)
    b,g,r = cv2.split(bar)
    rgb_img = cv2.merge([r,g,b])
    return rgb_img
```

Función de guardado

Save.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:

import cv2
get_ipython().run_line_magic('pylab', '')
import numpy as np
from PIL import Image

#**************Función guardar**************
#
#Función que guarda las imágenes de un array independientemente con un nombre predefinido.
#
#Argumentos:
# lis ---> lista de imágenes que queremos guardar.
# name---> nombre con el que quremos guardar las imágenes.


def guardar(lis,name):
    for i in range (0,len(lis)):
        m=lis[i]
        m.save(name+str(i)+'.jpg')
```

Función cálculo de la matriz de confusión

PlotConfusionMatrix_num.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import itertools
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import numpy as np

#**************Función que dibuja la matriz de confusión**************
#
#Función que calcula y dibuja la matriz de confusión de forma visual.
#
#Argumentos:
# y_real ---> etiquetas reales del conjunto de datos.
# y_pred---> etiquetas predichas por el modelo del conjunto de datos.
# Classes ---> nombre de las clases del modelo.
# Normalize --> Booleano que nos indica si se normaliza o no la matriz de confusión. Por defecto no se realiza la normalización.
# Title ---> Título de la figura resultante. Por defecto será 'Matriz de Confusión'.
# cmap ---> Definición de los colores de la matriz de confusión. Por defecto será en escala de azules.


def plot_confusion_matrix(y_real,y_pred, Classes,Normalize=False, Title='Matriz de Confusión',cmap=plt.cm.Blues):
    Confusion_Matrix=confusion_matrix(y_real, y_pred)
    plt.imshow(Confusion_Matrix,interpolation='nearest',cmap=cmap)
    plt.title(Title)
    plt.colorbar()
    tick_marks=np.arange(len(Classes))
    plt.xticks(tick_marks,Classes,rotation=45)
    plt.yticks(tick_marks, Classes)
    if Normalize:
        Confusion_Matrix=Confusion_Matrix.astype('float')/Confusion_Matrix.sum()#(axis=1)[:,np.newaxis]
        print('Matriz de confusion normalizada')
    else:
        print('Matriz de confusion sin normalizar')
    thresh=Confusion_Matrix.max()/2.
    for i,j in itertools.product(range(Confusion_Matrix.shape[0]),range(Confusion_Matrix.shape[1])):
        plt.text(j,i,Confusion_Matrix[i,j],horizontalalignment='center',
                 color='white' if Confusion_Matrix[i,j] > thresh else 'black')
        plt.tight_layout()
        plt.ylabel('Valores Reales')
        plt.xlabel('Valores Predichos')
```

Anexo B

Pretratamiento y estudio de los datos

En este anexo se presentan los cuadernos de *Jupyter* con el código empleado en el pretratamiento de los datos cedidos por Efor y en el posterior estudio de estos. Se adjuntan los siguientes cuadernos:

- **Obtención de *frames***. Se extraen las imágenes de los vídeos que se utilizarán para el diseño y estudio del modelo.
- **Estudio de la sala**. Se realiza el estudio de la sala sobre como fragmentar de forma óptima la imagen sin pérdida de información.
- **Preparación de los conjuntos de entrenamiento y validación**. Se generan de manera aleatoria los *frames* con los que entrenaremos y haremos una primera validación del modelo. También se fragmentan las imágenes y se guardan en carpetas para una inspección y selección manual posterior para formar los conjuntos.

Obtención de frames

```
In [1]: import cv2
        %pylab
        import numpy as np
        from PIL import Image
        import matplotlib
        import matplotlib.pyplot as plt
        import pickle
        import matplotlib.ticker as plticker
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

Proceso de lectura de los frames del vídeo y guardado de las imágenes de la sala ocupada.

```
In [2]: %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
```

```
In [3]: vidcap = cv2.VideoCapture('videosalaocupada.asf')
        success,image = vidcap.read()
        count = 0
        success = True
        while success:
          cv2.imwrite("frame%d.jpg" % count, image)
          success,image = vidcap.read()
          count += 1
```

Proceso de lectura de los frames del vídeo y guardado de las imágenes de la sala vacía.

```
In [4]: %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala vacía
```

```
In [5]: vidcap = cv2.VideoCapture('videosalavacia.asf')
        success,image = vidcap.read()
        count = 0
        success = True
        while success:
          cv2.imwrite("frame%d.jpg" % count, image)
          success,image = vidcap.read()
          count += 1
```

Proceso de lectura de los frames del vídeo y guardado de las imágenes del vídeo de validación.

```
In [6]: %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala validacion
```

```
In [7]: vidcap = cv2.VideoCapture('vídeo validación.asf')
        success,image = vidcap.read()
        count = 0
        success = True
        while success:
          cv2.imwrite("frame%d.jpg" % count, image)
          success,image = vidcap.read()
          count += 1
```

Estudio de la sala

```
In [1]: import cv2
        %pylab
        import numpy as np
        from PIL import Image
        import matplotlib
        import matplotlib.pyplot as plt
        import pickle
        import matplotlib.ticker as plticker
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: #CARGA DE LAS FUNCIONES DEFINIDAS
        %cd C:\Users\ialdea\TFM
        %run colors.py
        %run gridcrop.py
        %run list2img.py
        %run mallado.py
        %run mallado_num.py
```

Mallado cuadrado

```
In [3]: %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
```

```
In [4]: #Definición de las dimensiones del mallado
        rectangle_x=120
        rectangle_y=120
```

```
In [5]: frame0=Image.open('frame0.jpg')
        dim_x=int(frame0.size[0]/rectangle_x)
        dim_y=int(frame0.size[1]/rectangle_y)
```

```
In [6]: mallado(frame0,rectangle_x,rectangle_y)
```

Out[6]:

In [7]:
```
frame1=Image.open('frame3721.jpg')
mallado(frame1,120,120)
```

Out[7]:

Estudio de mallados más pequeños.

In [8]:
```
frame2=Image.open('frame9069.jpg')
mallado(frame2,65,65)
```

Out[8]:

In [9]:
```
mallado_num(frame2,65,65)
```

Out[9]:

Los siguientes fragmentos de la imagen anterior pueden conducir a error en las redes.

```
In [10]: P=gridcrop(frame2,65,65)
         lis_error=[300,258,129,398]
         foo=P[136]
         for i in lis_error:
             foo=np.concatenate((foo,P[i]),axis=1)
             b,g,r = cv2.split(foo)
             rgb_img = cv2.merge([r,g,b])
         cv2.imshow('fragmentos problemáticos',rgb_img)
```

Mallado Rectangular

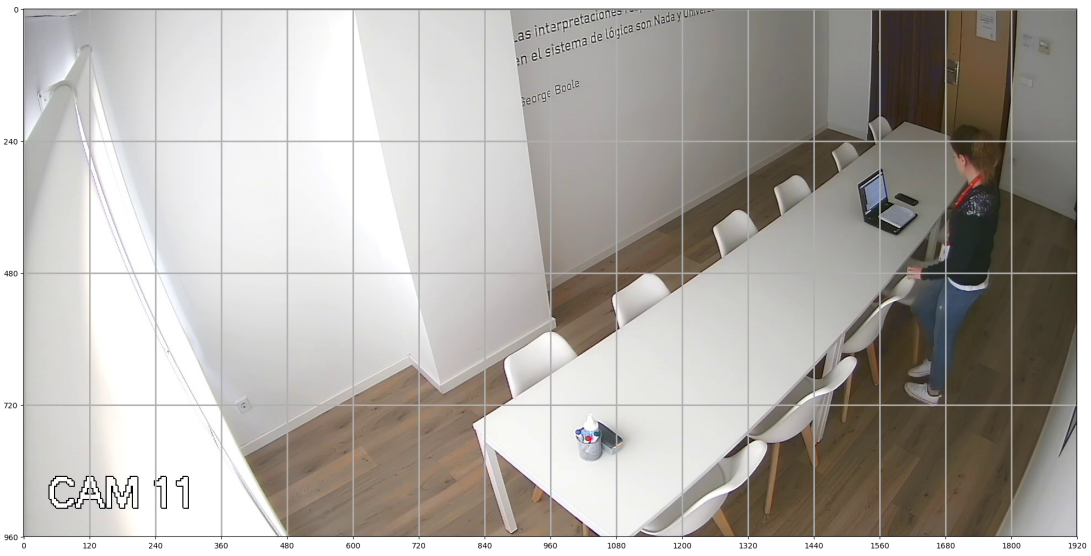
```
In [11]: mallado(frame1,120,240)
```

Out[11]:

Estructura del mallado rectangular finalmente utilizado eliminando la cabecera de las fotos para no inducir a error.

```
In [12]: %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
         frame=Image.open('frame3721.jpg')
         foo=gridcrop(frame,120,120)
         bar=list2img(foo[16:144],dim_x,dim_y-1)
         %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
         cv2.imwrite('img.jpg',bar)
         img=Image.open('img.jpg')
         mallado(img,120,240)
```

Out[12]:



In [13]: close('all')

Preparación de los conjuntos de entrenamiento y validación

```
In [1]: import cv2
        %pylab
        import numpy as np
        from PIL import Image
        import matplotlib
        import matplotlib.pyplot as plt
        import pickle
        import matplotlib.ticker as plticker
        import random
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: #CARGA DE LAS FUNCIONES DEFINIDAS
        %cd C:\Users\ialdea\TFM
        %run gridcrop.py
        %run Save.py
        %run list2img.py
```

Selección de los frames para el entrenamiento y la validación

```
In [3]: random.seed(0)
        n=1200
        #Frames del vídeo de la sala ocupada donde está la sala ocupada
        bussy = [random.randint(3526,24525) for _ in range(n)]
        m=300
        #Frames del vídeo de la sala ocupada donde está la sala vacía
        a = [random.randint(0,3526) for _ in range(m)]
        p=300
        #Frames del vídeo de la sala ocupada donde está la sala vacía
        b= [random.randint(24526,27006) for _ in range(p)]
        q=600
        #Frames del vídeo de la sala vacía
        c=[random.randint(0,26994) for _ in range(q)]
```

```
In [4]: lis_bussy_train=bussy[0:1000]
        lis_bussy_val=bussy[1000:len(bussy)]
        lis_empty_train0=a[0:250]+b[0:250]
        lis_empty_train1=c[0:500]
```

```
        lis_empty_val0=a[250:len(a)]+b[250:len(b)]
        lis_empty_val1=c[500:len(c)]
```

Guardado de los frames seleccionados y fragmentados en carpetas

Definición de las dimensiones para el corte de las imágenes

```
In [5]: #Tamaño de la primera fila que vamos a quitar
        quitx=120
        quity=120
        #Tamaño del mallado con el que cortamos las imágenes
        rectangle_x=120
        rectangle_y=240
        #Dimensiones de los frames
        dim_img_x=1920
        dim_img_y=1080
        #Número de trozos de imágenes resultantes
        dim_x=int(dim_img_x/quitx)
        dim_y=int(dim_img_y/quity)
```

Frames del conjunto de entrenamiento con la sala ocupada.

```
In [6]: for i in lis_bussy_train:
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
            frame=Image.open('frame'+str(i)+'.jpg')
            foo=gridcrop(frame,120,120)
            bar=list2img(foo[16:144],dim_x,dim_y-1)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
            cv2.imwrite('img.jpg',bar)
            img=Image.open('img.jpg')
            foo=gridcrop(img,120,240)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\selección de ocupada
            guardar(foo,'nb'+str(i)+'_')
```

Frames del conjunto de entrenamiento con la sala vacía.

```
In [7]: for i in lis_empty_train0:
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
            frame=Image.open('frame'+str(i)+'.jpg')
            foo=gridcrop(frame,120,120)
            bar=list2img(foo[16:144],dim_x,dim_y-1)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
            cv2.imwrite('img.jpg',bar)
            img=Image.open('img.jpg')
            foo=gridcrop(img,120,240)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\selección de vacía
            guardar(foo,'nb'+str(i)+'_')
        for i in lis_empty_train1:
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala vacía
```

```
            frame=Image.open('frame'+str(i)+'.jpg')
            foo=gridcrop(frame,120,120)
            bar=list2img(foo[16:144],dim_x,dim_y-1)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
            cv2.imwrite('img.jpg',bar)
            img=Image.open('img.jpg')
            foo=gridcrop(img,120,240)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\selección de vacía
            guardar(foo,'ne'+str(i)+'_')
```

Frames del conjunto de validación con la sala ocupada.

```
In [8]: for i in lis_bussy_val:
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
            frame=Image.open('frame'+str(i)+'.jpg')
            foo=gridcrop(frame,120,120)
            bar=list2img(foo[16:144],dim_x,dim_y-1)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
            cv2.imwrite('img.jpg',bar)
            img=Image.open('img.jpg')
            foo=gridcrop(img,120,240)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\selección de ocupada val
            guardar(foo,'nb'+str(i)+'_')
```

Frames del conjunto de validación con la sala vacía.

```
In [9]: for i in lis_empty_val0:
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
            frame=Image.open('frame'+str(i)+'.jpg')
            foo=gridcrop(frame,120,120)
            bar=list2img(foo[16:144],dim_x,dim_y-1)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
            cv2.imwrite('img.jpg',bar)
            img=Image.open('img.jpg')
            foo=gridcrop(img,120,240)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\selección de vacía val
            guardar(foo,'nb'+str(i)+'_')
        for i in lis_empty_val1:
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala vacía
            frame=Image.open('frame'+str(i)+'.jpg')
            foo=gridcrop(frame,120,120)
            bar=list2img(foo[16:144],dim_x,dim_y-1)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
            cv2.imwrite('img.jpg',bar)
            img=Image.open('img.jpg')
            foo=gridcrop(img,120,240)
            %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\selección de vacía val
            guardar(foo,'ne'+str(i)+'_')
```

Anexo C

Diseño, resultados y ejecución de los modelos

En este anexo queda recogido el código base usado para diseñar, usar y validar los modelos de CNN empleados en el proyecto del trabajo. Se adjuntan los siguientes cuadernos:

- **Diseño, entrenamiento y primeros resultados de la CNN**. Cuaderno principal del experimento donde se programan los modelos. Se muestra el código utilizado tomando de ejemplo el modelo de red finalmente seleccionado. Aparecen las definiciones del modelo, la lectura de los conjuntos, el entrenamiento de la red y una pequeña validación inicial.
- **Carga del modelo y clasificación**. Se muestra la forma en la que se cargan los modelos guardados y se usan para predecir las imágenes junto con un previo tratamiento de estas para que tengan la misma forma de entrada a la red que hemos modelado. Al final del cuaderno se muestra el código necesario para realizar el estudio visual aunque lo que verdaderamente nos interesa en un futuro es el código que nos dice si la sala está ocupada o vacía, ya que la visualización de la imagen nos sirve principalmente para el ver los fallos del modelo.
- **Análisis de la curva ROC**. Código necesario para el cálculo de la curva ROC sobre nuevas predicciones.
- **Estudio del modelo seleccionado**. Se amplia el estudio de validación con el cálculo de los conceptos definidos a partir de la matriz de confusión del estudio de la clasificación de la imagen global en vez de en los fragmentos.

Las líneas de código de estos cuadernos han sido escritas con la ayuda de la documentación de Keras y los ejemplos de esta que podemos encontrar en [13].

Diseño, entrenamiento y estudio de la CNN

```
In [1]: from scipy import*
        %pylab
        from numpy.fft import*
        from PIL import Image
        import numpy as np
        import cv2
        import matplotlib.pyplot as plt
        import matplotlib.ticker as plticker
        from sklearn.metrics import confusion_matrix

Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: import tensorflow as tf
        import tensorflow
        from tensorflow import keras
        from keras.preprocessing.image import ImageDataGenerator
        from keras.models import Sequential
        from keras.layers import Convolution2D, MaxPooling2D
        from keras.layers import Activation, Dropout, Flatten, Dense
        from keras.optimizers import SGD
        from keras import initializers
        from keras.utils import to_categorical
        from keras import models
        import numpy as np
        import matplotlib.pyplot as plt
        import pickle
        from keras.models import model_from_json

Using TensorFlow backend.
```

```
In [3]: #CARGAMOS LAS FUNCIONES DEFINIDAS
        %cd C:\Users\ialdea\TFM
        %run gridcrop.py
        %run Save.py
        %run list2img.py
        %run PlotConfusionMatrix.py
```

Definición de los conjuntos

```
In [4]: train_datagen = ImageDataGenerator(rescale=1./255)
        test_datagen = ImageDataGenerator(rescale=1./255)

In [5]: #Tamaño de las imágenes de entrada a la red
        img_width=120
        img_height=240

In [6]: #Conjunto de entrenamiento
        train_generator = train_datagen.flow_from_directory(
                "C://Users//ialdea//Documents//TFM//Caso Practico//data//train definitivo",
                target_size=(img_width, img_height),
                batch_size=20,
                class_mode='binary')

Found 2000 images belonging to 2 classes.


In [7]: #Conjunto test
        validation_generator = test_datagen.flow_from_directory(
             "C://Users//ialdea//Documents//TFM//Caso Practico//data//validation definitivo",
                target_size=(img_width, img_height),
                batch_size=20,
                shuffle=False,
                class_mode='binary')

Found 400 images belonging to 2 classes.


In [8]: #Codificación de etiquetas
        train_generator.class_indices

Out[8]: {'ocupada': 0, 'vacía': 1}
```

Información del modelo

```
In [9]: #******INFORMACIÓN DEL MODELO********
        #
        # número de imágenes que se consideran para el entrenamiento
        train_samples = 2000
        # número de imágenes se utilizan en la validación
        validation_samples = 400
        # número de veces que se ejecutará la red
        # sobre el conjunto de entrenamiento antes
        # de empezar con la validación
        epoch = 30
```

Diseño del modelo

In [10]:
```
# ** DISEÑO DE LA ARQUITECTURA **
#
from numpy.random import seed
seed(2)
from tensorflow import set_random_seed
set_random_seed(2)
init=initializers.glorot_uniform(seed=1)
model = Sequential()

#CAPAS DE CONVOLUCIÓN
model.add(Convolution2D(6, (3, 3), padding='valid', use_bias=True,
  kernel_initializer=init, bias_initializer='zeros',
  input_shape = (img_width, img_height,3), data_format="channels_last"))
model.add(Activation('elu'))

model.add(Convolution2D(6, (3, 3),padding='valid', use_bias=True,
  kernel_initializer=init,bias_initializer='zeros',
  input_shape = (img_width, img_height,3), data_format="channels_last"))
model.add(Activation('elu'))

#FLATTENING
model.add(Flatten())

#FULL CONNECTION
model.add(Dense(512,activation='sigmoid'))
model.add(Dense(256,activation='sigmoid'))
model.add(Dense(1,activation='sigmoid'))

# ** FIN DEL MODELO **
```

In [11]:
```
#*****CARACTERÍSTICAS DEL APRENDIZAJE***********
#
#Antes de compilar el modelo hay que configurar el proceso de aprendizaje,
#la función de perdida, el optimizador y medida para la cnn
model.compile(loss='binary_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
#Modelo resultante
print(model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 118, 238, 6)       168
_________________________________________________________________
activation_1 (Activation)    (None, 118, 238, 6)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 116, 236, 6)       330
_________________________________________________________________
activation_2 (Activation)    (None, 116, 236, 6)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 164256)            0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               84099584
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 257
=================================================================
Total params: 84,231,667
Trainable params: 84,231,667
Non-trainable params: 0
_________________________________________________________________
None
```

Entrenamiento del modelo

```
In [12]: #****ENTRENAMIENTO DEL MODELO*******
         from time import time
         start_time=time()
         train_model=model.fit_generator(
                 train_generator,
                 samples_per_epoch=train_samples,
                 nb_epoch=epoch,
                 validation_data=validation_generator,
                 nb_val_samples=validation_samples,
                 #steps_per_epoch=32/2,
                 verbose=1)
         end_time=time()-start_time
         hours=int(end_time/3600)
         minutes=(end_time-hours*3600)/60
         seconds=end_time%60
         print('El tiempo de entrenamiento es:',hours, 'horas ',int(minutes),
               'minutos y' ,seconds,' segundos')
```

```
Epoch 1/30
100/100 [==============================] - 536s 5s/step -
        loss: 0.6584 - acc: 0.6265 - val_loss: 0.6294 - val_acc: 0.6850
Epoch 2/30
100/100 [==============================] - 350s 3s/step -
        loss: 0.6224 - acc: 0.6785 - val_loss: 0.5995 - val_acc: 0.7025
Epoch 3/30
100/100 [==============================] - 299s 3s/step -
        loss: 0.5964 - acc: 0.6950 - val_loss: 0.5720 - val_acc: 0.7250
Epoch 4/30
100/100 [==============================] - 296s 3s/step -
        loss: 0.5695 - acc: 0.7165 - val_loss: 0.5631 - val_acc: 0.7000
Epoch 5/30
100/100 [==============================] - 305s 3s/step -
        loss: 0.5501 - acc: 0.7280 - val_loss: 0.5468 - val_acc: 0.7300
Epoch 6/30
100/100 [==============================] - 308s 3s/step -
        loss: 0.5275 - acc: 0.7425 - val_loss: 0.5328 - val_acc: 0.7375
Epoch 7/30
100/100 [==============================] - 306s 3s/step -
        loss: 0.5154 - acc: 0.7470 - val_loss: 0.5234 - val_acc: 0.7250
Epoch 8/30
100/100 [==============================] - 312s 3s/step -
        loss: 0.4953 - acc: 0.7510 - val_loss: 0.5003 - val_acc: 0.7450
Epoch 9/30
100/100 [==============================] - 297s 3s/step -
        loss: 0.4717 - acc: 0.7755 - val_loss: 0.4747 - val_acc: 0.7825
Epoch 10/30
100/100 [==============================] - 321s 3s/step -
        loss: 0.4374 - acc: 0.8095 - val_loss: 0.4641 - val_acc: 0.7825
Epoch 11/30
100/100 [==============================] - 353s 4s/step -
        loss: 0.4229 - acc: 0.8070 - val_loss: 0.3929 - val_acc: 0.8425
Epoch 12/30
100/100 [==============================] - 318s 3s/step -
        loss: 0.3791 - acc: 0.8370 - val_loss: 0.4204 - val_acc: 0.7725
Epoch 13/30
100/100 [==============================] - 319s 3s/step -
        loss: 0.3625 - acc: 0.8490 - val_loss: 0.3711 - val_acc: 0.8375
Epoch 14/30
100/100 [==============================] - 299s 3s/step -
        loss: 0.3132 - acc: 0.8745 - val_loss: 0.3288 - val_acc: 0.8525
Epoch 15/30
100/100 [==============================] - 331s 3s/step -
        loss: 0.3082 - acc: 0.8705 - val_loss: 0.2817 - val_acc: 0.9025
Epoch 16/30
100/100 [==============================] - 335s 3s/step -
        loss: 0.2595 - acc: 0.9040 - val_loss: 0.2486 - val_acc: 0.9275
```

```
Epoch 17/30
100/100 [==============================] - 318s 3s/step -
        loss: 0.2273 - acc: 0.9190 - val_loss: 0.3585 - val_acc: 0.8325
Epoch 18/30
100/100 [==============================] - 321s 3s/step -
        loss: 0.2181 - acc: 0.9245 - val_loss: 0.2037 - val_acc: 0.9425
Epoch 19/30
100/100 [==============================] - 327s 3s/step -
        loss: 0.2016 - acc: 0.9305 - val_loss: 0.1867 - val_acc: 0.9475
Epoch 20/30
100/100 [==============================] - 328s 3s/step -
        loss: 0.1686 - acc: 0.9425 - val_loss: 0.1650 - val_acc: 0.9625
Epoch 21/30
100/100 [==============================] - 317s 3s/step -
        loss: 0.1584 - acc: 0.9495 - val_loss: 0.1912 - val_acc: 0.9275
Epoch 22/30
100/100 [==============================] - 345s 3s/step -
        loss: 0.1298 - acc: 0.9635 - val_loss: 0.1549 - val_acc: 0.9575
Epoch 23/30
100/100 [==============================] - 334s 3s/step -
        loss: 0.1223 - acc: 0.9625 - val_loss: 0.1414 - val_acc: 0.9550
Epoch 24/30
100/100 [==============================] - 310s 3s/step -
        loss: 0.1147 - acc: 0.9670 - val_loss: 0.1197 - val_acc: 0.9725
Epoch 25/30
100/100 [==============================] - 301s 3s/step -
        loss: 0.1137 - acc: 0.9690 - val_loss: 0.1479 - val_acc: 0.9575
Epoch 26/30
100/100 [==============================] - 307s 3s/step -
        loss: 0.0815 - acc: 0.9805 - val_loss: 0.1505 - val_acc: 0.9500
Epoch 27/30
100/100 [==============================] - 329s 3s/step -
         loss: 0.0685 - acc: 0.9845 - val_loss: 0.1427 - val_acc: 0.9525
Epoch 28/30
100/100 [==============================] - 326s 3s/step -
        loss: 0.0774 - acc: 0.9800 - val_loss: 0.1004 - val_acc: 0.9700
Epoch 29/30
100/100 [==============================] - 336s 3s/step -
        loss: 0.0745 - acc: 0.9770 - val_loss: 0.0885 - val_acc: 0.9775
Epoch 30/30
100/100 [==============================] - 337s 3s/step -
        loss: 0.0516 - acc: 0.9885 - val_loss: 0.0898 - val_acc: 0.9725


El tiempo de entrenamiento es: 2 horas  43 minutos y 40.3520245552063  segundos
```

Análisis de los resultados del entrenamiento

Estudio gráfico de la función de pérdida y de la tasa de bien clasificados a lo largo de las épocas:

```
In [13]: plt.subplot(311)
         plt.plot(train_model.epoch,train_model.history['loss'],marker='o',linestyle='--',
                  color='b',label='Conjunto entrenamiento')
         plt.plot(train_model.epoch,train_model.history['val_loss'],marker='o',linestyle='--',
                  color='g',label='Conjunto de validación')
         plt.xlabel('Épocas')
         plt.ylabel('Función de pérdida')
         plt.title('Comparación de la función de pérdida \n
          en el conjunto de entrenamiento y de validación')
         plt.legend(loc='best')
         plt.subplot(313)
         plt.plot(train_model.epoch,train_model.history['acc'],marker='o',linestyle='--',
                  color='b',label='Conjunto entrenamiento')
         plt.plot(train_model.epoch,train_model.history['val_acc'],marker='o',linestyle='--',
                  color='g',label='Conjunto de validación')
         plt.xlabel('Épocas')
         plt.ylabel('Accuracy')
         plt.title('Comparación de la tasa de bien clasificados \n
          en el conjunto de entrenamiento y de validación')
         plt.legend(loc='best')

Out[13]: <matplotlib.legend.Legend at 0x181964e6f60>
```

Cálculo de la **matriz de confusión** de la red:

```
In [14]: validation_generator = test_datagen.flow_from_directory(
             "C://Users//ialdea//Documents//TFM//Caso Practico//data//validation definitivo",
                 target_size=(img_width, img_height),
                 batch_size=1,
                 shuffle=False,
                 class_mode='binary')
         validation_generator.reset()
         nb_samples=len(validation_generator.filenames)
         predictions=model.predict_generator(validation_generator,steps=nb_samples,
             verbose=1,workers=1)

Found 400 images belonging to 2 classes.
400/400 [==============================] - 19s 48ms/step


In [15]: labels_predictions=[0] * len(predictions)
         for i in range (0,len(predictions)):
             if predictions[i]<0.5:
                 labels_predictions[i]=0
```

```
            else:
                labels_predictions[i]=1
```

```
In [16]: plot_confusion_matrix(validation_generator.classes, labels_predictions,
             ['ocupada','vacía'],Normalize=True)
```

```
Matriz de confusion normalizada
```

Guardado del modelo

```
In [17]: %cd C:\Users\ialdea\TFM

         #Guardado del modelo en JSON
         #
         model_json=model.to_json()
         with open ("CNNModel_11.json","w") as json_file:
             json_file.write(model_json)

         #Guardado de los pesos del modelo en HDF5
         #
         model.save_weights("CNNModel_11.h5")
```

```
C:\Users\ialdea\TFM
```

Carga del modelo y clasificación

```
In [1]: import tensorflow as tf
        import tensorflow

        from tensorflow import keras
        from keras.preprocessing.image import ImageDataGenerator
        from keras.models import Sequential
        from keras.layers import Convolution2D, MaxPooling2D
        from keras.layers import Activation, Dropout, Flatten, Dense
        from keras.optimizers import SGD
        from keras.utils import to_categorical
        from keras import models
        from keras.models import model_from_json

        import numpy as np
        import matplotlib
        import matplotlib.ticker as plticker
        import matplotlib.pyplot as plt
        import pickle
        from PIL import Image
        import cv2

        %matplotlib inline
        %pylab
```

```
Using TensorFlow backend.
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: #CARGA DE LAS FUNCIONES
        %cd C:\Users\ialdea\TFM
        %run colors.py
        %run gridcrop.py
        %run list2img.py
        %run Save.py
```

Carga el modelo

In [3]:
```
#CARGA DEL MODELO
#cargar json y crear el modelo
json_file=open('CNNModel_11.json','r')
loaded_model_json=json_file.read()
json_file.close()
first_model=model_from_json(loaded_model_json)
#cargar pesos del modelo guardado
first_model.load_weights("CNNModel_11.h5")
```

Definición de las características de aprendizaje

Se vuelve a definir las características de aprendizaje del modelo que se entrenó.

In [4]:
```
#*****CARACTERÍSTICAS DEL APRENDIZAJE***********
#
first_model.compile(loss='binary_crossentropy',
                    optimizer='sgd',
                    metrics=['accuracy'])
```

Definición y adaptación del frame a predecir

In [5]:
```
#NÚMERO DE FRAME A PREDECIR
 nframe=6337#12722#1358#6307#18494
 #DEFINICIÓN DE DIMESIONES (EN PÍXELES)
 quitx=120
 quity=120
 img_dimx=1920
 img_dimy=1080
 dim_x=int(img_dimx/quitx)
 dim_y=int(img_dimy/quity)
 #GUARDADO DE LA IMAGEN EN UNA CARPETA AUXILIAR
 #%cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
 #%cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala vacía
 %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala validacion
 #%cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala watson
 frame=Image.open('frame'+str(nframe)+'.jpg')
 foo=gridcrop(frame,120,120)
 bar=list2img(foo[16:144],dim_x,dim_y-1)
 %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa2
 cv2.imwrite('img.jpg',bar)
 img=Image.open('img.jpg')
 foo=gridcrop(img,120,240)
 %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pruebas2\all_classes
 guardar(foo,'n')
```

Lectura de los datos

In [6]:
```
img_width=120
img_height=240
test_datagen = ImageDataGenerator(rescale=1./255)
imgroom = test_datagen.flow_from_directory(
        "C:/Users/ialdea/Documents/TFM/Caso Practico/data/pruebas2",
        target_size=(img_width, img_height),
        batch_size=1,
        shuffle=False,
        class_mode='binary')
nb_samples=len(imgroom.filenames)
```

```
Found 64 images belonging to 1 classes.
```

Predicción

In [7]:
```
#Predicción
imgroom.reset()
predictions0=first_model.predict_generator(imgroom,steps=nb_samples,
    verbose=1,workers=1)
predictions=[0] * len(imgroom.filenames)
for i in range (0,len(imgroom.filenames)):
    bar=imgroom.filenames.index('all_classes\\n'+str(i)+'.jpg')
    predictions[i]=predictions0[bar][0]
ocup=[]
vac=[]
for i in range (0,len(predictions0)):
    foo=[i]
    if predictions[i]<0.2:
        ocup=ocup+foo
    else:
        vac=vac+foo
if len(ocup)>=1:
    print('La sala está ocupada')
else:
    print('La sala está vacía')
```

Out[7]:
```
64/64 [==============================] - 3s 52ms/step

    La sala está vacía
```

Estudio visual de la predicción

In [8]:
```
#Estudio visual de la predicción
 #%cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala ocupada
 #%cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala vacía
 %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala validacion
 #%cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala watson
 frame=Image.open('frame'+str(nframe)+'.jpg')
 dim_x=int(frame.size[0]/120)
 dim_y=int(frame.size[1]/120)
 foo=gridcrop(frame,120,120)
 foo1=list2img(foo[0:16],dim_x,1)
 bar=list2img(foo[16:144],dim_x,dim_y-1)
 %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa2
 cv2.imwrite('img.jpg',bar)
 img=Image.open('img.jpg')
 dim_xx=int(img.size[0]/120)
 dim_yy=int(img.size[1]/240)
 foo=gridcrop(img,120,240)
 foo=change_colorgreen(vac,foo)
 foo=change_colorred(ocup,foo)
 img_predict=list2img(foo,dim_xx,dim_yy)
 img_predict=np.concatenate((foo1,img_predict),axis=0)
 %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\prediccion
 cv2.imwrite('newprediction.jpg',img_predict)
```

Out[8]:
```
True
```

Análisis de la curva ROC

In [1]:
```
import tensorflow as tf
import tensorflow

from tensorflow import keras
from keras.preprocessing.image import ImageDataGenerator
from keras import models
from keras.models import model_from_json

import numpy as np
import matplotlib
import matplotlib.ticker as plticker
import matplotlib.pyplot as plt
import pickle
from PIL import Image
import cv2

import itertools
from sklearn.metrics import confusion_matrix
import sklearn.metrics as metrics
%matplotlib inline
%pylab
```

```
Using TensorFlow backend.
```

```
Using matplotlib backend: Qt5Agg
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
#CARGA DE LAS FUNCIONES
%cd C:\Users\ialdea\TFM
%run colors.py
%run gridcrop.py
%run list2img.py
%run Save.py
```

Carga el modelo

```
In [3]: #CARGA DEL MODELO
        #cargar json y crear el modelo
        json_file=open('CNNModel_11.json','r')
        loaded_model_json=json_file.read()
        json_file.close()
        first_model=model_from_json(loaded_model_json)
        #cargar pesos del modelo guardado
        first_model.load_weights("CNNModel_11.h5")
```

Definición de las características de aprendizaje

```
In [4]: #*****CARACTERÍSTICAS DEL APRENDIZAJE***********
        #
        first_model.compile(loss='binary_crossentropy',
                            optimizer='sgd',
                            metrics=['accuracy'])
```

Cálculo de las predicciones y curva ROC

```
In [5]: img_width=120
         img_height=240
         test_datagen = ImageDataGenerator(rescale=1./255)
         validation_generator = test_datagen.flow_from_directory(
                 "C:\\Users\\ialdea\\Documents\\TFM\\Caso Practico\\data\\roc",
                 target_size=(img_width, img_height),
                 batch_size=1,
                 shuffle=False,
                 class_mode='binary')
         validation_generator.reset()
         nb_samples=len(validation_generator.filenames)
         predictions=first_model.predict_generator(validation_generator,steps=nb_samples,
             verbose=1,workers=1)

Found 929 images belonging to 2 classes.
929/929 [==============================] - 45s 48ms/step


In [6]: number=6
         numberimages=929
         a=np.arange(number)
         fpr=[0] * len(a)
         tpr=[0] * len(a)
         d={}
         for i in a:
             d['labels_predictions{0}'.format(i+1)]=[0] * len(predictions)
             value=(i+1)/10
             for j in range (0,len(predictions)):
```

```
        if predictions[j]<value:
            d['labels_predictions'+str(i+1)][j]=0
        else:
            d['labels_predictions'+str(i+1)][j]=1
    _, fp, _,tp=confusion_matrix(validation_generator.classes,
    d['labels_predictions'+str(i+1)]).ravel()
    fpr[i]=fp/numberimages
    tpr[i]=tp/numberimages
tpr=[1]+tpr+[0]
fpr=[1]+fpr+[0]
tpr=tpr[::-1]
fpr=fpr[::-1]
roc_auc=metrics.auc(fpr, tpr)
plt.title('Curva ROC')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.plot(fpr,tpr,'o')
plt.ylabel('Tasa de Verdaderos Positivos')
plt.xlabel('Tasa de Falsos Positivos')
grid()
plt.show()
```

Estudio del modelo seleccionado

In [1]:
```
import tensorflow as tf
import tensorflow

from tensorflow import keras
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras.optimizers import SGD

from keras.utils import to_categorical
from keras import models
import numpy as np

%matplotlib inline

import matplotlib.pyplot as plt
import pickle
from PIL import Image
from keras.models import model_from_json

import random
import itertools
```

```
Using TensorFlow backend.
```

In [2]:
```
#CARGA DE LAS FUNCIONES
%cd C:\Users\ialdea\TFM
%run PlotConfusionMatrix.py
%run gridcrop.py
%run list2img.py
%run save.py
```

Carga del modelo y definición de características

In [3]:
```
#CARGA DEL MODELO
json_file=open('CNNModel_11.json','r')
loaded_model_json=json_file.read()
json_file.close()
```

```
first_model=model_from_json(loaded_model_json)
#cargar pesos al nuevo modelo
first_model.load_weights("CNNModel_11.h5")
#
#*****CARACTERÍSTICAS DEL APRENDIZAJE***********
#
first_model.compile(loss='binary_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Matriz de confusión

```
In [4]: #SELECCIÓN DE LOS FRAMES DE VALIDACIÓN DEL VÍDEO DE VALIDACIÓN
from time import time
start_time=time()
random.seed(0)
n=1800
test = [random.randint(0,18001) for _ in range(n)]
vac=test.copy()
#
#*****PREDICCIÓN*****
#
#def de las dimensiones
quitx=120
quity=120
rectangle_x=120
rectangle_y=240
img_dimx=1920
img_dimy=1080
dim_x=int(img_dimx/quitx)
dim_y=int(img_dimy/quity)
test_datagen = ImageDataGenerator(rescale=1./255)
#
ocup=[]
for j in test:
    #lectura del frame seleccionado y adaptación a la entrada de la red
    %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\sala validacion
    frame=Image.open('frame'+str(j)+'.jpg')
    foo=gridcrop(frame,120,120)
    bar=list2img(foo[16:144],dim_x,dim_y-1)
    %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pausa
    cv2.imwrite('img.jpg',bar)
    img=Image.open('img.jpg')
    foo=gridcrop(img,120,240)
    %cd C:\Users\ialdea\Documents\TFM\Caso Practico\data\pruebas\all_classes
    guardar(foo,'n')
    #creacion del conjunto de prediccion
    imgroom = test_datagen.flow_from_directory(
```

```
        "C:/Users/ialdea/Documents/TFM/Caso Practico/data/pruebas",
        target_size=(rectangle_x, rectangle_y),
        batch_size=1,
        shuffle=False,
        class_mode='binary')
    nb_samples=len(imgroom.filenames)
    #predicción con el modelo cargado
    imgroom.reset()
    predictions0=first_model.predict_generator(imgroom,steps=nb_samples,
            verbose=1,workers=1)
    predictions=[0] * len(imgroom.filenames)
    for i in range (0,len(imgroom.filenames)):
        bar=imgroom.filenames.index('all_classes\\n'+str(i)+'.jpg')
        predictions[i]=predictions0[bar][0]
        if predictions[i]<0.2:
            ocup=ocup+[j]
            break

for i in ocup:
    vac.remove(i)
countvacg=0
countvacb=0
countocupg=0
countocupb=0
for i in vac:
    if i <=6319:
        countvacg += 1
    else:
        countvacb += 1
        print ('El frame mal clasificado como vacía es el '+str(i))

for i in ocup:
    if i >=6334:
        countocupg += 1
    else:
        countocupb += 1
        print ('El frame mal clasificado como ocupada es el '+str(i))
Title='Matriz de confusión'
cmap=plt.cm.Blues
Classes=['ocupada','vacía']
Confusion_Matrix=array([[countocupg,countocupb],[countvacb,countvacg]])
plt.imshow(Confusion_Matrix,interpolation='nearest',cmap=cmap)
plt.title(Title)
plt.colorbar()
tick_marks=np.arange(len(Classes))
plt.xticks(tick_marks,Classes,rotation=45)
plt.yticks(tick_marks, Classes)
thresh=Confusion_Matrix.max()/2.
```

```
        for i,j in itertools.product(range(Confusion_Matrix.shape[0]),
                range(Confusion_Matrix.shape[1])):
            plt.text(j,i,Confusion_Matrix[i,j],horizontalalignment='center',
            color='white' if Confusion_Matrix[i,j] > thresh else 'black')
        plt.tight_layout()
        plt.ylabel('Valores Reales')
        plt.xlabel('Valores Predichos')
        end_time=time()-start_time
        hours=int(end_time/3600)
        minutes=(end_time-hours*3600)/60
        seconds=end_time%60
        print('El tiempo de ejecución es:',hours, 'horas ',int(minutes),
              'minutos y' ,seconds,' segundos')
```

```
El frame mal clasificado como vacía es el 6341
El frame mal clasificado como vacía es el 6325
El frame mal clasificado como vacía es el 6337
El frame mal clasificado como ocupada es el 5251
El frame mal clasificado como ocupada es el 5382
El tiempo de entrenamiento es: 1 horas  52 minutos y 12.464517593383789  segundos
```

```
In [5]: #POSITIVO--VACÍA, NEGATIVO--OCUPADA
        IndPositivReal=countvacg+countvacb
        IndNegativReal=countocupg+countocupb
        IndPositivFake=countvacg+countocupb
        IndNegativFake=countocupg+countvacb
        #TASA DE BIEN CLASIFICADOS
        ACC=(countocupg+countvacg)/n
        #TASA DE MAL CLASIFICADOS
        TMC=(countvacb+countocupb)/n
        #SENSIBILIDAD
        TPR=countvacg/(IndPositivReal)
        #ESPECIFICIDAD
        TNR=countocupg/(IndNegativReal)
        #PRECISIÓN
        PPV=countvacg/IndPositivFake
        #TASA DE FALSOS POSITIVOS
        FPR=countocupb/IndNegativReal
        #TASA DE FALSOS NEGATIVOS
        FNR=countvacb/IndPositivReal
        #Resultados
        print('Los resultados obtenidos a partir de la matriz de confusión:')
        print('Tasa de bien clasificados, '+str(ACC*100)+'%')
        print('Tasa de mal clasificados, '+str(TMC*100)+'%')
        print('Sensibilidad, '+str(TPR*100)+'%')
        print('Especificidad, '+str(TNR*100)+'%')
```

```
        print('Precisión, '+str(PPV*100)+'%')
        print('Tasa de falsos positivos, '+str(FPR*100)+'%')
        print('Tasa de falsos negativos, '+str(FNR*100)+'%')
```

```
Los resultados obtenidos a partir de la matriz de confusión:
Tasa de bien clasificados, 99.72222222222223%
Tasa de mal clasificados, 0.2777777777777778%
Sensibilidad, 99.55223880597015%
Especificidad, 99.82300884955752%
Precisión, 99.70104633781763%
Tasa de falsos positivos, 0.17699115044247787%
Tasa de falsos negativos, 0.4477611940298507%
```

Bibliografía

[1] Benson, C.C.; Lajish, V.L.; Rajamani, Kumar. *A novel skull stripping and enhancement algorithm for the improved brain tumor segmentation using mathematical morphology*, Int J Image Graph Signal Process, vol. 8, no. 7, p. 59–66, 2016, disponible en `https://www.researchgate.net/profile/Benson_C_C3/publication/305515800_A_Novel_Skull_Stripping_and_Enhancement_Algorithm_for_the_Improved_Brain_Tumor_Segmentation_using_Mathematical_Morphology/links/57985e2708aeb0ffcd06ef63.pdf`.

[2] Billah, Mohammad Ehtasham. *Classifying Microscopic Images for Acute Lymphoblastic Leukemia (ALL) using Bayesian Convolutional Neural Networks*, 2018, disponible en `http://www.diva-portal.org/smash/get/diva2:1233518/FULLTEXT01.pdf`.

[3] Cui, Zhihua et al. *Malicious code detection based on CNNs and multi-objective algorithm*, Journal of Parallel and Distributed Computing, vol. 129, p. 50–58, 2019.

[4] Diniz, Pedro Henrique Bandeira et al. *Detection of white matter lesion regions in MRI using SLIC0 and convolutional neural network*, Computer methods and programs in biomedicine, vol. 167, p. 49–63, 2018.

[5] GIMP, *GIMP-GNU Image manipulation program*, `https://www.gimp.org/`.

[6] Goldberg, Yoav ; Levy, Omer. *word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method*, arXiv preprint arXiv:1402.3722, 2014, disponible en `https://arxiv.org/pdf/1402.3722.pdf`.

[7] Gonzalez, Rafael C.; Woods, Richard E. *Digital Image Processing*, 2.ª ed., Prentice Hall, disponible en `http://web.ipac.caltech.edu/staff/fmasci/home/astro_refs/Digital_Image_Processing_2ndEd.pdf`

[8] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron. *Deep learning*, MIT press, 2016.

[9] Harbola, Shubhi ; Coors, Volker. *One dimensional convolutional neural network architectures for wind prediction*, Energy Conversion and Management, vol. 195, p. 70–75, 2019.

[10] Haykin, Simon S., et al. *Neural networks and learning machines*, 3.ª ed., Pearson education Upper Saddle River, 2009.

[11] IBM, *Watson IBM*, `https://www.ibm.com/watson`.

[12] Jupyter Notebook, *The Jupyter Notebook*, `https://jupyter-notebook.readthedocs.io/en/stable/`.

[13] Keras, *Keras:The Python Deep Learning Library*, `https://keras.io`.

[14] Koustubh, *ResNet,AlexNet,VGGNet,Inception: Understanding various architectures of Convolutional Networks*, https://cv-tricks.com/cnn/understand-resnet-alexnet-vgg-inception/.

[15] Liao, Shiyang, et al. *CNN for situations understanding based on sentiment analysis of twitter data*, Procedia computer science, vol. 111, p. 376–381, 2017, disponible en https://www.sciencedirect.com/science/article/pii/S1877050917312103.

[16] Machine Learning Cheatsheet, *Activation fuction*, https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html.

[17] Microsoft Azure, *Cognitive Services*, https://azure.microsoft.com/es-es/services/cognitive-services/.

[18] Muñoz Fernández, Álvaro. *Implementación del algoritmo SLICO en EspIN*, ETSI_Informatica, 2018, disponible en http://oa.upm.es/52471/1/TFG_ALVARO_MUNOZ_FERNANDEZ.pdf.

[19] Rstopup, *Dibujar las líneas de la cuadrícula sobre una imagen en matplotlib*, https://rstopup.com/dibujar-las-lineas-de-la-cuadricula-sobre-una-imagen-en-matplotlib.html.

[20] Ruder, Sebastian. *An overview of gradient descent optimization algorithms*, arXiv preprint arXiv:1609.04747, 2016, disponible en, https://arxiv.org/pdf/1609.04747.pdf.

[21] Scikit-learn, *Confusion matrix*, https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html.

[22] Szegedy, Christian, et al. *Going Deeper with Convolutions*, Computer Vision and Pattern Recognition (CVPR), disponible en, http://arxiv.org/abs/1409.4842

[23] TensorFlow, *An end-to-end open source machine learning platform*, https://www.tensorflow.org/.

[24] Theano, *Theano*, http://deeplearning.net/software/theano/.

[25] Toulis, Panos; Horel, Thibaut; Airoldi, Edoardo M. *Stable robbins-monro approximations through stochastic proximal updates*, arXiv preprint arXiv:1510.00967, 2015, disponible en https://arxiv.org/pdf/1510.00967.pdf.

[26] Wikipedia, *Stochastic gradient descent*, https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[27] Zhang, Ye; Wallace, Byron. *A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification*, arXiv preprint arXiv:1510.03820, 2015, disponible en https://arxiv.org/pdf/1510.03820.pdf.

[28] Zhang, Zhifei. *Derivation of Backpropagation in Convolutional Neural Network (CNN)*, University of Tennessee, Knoxville, TN, 2016.