UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

MÁSTER UNIVERSITARIO EN

BIG DATA ANALYTICS - MBI

TRABAJO FIN DE MÁSTER

Reto IBM: Reconocimiento de logotipos mediante modelos de redes neuronales convolucionales

JORGE GARCÍA CANTÚA

CURSO 2020-2021

TÍTULO: Reto IBM: Reconocimiento de logotipos mediante modelos de redes neuronales convolucionales

AUTOR: Jorge García Cantúa

TITULACIÓN: Máster Universitario en Big Data Analytics - MBI

DIRECTOR DEL PROYECTO: Luis Fernández Ortega y Jose Javier Ruiz Cobo

FECHA: Octubre de 2021

RESUMEN

En este trabajo mostraremos con un enfoque didáctico y guiado la construcción de una red neuronal convolucional desde cero.

Partiremos de una red neuronal convolucional sencilla e iremos realizando una serie de modificaciones en la misma, con el fin de evolucionar dicha red y conseguir mejores resultados en nuestras métricas de calidad.

Durante dichos experimentos nos encontraremos situaciones en las que no consigamos ningún tipo de mejoras en nuestros modelos, pero aun así, mostraremos todos los resultados obtenidos, ya que saber que tipo de modificaciones no resultan eficaces también es importante.

Tras los experimentos, conseguimos una serie de resultados que, poco a poco, van evolucionando y mejorando.

Al final del trabajo se propondrán una serie de mejoras a futuro y posibles aplicaciones de nuestro modelo.

Palabras clave: TensorFlow, Keras, Redes neuronales, Redes Neuronales Convolucionales, Python, Clasificador.

ABSTRACT

In this work we are going to show, in an easy way, how to construct a convolutional neural network from zero.

We will start from a simple convolutional neural network and we will do several changes in it in order to improve our quality metrics results.

During experiments we will encounter situations in which we do not achieve any improvements in our models, but we will still show all the results obtained. It is also important to know what kind of modifications are not effective.

After experiments, we have achieved results that, step by step, are improving and evolving.

At the end of the document we will propose several improvements and some ways to use our model.

Key words: TensorFlow, Keras, Neural Network, Convolutional Neural Network (CNN), Python, Classifier.

AGRADECIMIENTOS

Dedicado a todos aquellos que me apoyan en cada aventura que emprendo.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1. INTRODUCCIÓN

1.1 Motivación

Hoy en día, el aprendizaje automático, o *machine learning*, se ha establecido como una rama fundamental de la informática, necesaria en cualquier compañía y presente en la vida cotidiana de cualquier persona, ya sea en una herramienta de ordenador para tareas ofimáticas, en un escáner de acceso biométrico o incluso en un coche de conducción autónoma.

Esto se debe a que, en la actualidad, las empresas generan al día millones de datos, es decir, grandes volúmenes de datos que pueden ser explotados para potenciar la actividad de la compañía, esto es lo que se denomina hoy en día como *Big Data*. Como nos cuentan en [1], "*no existe unanimidad en la definición de Big Data, aunque sí un cierto consenso en la fuerza disruptiva que suponen los grandes volúmenes de datos y la necesidad de su captura, almacenamiento y análisis*".

El Big Data ha generado multitud de caminos a investigar dentro del *machine learning*: regresiones para realizar predicciones en base a datos del pasado, análisis de audio, análisis de texto o reconocimiento de objetos en imágenes, entre otros.

En el presente, el reconocimiento de objetos por imágenes se ha convertido en una aplicación del *machine learning* muy importante, ya sea en labores de seguridad, marketing, procesos industriales o estrategia empresarial entre otros. Para este tipo de trabajos son muy empleadas las redes neuronales, más concretamente la variante denominada Redes Neuronales Convolucionales o, en inglés, *Convolutional Neural Network (CNN)*, las cuales han adquirido una gran popularidad en los últimos años debido a su gran potencia dentro del *machine learning*.

Por todo ello, este trabajo estará enfocado en el desarrollo de una red neuronal convolucional capaz de clasificar imágenes de diferentes marcas.

1.2 Estado del arte

Como ya hemos mencionado previamente, el aprendizaje automático se ha consolidado como una de las ramas más importantes dentro del mundo informático, con un amplio camino de posibilidades que investigar.

A rasgos generales, podemos diferenciar entre dos grandes familias de algoritmos de aprendizaje automático: algoritmos de aprendizaje supervisado y algoritmos de aprendizaje no supervisado.

Los algoritmos de aprendizaje supervisado son aquellos en los que poseemos datos etiquetados para el entrenamientos, es decir, entrenamos nuestro algoritmo con datos en los que a priori se conoce la clase a la que pertenecen.

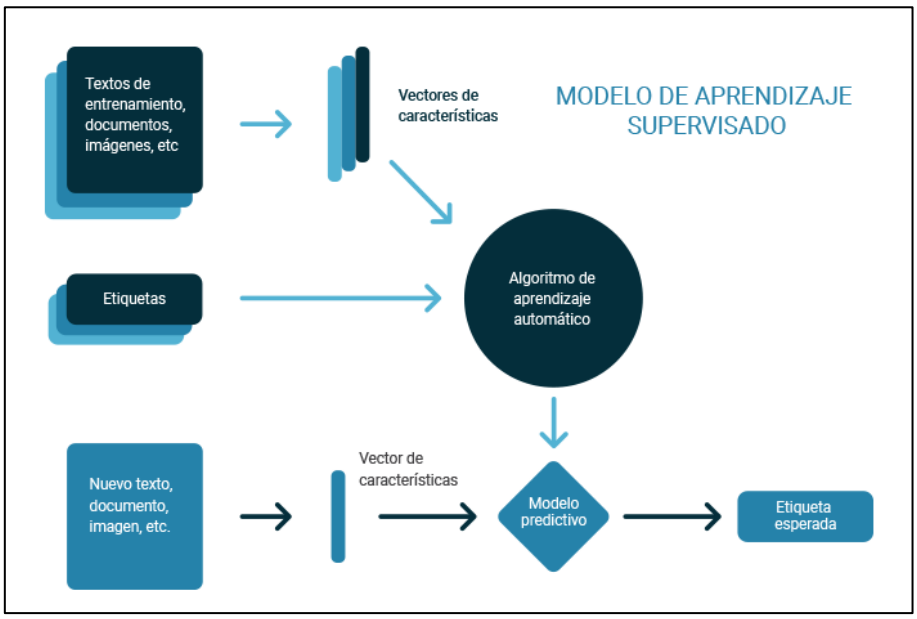


Ilustración 1: Diagrama de flujo de algoritmo de aprendizaje supervisado [2]

Los algoritmos de aprendizaje no supervisado son aquellos en los que se poseen datos no etiquetados, por lo que nuestro algoritmo tratara de extraer patrones y características similares midiendo la distancia entre las muestras.

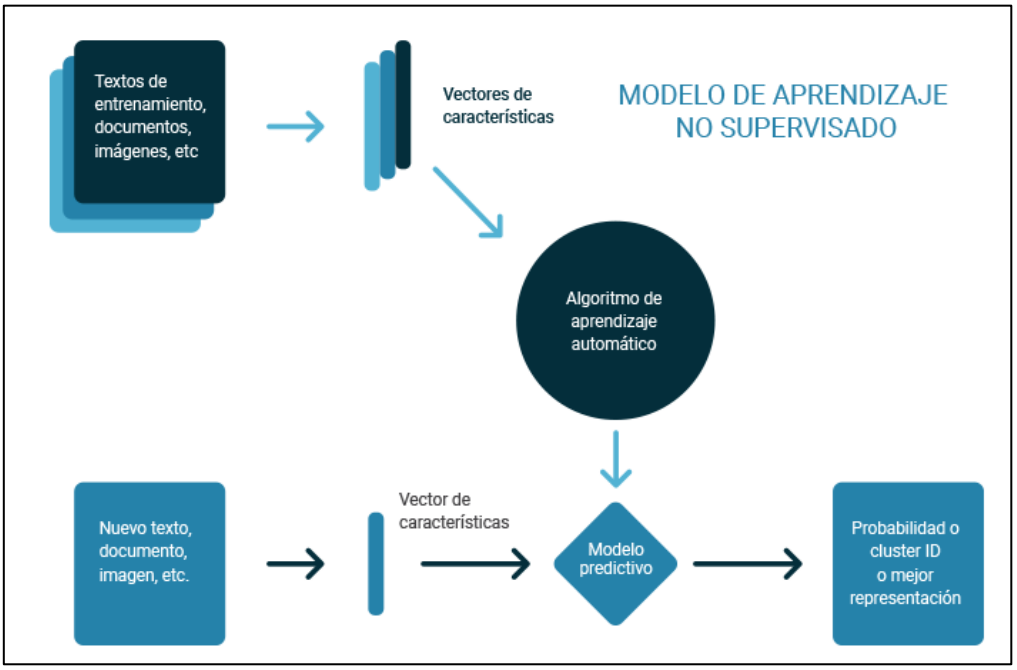


Ilustración 2: Diagrama de flujo de algoritmo de aprendizaje no supervisado [2]

En algunos libros como [3] también se menciona una tercera agrupación que son los algoritmos de aprendizaje por refuerzo. El aprendizaje por refuerzo está basado en un sistema de recompensas en el cual, un agente, que será nuestro modelo de aprendizaje automático, realiza una acción en un ambiente o entorno y, en función del estado en el

que se encuentre ese entorno, recibirá una recompensa positiva o negativa, en base a si ha desempeñado bien la acción o si se ha equivocado. [4]

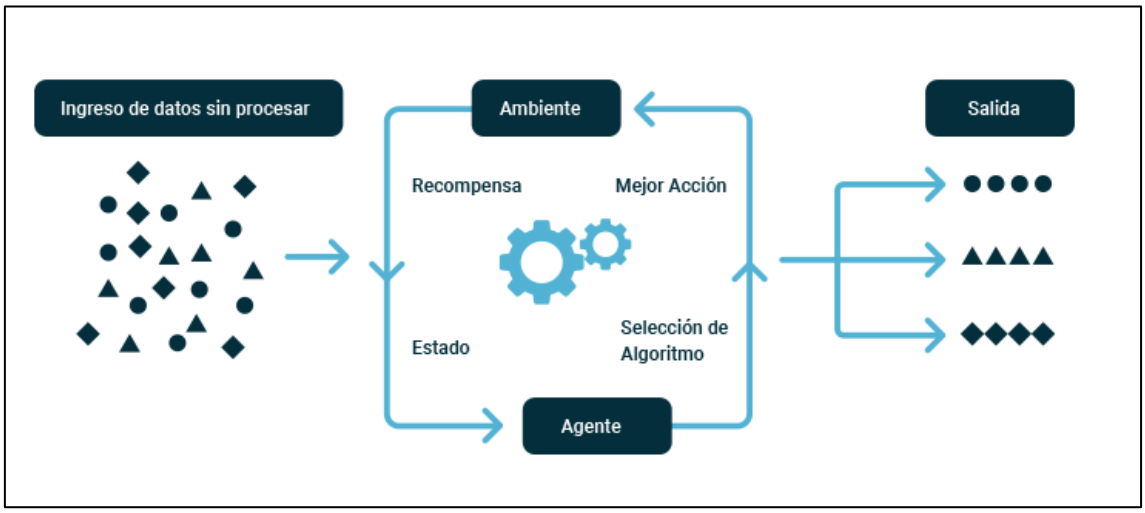


Ilustración 3: Diagrama de flujo de algoritmo de aprendizaje por refuerzo [2]

En algunas literaturas, aunque no es tan usual, también podemos encontrar una agrupación basada en los métodos simbólicos y en los métodos subsimbólicos.

Los métodos simbólicos están formados por aquellos sistemas de inteligencia artificial que emulan la manera de razonar humana, produciendo por tanto explicaciones y razonamientos comprensibles para los seres humanos.

Los métodos subsimbólicos están formados por aquellos algoritmos de inteligencia artificial que se centran en imitar físicamente el funcionamiento del cerebro humano, es decir, parte de la idea de que para simular la inteligencia es preciso simular los mecanismos físicos del cerebro.

Este tipo de algoritmos producen una representación no directamente interpretable por los seres humanos. Dentro de los métodos subsimbólicos nos encontramos con las redes neuronales.

Las redes neuronales conforman lo que hoy en día conocemos como *Deep Learning* y están formadas por capas de neuronas.

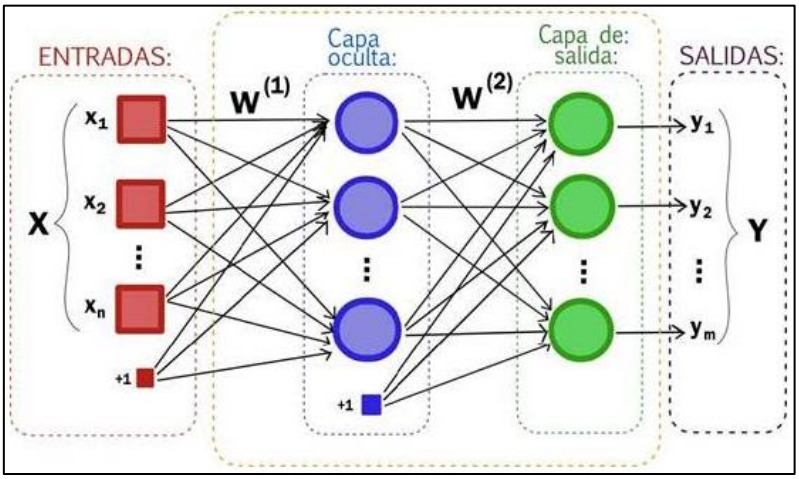


Ilustración 4: Ejemplo de red neuronal [5]

Las mencionadas neuronas reciben una serie de entradas provenientes de los datos de entrada y, además, por cada entrada tendrán un peso que deberán aprender durante el entrenamiento. Estas entradas y pesos pasan por una función matemática que nos devolverá un valor numérico, el cual nos será de ayuda a la hora de tomar una decisión. Las redes neuronales en su proceso de aprendizaje utilizan funciones de retropropagación en las que ajustan los pesos en función de la salida dada por la red neuronal y el valor esperado. Las redes neuronales están formadas por distintas capas: la primera es conocida como capa de entrada, la última como capa de salida y el resto como capas ocultas.

Dentro de las redes neuronales nos encontramos con una variante denominada redes neuronales convolucionales, muy empleadas cuando en nuestro proyecto debemos trabajar con imágenes.

Como se menciona en [6], las redes neuronales convolucionales (CNN) son un tipo concreto de redes neuronales. Dichas redes neuronales ya eran utilizadas a finales del siglo XX, pero en los últimos años han tenido un gran auge debido a los grandes resultados obtenidos en reconocimiento de imágenes, impactando así en el área de visión por computador.

Las redes neuronales convolucionales hacen la suposición de que siempre van a recibir imágenes, lo que nos permite configurar una serie de parámetros que en otro tipo de redes no podríamos realizar.

Las redes neuronales convolucionales tienen dos tipos de capas que las diferencian de las demás redes neuronales y que están especializadas en dos de operaciones: capa de convolución y capa de *pooling*.

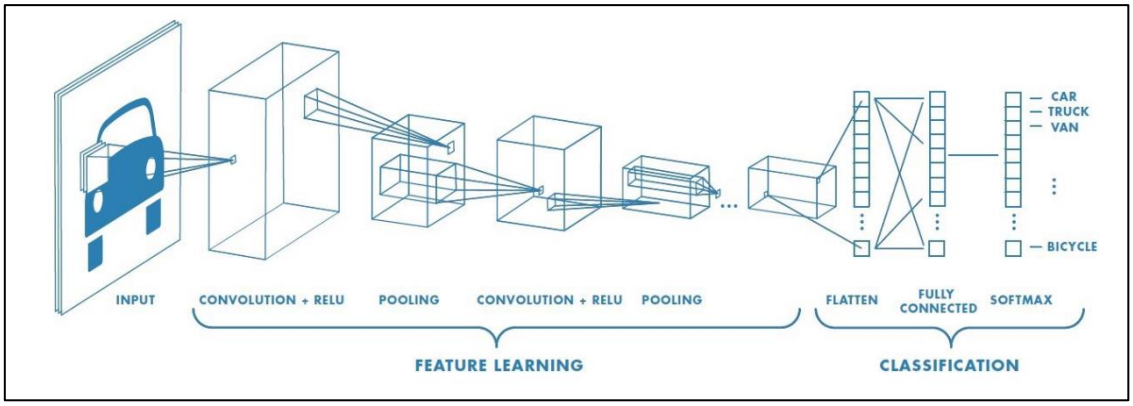


Ilustración 5: Ejemplo de Red Neuronal Convolucional (CNN) [7]

La capa de convolución es aquella en la que se aplican sobre la imagen de entrada una serie de filtros que nos permitirán extraer características de dicha imagen. Estos filtros se tratan de ventanas que contienen una serie de pesos que aplicarán al pixel sobre el que se encuentren en cada iteración, ya que se van desplazando sobre la imagen. Los pesos serán aprendidos por la neurona en cada iteración durante el entrenamiento.

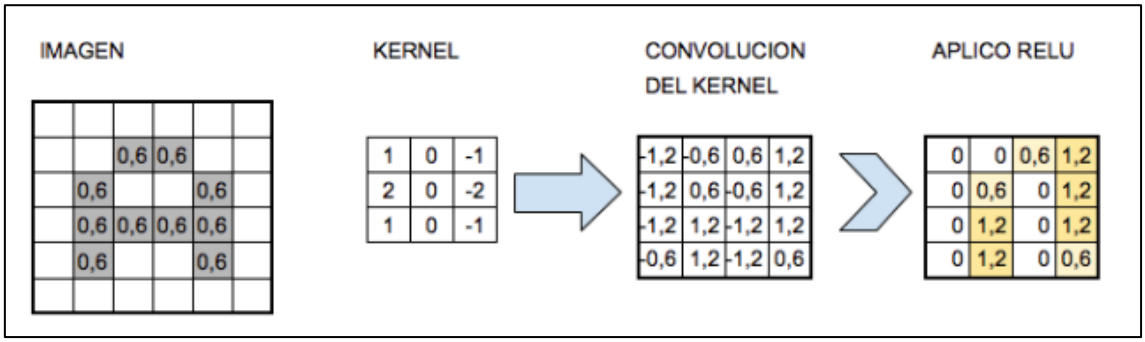


Ilustración 6: Ejemplo de aplicación de kernel de convolución [8]

La capa de *pooling* se trata de una máscara que se aplica sobre la salida de la capa de convolución y que reduce el volumen espacial de la entrada que reciba dicha capa.

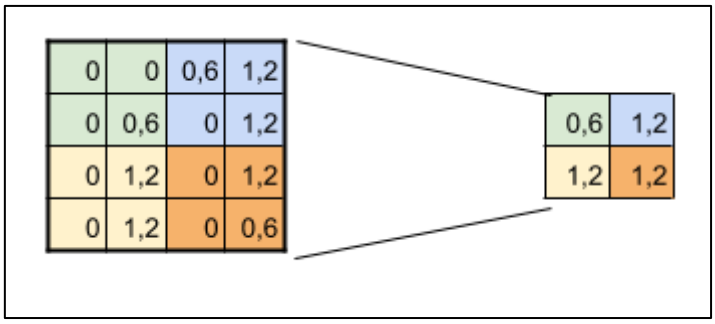


Ilustración 7: Ejemplo de capa Max-Pooling de 2x2 [8]

Dentro del reconocimiento de imágenes y objetos que, como hemos visto, es un mundo complejo y lleno de infinitas posibilidades que explorar, podemos incluir el reconocimiento de logotipos que es lo abarca nuestro Trabajo Final de Máster.

1.3 Objetivos

El principal objetivo de este Trabajo Final de Máster consiste en la implementación, mediante Python, TensorFlow y Keras, de una red neuronal clasificadora capaz de identificar cada uno de los logotipos asociados a una marca que le demos como entrada a nuestro modelo.

El conjunto de datos constará de 23 tipos de logotipos, es decir, 23 clases de las cuales tendremos distintas imágenes de dichos logotipos, en total 1611 imágenes. Dichas imágenes ya estarán separadas por marcas, es decir, tendremos un conjunto de datos etiquetado. Este dataset podrá ser ampliado mediante transformaciones de las imágenes proporcionadas.

Un objetivo fundamental que debemos conseguir en la implementación de nuestra red neuronal será alcanzar una precisión en el modelo superior al 50%, es decir, que sea mejor que un clasificador trivial.

Para implementar nuestra red neuronal y alcanzar el mayor nivel de precisión, se seguirá un enfoque similar al siguiente:

1. Desarrollo de script de carga de datos.
2. Preprocesamiento de los datos proporcionados.
3. Implementación de una red neuronal inicial de referencia.
4. Ajustes y mejoras sobre dicha red neuronal de referencia.
5. Comparación de los resultados obtenidos a través de diferentes métricas.
6. Selección del modelo más adecuado.

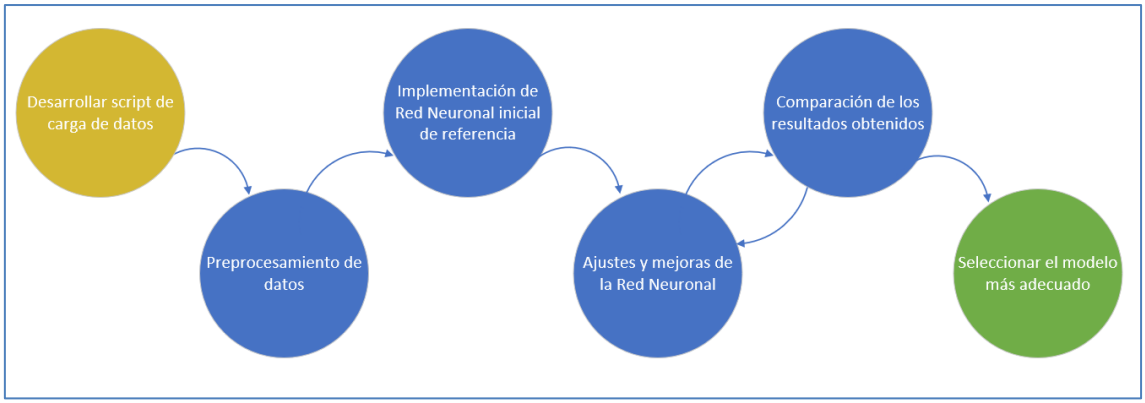


Ilustración 8: Diagrama de flujo de objetivos

1.4 Tecnologías empleadas

1.4.1 Python

Hoy en día, existen numerosos lenguajes de programación destinados a distintos fines y enfocados a diferentes paradigmas de programación. La elección de un lenguaje u otro dependerá del problema que queramos resolver y también del programador, es decir, debemos resolver el problema con un lenguaje adecuado, pero también debemos valorar aquellos lenguajes con los que nos manejemos con mayor soltura.

En problemas de ciencia de datos como el que nos ocupa en este trabajo, Python se ha convertido en uno de los referentes a nivel mundial, convirtiéndose en uno de los tres lenguajes de programación más populares.

Python es un lenguaje de programación de propósito general de tipo *open source*, el cual se puede emplear como un lenguaje de programación orientado a objetos, lenguaje funcional o lenguaje procedural. [9]

Python puede ser empleado para construir programas estructurados en clases, pero también se puede emplear para construir scripts de código útiles en múltiples ámbitos de la programación.

Algunos de los aspectos que destacan de Python son [10]:

- **Calidad del software**: Python se considera un lenguaje fácil de leer, así como, en general, reutilizable y mantenible. Su uniformidad lo hace un lenguaje fácil de entender.
- **Productividad en el desarrollo**: En Python hay que escribir menos código, lo cual se traduce en menor tiempo de escritura, menor dedicación a la depuración

del código y menor esfuerzo de mantenimiento. El código de Python suele ser de un tercio a un quinto del tamaño de un código equivalente en Java, C o C++.

- **Portabilidad en los programas**: Python ofrece multitud de opciones para poder trabajar y ejecutar nuestro código en una gran variedad de sistemas.
- **Soporte a librerías**: Python posee un largo conjunto de librerías preconstruidas y funcionalidades portables. También pueden ser utilizadas numerosas librerías de terceros.
- **Componentes para la integración**: A través de una gran variedad de mecanismos de integración, los códigos de Python se pueden comunicar fácilmente con otras aplicaciones. Dichas aplicaciones pueden estar escritas en otros lenguajes como C o Java, e incluso se puede comunicar con frameworks entre otras cosas.

1.4.2 TensorFlow

TensorFlow fue lanzado por Google en 2015 y la documentación oficial lo define como una plataforma de código abierto de extremo a extremo para el aprendizaje automático, la cual cuenta con un ecosistema de herramientas, bibliotecas y recursos de la comunidad que permite crear y desarrollar aplicaciones con tecnología de aprendizaje automático fácilmente. [11]

Se trata de un *framework* útil para cualquier cómputo que requiera un alto rendimiento y una fácil distribución. Además, es capaz de manejar diferentes arquitecturas: CPU, GPU y TPU.

Es un framework magnifico en lo referido a la creación de redes neuronales, desde redes neuronales pequeñas hasta redes neuronales complejas para el reconocimiento de imágenes o procesamiento de lenguaje natural (*NLP*). [12]

1.4.3 Keras

Keras es una API de alto nivel escrita en Python con múltiples *backends*, entre los que se encuentra: Google TensorFlow, Microsoft CNTK, Amazon MxNet y Theano. El objetivo principal de Keras es ofrecer una librería accesible y fácil de utilizar para desarrollar experimentos rápidos. [12] [13]

Keras fue desarrollada y es mantenida por Fraçois Chollet, ingeniero de Google, y su código se encuentra liberado bajo la licencia permisiva del MIT. [6] Keras se convirtió en la API oficial de alto nivel de TensorFlow en la versión 2 de TensorFlow.

1.4.4 Google Colaboratory

Google Colab o Colaboratory se trata de un entorno de desarrollo de código Python alojado en la nube de Google, al cual podemos acceder con un simple navegador web. En dicho entorno se emplean los denominados *Jupyter Notebooks,* muy útiles en el mundo de la ciencia de datos para un desarrollo ágil de programas escritos en Python [6].

Las características principales de Google Colab son [14]:

- No requiere de una configuración previa ya que se proporcionan las librerías cargadas.
- Ofrece acceso gratuito, aunque limitado, a GPUs y TPUs, lo cual puede ser muy útil a la hora de ejecutar modelos de redes neuronales.
- Facilidad a la hora de compartir contenido ya que todos los notebooks se almacenan en Google Drive, además podemos integrar Google Colab con Drive para leer ficheros de allí.

Los recursos ofrecidos por Google de manera gratuita son más que suficientes para ejecutar modelos de carácter académico, pero, en caso de necesitar mayor potencia de cómputo, podemos pagar por una versión premium con mejores características computacionales.

1.5 Estructura del proyecto

La presente memoria se divide en cuatro capítulos, cuyo contenido se resume a continuación:

1) En el capítulo uno se realiza una introducción al trabajo realizado y a las tecnologías empleadas.
2) En el segundo capítulo se presentan los datos empleados.
3) En el capítulo número tres se exponen los experimentos realizados y como se han llevado a cabo. Durante este capítulo realizaremos una comparación en los seis primeros experimentos del tiempo que lleva ejecutar nuestros modelos en arquitecturas basadas en CPU y en arquitecturas basadas en GPU.
4) En el cuarto capítulo realizaremos una explicación de los resultados obtenidos, así como expondremos una serie de conclusiones.

Capítulo 2. CONJUNTO DE DATOS

2.1 Conjunto de datos original

El conjunto de datos del que disponemos inicialmente y con el que vamos a trabajar para entrenar y validar nuestra red neuronal, está compuesto por un conjunto de imágenes en color, etiquetadas y de tamaño 256 x 256 píxeles.

Dicho conjunto de datos consta de un total de 23 clases, en total 1611 imágenes. Cada clase hace referencia a una marca conocida que nuestra red neuronal deberá aprender a identificar.

En la siguiente tabla podemos ver los nombres de las clases y el número de imágenes en cada una de ellas:

Nombre de clase	Número de ejemplos
AMD	70
Aquafina	70
D-Link	70
Disney	70
Domino's Pizza	70
Hellmann's	70
IBM	71
Kitkat	70
LG	70
Lipton	70
McDonalds	70
Milka	70
Monster	70
Nestea	70
Pac-Man	70
Pepsi	70
Pizza Hut	70
Red Bull	70
Samsung	70
Sony	70
Tic Tac	70
Universal	70
Dell	70

Tabla 1: Número de imágenes por clase

Como podemos observar, las clases se encuentran balanceadas teniendo todas ellas un total de 70 ejemplos, a excepción de la clase “IBM” que tiene 71 ejemplos.

Para facilitar el aprendizaje y mejorar los resultados de nuestro algoritmo, vamos a etiquetar cada clase con un valor numérico del 0 al 22. La correspondencia de clases con los números asignados puede verse en la siguiente tabla:

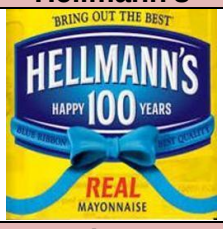
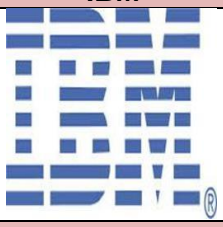
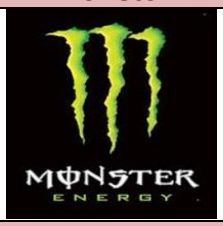
Nombre de clase	Número de clase asignado
AMD	**0**
Aquafina	**1**
D-Link	**2**
Disney	**3**
Domino's Pizza	**4**
Hellmann's	**5**
IBM	**6**
Kitkat	**7**
LG	**8**
Lipton	**9**
McDonalds	**10**
Milka	**11**
Monster	**12**
Nestea	**13**
Pac-Man	**14**
Pepsi	**15**
Pizza Hut	**16**
Red Bull	**17**
Samsung	**18**
Sony	**19**
Tic Tac	**20**
Universal	**21**
Dell	**22**

Tabla 2: Correspondencia de clases con su número

De este modo, procederemos a renombrar cada uno de los directorios propuestos con su valor numérico correspondiente y, cada una de las imágenes de marcas, con el número de la clase asociado y un número de secuencia. Con ello, además de facilitar el aprendizaje de nuestro algoritmo, también nos facilitaremos las tareas preliminares de cargas de datos.

A continuación, vamos a visualizar algunos de los ejemplos que componen cada una de nuestras clases:

AMD	Aquafina
	
Disney	**D-Link**

	
Domino's Pizza	Hellmann's
	
IBM	KitKat
	
LG	Lipton
	
McDonalds	Milka
	
Monster	Nestea
	
Pac-Man	Pepsi
	
Pizza Hut	Red Bull

	
Samsung	**Sony**
	
Tic Tac	**Universal**
	

Tabla 3: Ejemplo de imágenes por clase

Como clase extra, nos corresponde en este caso la clase “Dell”:

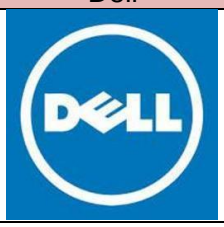
Dell


Tabla 4: Ejemplo de imagen perteneciente a la clase extra

En algunas clases podemos encontrar imágenes que incorporarán ruido a nuestro clasificador, por ejemplo, las siguientes dos imágenes se encuentran dentro de la clase “Pac-Man”:

Ilustración 9: Ejemplo de imágenes con ruido

Como podemos observar, estas imágenes tratan de confundir a nuestro algoritmo con la clase “LG”.

Trataremos de que todos estos problemas afecten lo menos posible a la eficacia de nuestro clasificador.

2.2 Carga de datos

Para la carga de datos, hemos diseñado una función en Python que nos permite leer el directorio donde se encuentran almacenadas nuestras imágenes y cargar una a una las imágenes en una lista, al mismo tiempo, cargamos las etiquetas en otra lista diferente.

Posteriormente, hemos diseñado otra función que nos permite separar de manera aleatoria nuestro conjunto de datos, según un porcentaje de división dado como parámetro, en un conjunto de validación y otro de entrenamiento.

Finalmente, obtenemos cuatro Numpy arrays:

- Array con las imágenes de entrenamiento.
- Array con las etiquetas numéricas vinculadas a las clases de las imágenes de entrenamiento.
- Array con las imágenes de validación.
- Array con las etiquetas numéricas vinculadas a las clases de las imágenes de validación.

Cabe mencionar que cada una de las imágenes almacenadas en el array se trata a su vez de un Numpy array de tamaño (256, 256, 3), es decir, posee un alto y un ancho de 256 píxeles y 3 canales de color, ya que se trata de imágenes RGB.

En general, el criterio que hemos decidido para dividir nuestros datos es el siguiente:

- El 80% de los datos irá destinado al entrenamiento de nuestra red neuronal.
- El 20% de los datos irá destinado a probar nuestro algoritmo de clasificación.

Podemos destacar que, tanto para la fase de entrenamiento como para las pruebas, cada clase constará del mismo número de imágenes que el resto, es decir, una clase no tendrá más imágenes que otra ni en la fase de entrenamiento ni en la fase de pruebas.

Capítulo 3. EXPERIMENTOS

3.1 Preprocesamiento de datos

Antes de comenzar a construir nuestro modelo, entrenarlo y evaluarlo, debemos hacer una serie de preprocesamientos de los datos que contribuirán de manera favorable al rendimiento de nuestro algoritmo.

Como hemos comentado en apartados anteriores, tras la carga de los datos obtenemos:

- Los vectores de imágenes, que son matrices de números enteros cuyos valores van del 0 al 255.
- Los vectores de etiquetas, que contienen un número del 0 al 22 en función de la clase correspondiente a cada imagen.

A ambos tipos de vectores les aplicaremos las ya mencionadas técnicas de preprocesamiento.

Por un lado, al vector de etiquetas le aplicaremos una técnica denominada "*One-Hot encoding*". Esta técnica consiste en transformar cada uno de los números en un vector binario de N posiciones, que contendrá un uno en la posición correspondiente a dicho número, donde N es la cantidad de clases que contenga nuestro problema, en este caso 23 clases. Por ejemplo, si queremos codificar la clase número cuatro, el vector quedaría de la siguiente manera:

0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Como podemos observar, tenemos un "1" en la posición cinco del vector para representar la clase cuatro, recordemos que las clases se empiezan a enumerar en el valor 0.

Hemos tomado la decisión de aplicar esta transformación ya que nos encontramos con una variable dependiente de tipo categórica nominal, es decir, no existe una relación de orden entre unas clases y otras, por ello, se transforma en un vector binario para mejorar el desempeño de nuestro algoritmo.

Por otro lado, nuestros vectores de imágenes los normalizaremos dividiendo todos sus valores en las tres capas entre 255. Debemos tener en cuenta que dividimos entre 255 porque el rango de valores posibles para un pixel va de 0 a 255.

Aplicamos esta normalización para que el valor de cada pixel de la matriz sea más pequeño, concretamente un valor entre 0 y 1, y que sea un rango más próximo al de los pesos que nuestra red neuronal debe aprender.

3.2 Métricas empleadas

3.2.1 Macro F1-Score

Una de las métricas de calidad que vamos a emplear en nuestros experimentos, siendo la que tomaremos como mayor referencia, es la F1-Score, pero en concreto la versión macro-media F1-Score. Antes de entrar en detalle con esta variante, veamos que es la medida F1-Score.

En general, en *machine learning* se prefieren clasificadores con altos valores de precisión y cobertura, pero por lo general existe un balance entre ellos, es decir, cuando uno de ellos aumenta, el otro decrece. [15]

La medida F1-Score trata de realizar una media entre la precisión y la cobertura, pero no una media aritmética sino una media armónica, es decir, en este caso penaliza cuando hay valores bajos. La fórmula de la F1-Score es la siguiente:

$$\text{F1 Score} = 2 \cdot \frac{precisión \cdot cobertura}{\text{precisión} + \text{cobertura}}$$

Pongamos dos ejemplos para una mejor compresión:

- Imaginemos que tenemos una precisión del 100% y una cobertura de 0%, la medida F1-Score nos daría un resultado de 0%, no de 50%.
- En el caso de tener una cobertura de 100% y una precisión del 80%, obtendríamos una F1-Score del 75% y no del 80%

En conclusión, la F1-Score busca siempre que ambas métricas sean elevadas.

En problemas multiclase podemos emplear la medida F1-Score mediante la variante Macro F1-Score, que pondera por igual todas las clases, o mediante la variante Micro F1-Score, que pondera la métrica en función del número de ejemplos de cada clase.

En nuestro caso, como ya hemos mencionado, vamos a emplear la Macro F1-Score ya que todas las clases están equilibradas. La Macro F1-Score consiste en calcular las métricas F1-Score de cada clase por separado y posteriormente hacer una media aritmética de todas ellas. La fórmula sería la siguiente:

$$Macro\ \text{F1 Score} = \frac{\sum_{n}^{1} \text{F1 Score}_{clase\ n}}{n}$$

3.2.2 Accuracy

La segunda métrica de calidad que emplearemos es el *accuracy*, esta métrica será empleada a modo de apoyo junto con la métrica F1-Score ya vista con el fin de poder validar con cierta veracidad que los resultados obtenidos son correctos.

Esta es una métrica algo más sencilla, únicamente consiste en calcular el número de aciertos de nuestro clasificador entre el número total de ejemplos o predicciones. [16]

La fórmula para calcular el *accuracy* es la siguiente:

$$accuracy = \frac{\text{número total de aciertos}}{\text{número total de predicciones}}$$

3.2.3 Categorical crossentropy

La métrica que hemos empleado con la función de pérdida es la denominada *categorical crossentropy*.

Es un tipo de cáculo de función de pérdida muy empleado, que parte de la *Cross-Entropy Loss*. Hemos elegido esta métrica debido a que es la que mejor encaja con nuestro tipo de problema, en concreto, es una de las más adecuadas para emplear en los siguientes casos [17]:

- Problemas en los que tengamos dos o más clases.
- Las etiquetas deben estar representadas como un vector *one-hot encoding*.
- Es muy adecuada cuando tienes una capa de salida de tipo “Softmax”.

Por todo ello, nos decantamos por esta métrica para el cálculo de la función de pérdida.

3.3 Experimento 1

3.3.1 Configuración de modelo y resultados

En este apartado, vamos a entrar más en detalle con las redes neuronales.

En primer lugar, revisaremos la configuración de una primera red neuronal, así como los resultados obtenidos, tomando estos como punto de referencia a mejorar.

La topología de esta primera red neuronal consistiría en las siguientes capas, mostradas en orden de entrada a salida:

1. Capa 1 de convolución: Capa de entrada de elementos de 256x256 y 3 capas. En este caso se aplican 64 filtros de 5x5 y una función de activación *ReLU*.
2. Capa 1 de *pooling*: En esta capa empleamos un filtro de 2x2 para reducir a la mitad la salida de la anterior capa.

3. Capa 1 de aplanamiento: Convierte la salida de la anterior capa en un vector de una dimensión.
4. Capa de salida: La capa de salida emplea una función *Softmax* de 23 probabilidades, una para cada una de las clases. En este caso, debería ser la función más adecuada para nuestro problema de clasificación multiclase.

```
Model: "Experimento_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 64)      4864
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 64)      0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 1016064)           0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                23369495
=================================================================
Total params: 23,374,359
Trainable params: 23,374,359
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 10: Topología de red neuronal experimento 1

En cuanto a la configuración elegida para los hiperparámetros de nuestra primera aproximación de red neuronal es la siguiente:

- La función de perdida elegida es "*categorical_crossentropy*", esto es así porque necesitamos una salida categórica y, además, nos encontramos en un problema multiclase. Para poder utilizar esta función de pérdida debemos emplear una función de activación *Softmax* en nuestra capa de salida, al igual que en nuestra red como hemos mencionado previamente.
- El optimizador elegido es el "*sgd*" (*Stochastic gradient descent*) debido a que hemos decidido que es el más apropiado desde nuestro punto de vista para minimizar la función de perdida.
- Las métricas elegidas son la precisión y la F1-Score, aunque la medida principal será la F1-Score.

En este caso para el entrenamiento elegimos:

- Un tamaño de paquete de 100 para el parámetro "*batch_size*".
- Cinco etapas de entrenamiento en el parámetro "*epochs*".
- El parámetro "*verbose*" lo ponemos a 1 para que nos muestre el progreso del entrenamiento.

En cuanto a los resultados obtenidos, alcanzamos en la fase de entrenamiento con los datos de entrenamiento una precisión del 48.45% y un F1-Score del 46.59%. Podemos ver en la siguiente gráfica la evolución de las métricas en cada una de las etapas del entrenamiento:

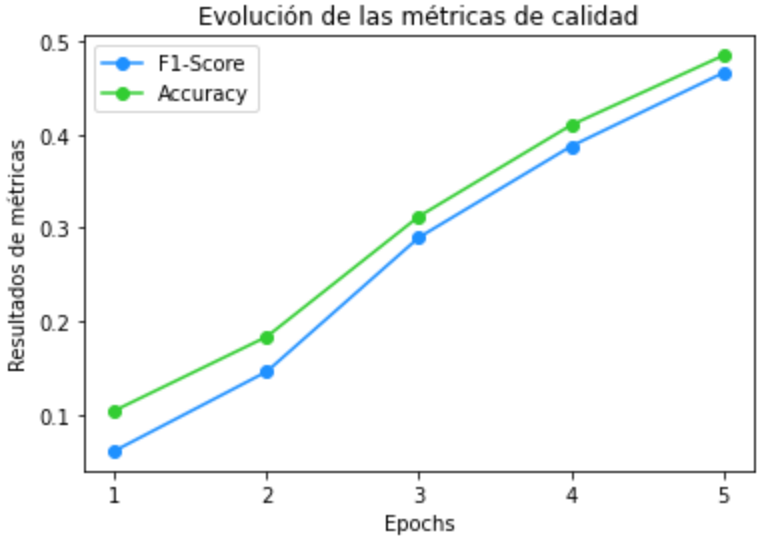


Ilustración 11: Evolución de métricas de calidad en experimento 1

En cuanto al valor de pérdida obtenido al final del entrenamiento es de 1.99. Podemos ver en la siguiente gráfica la evolución de la función de pérdida en cada una de las etapas:

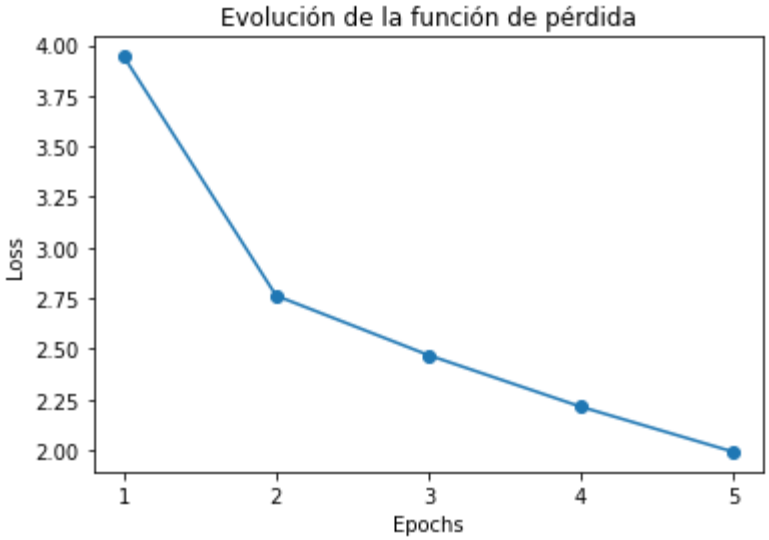


Ilustración 12: Evolución de pérdida en experimento 1

Como vemos, y como es lógico, a medida que avanzan las etapas de entrenamiento, nuestras métricas de calidad aumentan y nuestra función de pérdida disminuye, pero alcanzamos una precisión del 48% y un F1-Score de 46%, es decir, es peor que un clasificador trivial.

Cuando lo ponemos a prueba con datos que no ha visto nunca, es decir, con el conjunto de validación, obtenemos una precisión del 36.22% y un F1-Score del 34.78%. Como vemos, es un resultado que no es aceptable.

En los próximos experimentos trataremos de modificar los hiperparámetros de nuestra red neuronal y su topología para incrementar la calidad de nuestro modelo.

3.3.2 Ejecución con CPU vs Ejecución con GPU

Desde el experimento 1 hasta el experimento 6, incluidos, tendremos este segundo apartado denominado “Ejecución con CPU vs Ejecución con GPU” en el que compararemos los tiempos de ejecución del mismo modelo de red neuronal al emplear una arquitectura basada en CPU y una arquitectura basada en GPU.

En el caso de este primer experimento, podemos observar los resultados de los tiempos en los siguientes gráficos:

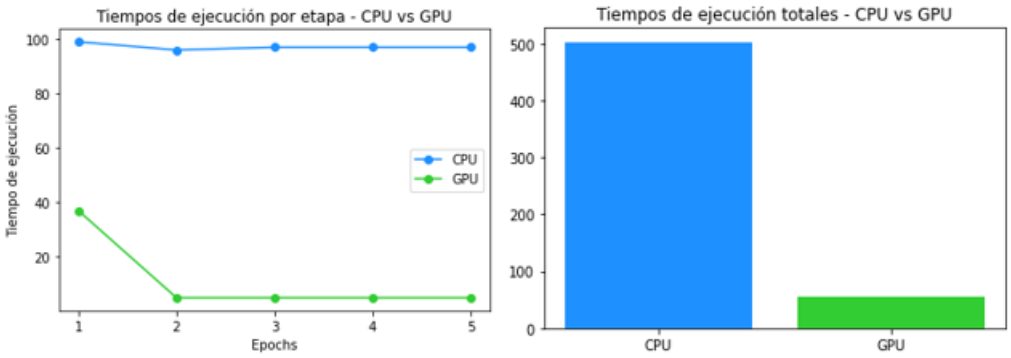


Ilustración 13: Comparación de tiempos en experimento 1

En primer lugar, los colores empleados en los gráficos son azul para representar los tiempos de la ejecución con CPU y verde para el caso de la ejecución con GPU. En segundo lugar, en el lado izquierdo nos encontramos con una gráfica que representa los tiempos empleados por cada arquitectura en cada una de las etapas durante el entrenamiento, y en el lado derecho nos encontramos con una gráfico de barras que representa el tiempo total de cada arquitectura empleado durante la fase de entrenamiento.

Si nos fijamos en el gráfico de los tiempos de cada etapa, únicamente el tiempo empleado en la primera etapa con GPU forma un pico más alto, pero después de ello, el resto son tiempos muy bajos en comparación con la CPU que son bastante elevados, encontrándonos en un modelo muy simple.

Si nos fijamos en el gráfico de la derecha, vemos que el tiempo total empleado por la GPU (56.35 segundos) es notablemente muy inferior que el utilizado para la CPU (502.44 segundos), concretamente la GPU representa únicamente un 11.22% del tiempo empleado por la CPU.

3.4 Experimento 2

3.4.1 Configuración de modelo y resultados

En este caso, añadiremos dos nuevas capas:

- Una segunda capa de convolución después de la primera capa de *pooling* que extraerá 64 filtros de tamaño 5x5 con una función de activación de tipo “ReLU”.

- Una segunda capa de *pooling* después de la nueva capa de convolución que aplicará un tamaño de ventana de 2x2.

Con esta nueva topología, el resumen de los parámetros a extraer quedaría de la siguiente forma:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 64)      4864
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 64)      0
_________________________________________________________________
Capa_conv_2 (Conv2D)         (None, 122, 122, 64)      102464
_________________________________________________________________
Capa_Pool_2 (MaxPooling2D)   (None, 61, 61, 64)        0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 238144)            0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                5477335
=================================================================
Total params: 5,584,663
Trainable params: 5,584,663
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 14: Topología de red neuronal experimento 2

En este caso, tras el entrenamiento, obtenemos una precisión del 36.41% y un F1-Score del 35.92%. Podemos ver un resumen de la evolución de las métricas durante el entrenamiento en la siguiente gráfica:

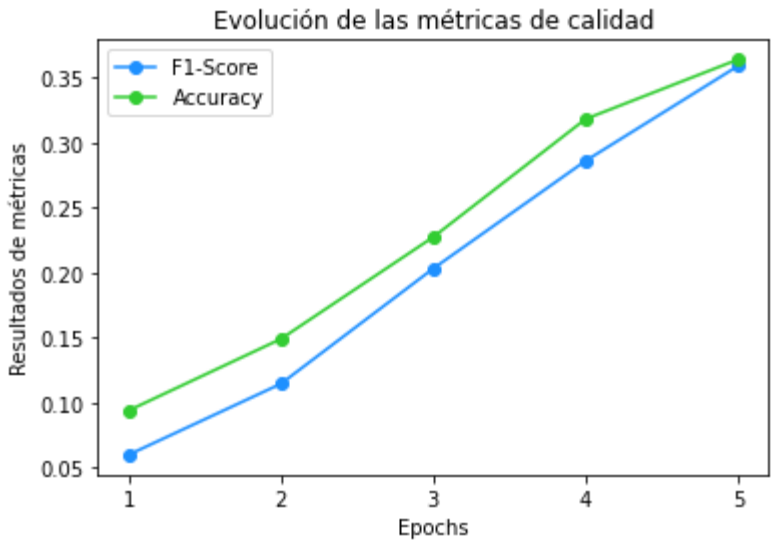


Ilustración 15: Evolución de métricas de calidad en experimento 2

Durante el entrenamiento obtenemos una pérdida de 2.30. Podemos observar en la siguiente gráfica la evolución de la función de pérdida:

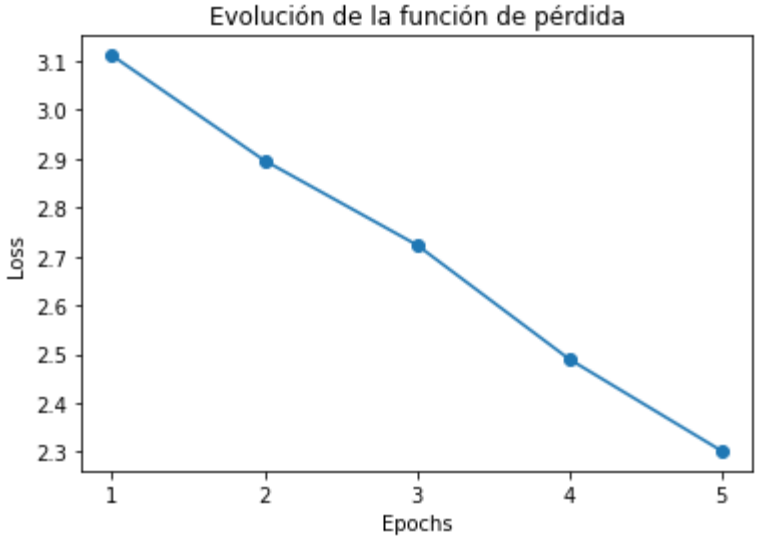


Ilustración 16: Evolución de pérdida en experimento 2

Con este diseño, obtenemos una precisión del 29.10%, una F1-Score del 23.23% y una pérdida de 2.38 al someter a nuestro algoritmo a una evaluación con datos nunca vistos. Los resultados obtenidos siguen sin ser buenos, de hecho, han empeorado los resultados notablemente.

3.4.2 Ejecución con CPU vs Ejecución con GPU

En el caso del segundo experimento, podemos observar los resultados de los tiempos en los siguientes gráficos:

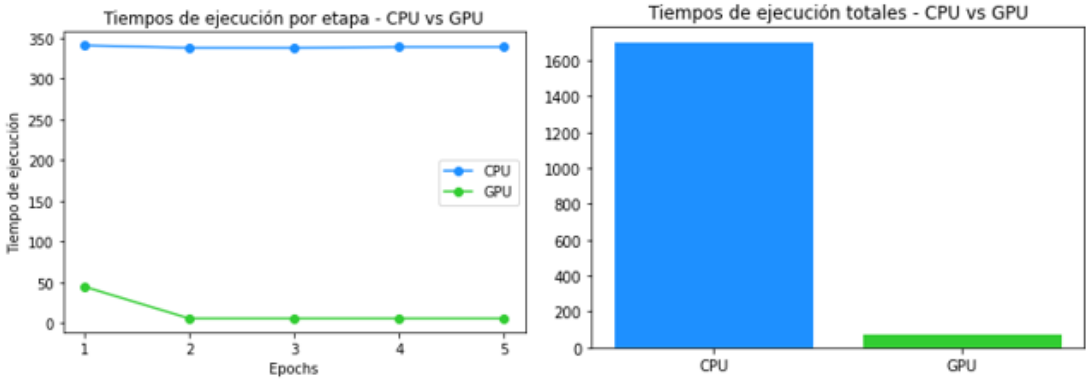


Ilustración 17: Comparación de tiempos en experimento 2

Si nos fijamos en el gráfico de la izquierda, observamos que los tiempos con CPU rondan los 340 segundos, sin embargo, los tiempos con GPU vuelven a ser mucho más reducidos, salvo en el caso de la primera etapa en la que tarda 45 segundos, el resto de las etapas son 6 segundos.

Si nos fijamos en el gráfico de la derecha y comparamos los tiempos totales, vemos que la CPU ha tardado 1702.3 segundos mientras que la arquitectura con GPU ha tardado 71.4 segundos, un tiempo notablemente inferior, en concreto, el tiempo con GPU representa un 4.19% del tiempo de la CPU.

3.5 Experimento 3

3.5.1 Configuración de modelo y resultados

En esta prueba, vamos a extraer 64 filtros en la primera capa de convolución y 32 en la segunda, en ambos casos serán filtros de 5x5. Además, en lugar de 5 etapas de entrenamiento, estableceremos 10 etapas.

En este caso, la topología de la red quedaría de la siguiente manera:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 64)      4864
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 64)      0
_________________________________________________________________
Capa_conv_2 (Conv2D)         (None, 122, 122, 32)      51232
_________________________________________________________________
Capa_Pool_2 (MaxPooling2D)   (None, 61, 61, 32)        0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 119072)            0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                2738679
=================================================================
Total params: 2,794,775
Trainable params: 2,794,775
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 18: Topología de red neuronal experimento 3

Obtenemos una precisión durante el entrenamiento del 59.01% y un F1-Score del 59.13%, pudiendo ver en la siguiente gráfica su evolución a lo largo del entrenamiento:

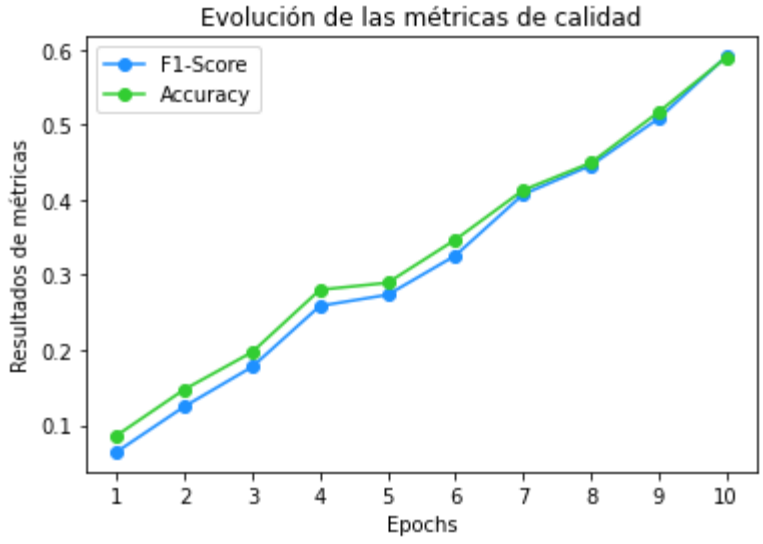


Ilustración 19: Evolución de métricas de calidad en experimento 3

En cuanto a la pérdida, obtenemos 1.55. Podemos ver la evolución resumida en la siguiente gráfica:

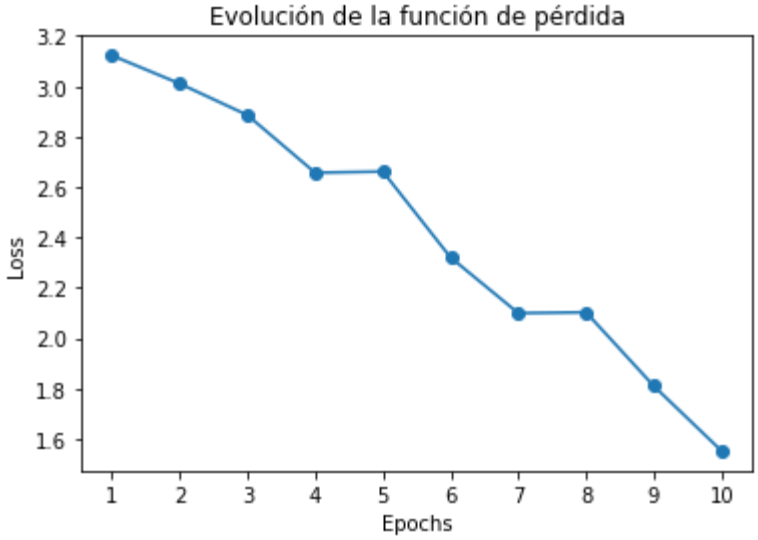


Ilustración 20: Evolución de pérdida en experimento 3

Con los datos de prueba en este caso obtenemos un 47.06% de precisión, un F1-Score de 44.43% y una pérdida de 2.15, en este caso vemos como obtenemos una mayor precisión con datos que no han sido vistos previamente con respecto a los anteriores experimentos.

3.5.2 Ejecución con CPU vs Ejecución con GPU

En el caso del tercer experimento, podemos observar los resultados de los tiempos en los siguientes gráficos:

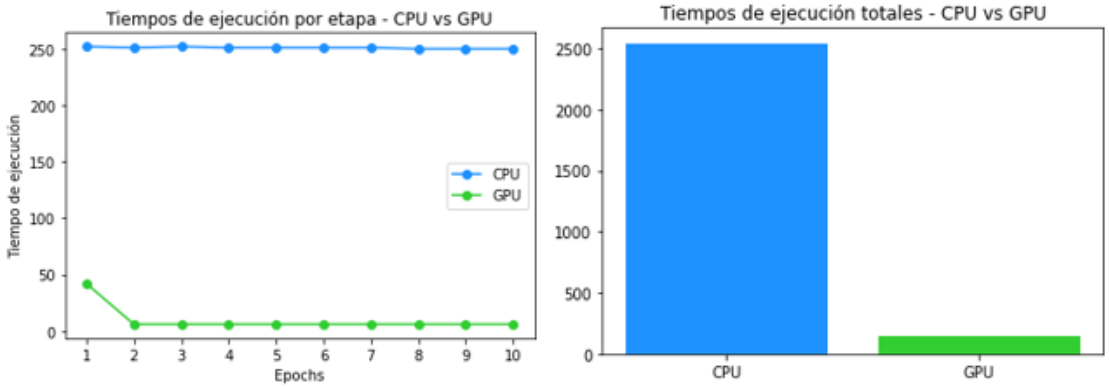


Ilustración 21: Comparación de tiempos en experimento 3

En este experimento nos encontramos que, en el caso de la arquitectura basada en CPU, tenemos tiempos por etapa de 250 segundos mientras que, en el caso de la arquitectura basada en GPU, tenemos en la primera etapa un tiempo algo más elevado, 42 segundos, y en el resto de las etapas un tiempo de 6 segundos.

Si nos fijamos en los tiempos totales, en el caso de la ejecución con CPU obtenemos un tiempo de 2542.58 segundos y en el caso de la ejecución con GPU un tiempo de 142.93

segundos, es decir, el tiempo con GPU representa únicamente un 5.62% del tiempo que tarda la ejecución únicamente con CPU.

3.6 Experimento 4

3.6.1 Configuración de modelo y resultados

En este caso, extraeremos en cada capa de convolución 32 filtros de tamaño 5x5 y continuaremos con las 10 etapas de entrenamiento hasta que mencionemos lo contrario.

El resumen de la topología y los parámetros a aprender sería el siguiente:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 32)      2432
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 32)      0
_________________________________________________________________
Capa_conv_2 (Conv2D)         (None, 122, 122, 32)      25632
_________________________________________________________________
Capa_Pool_2 (MaxPooling2D)   (None, 61, 61, 32)        0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 119072)            0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                2738679
=================================================================
Total params: 2,766,743
Trainable params: 2,766,743
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 22: Topología de red neuronal experimento 4

El resultado obtenido para la precisión durante el entrenamiento es del 51.79% y para la medida F1-Score del 51.78%. En la siguiente gráfica se muestra la evolución de las métricas durante el entrenamiento:

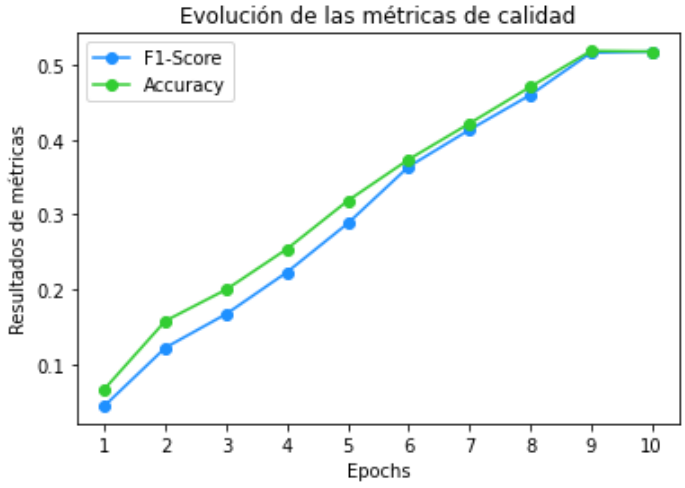


Ilustración 23: Evolución de métricas de calidad en experimento 4

El valor obtenido en el entrenamiento para la función de pérdida es de 1.73, pudiendo ver un resumen de la evolución durante el entrenamiento en el siguiente gráfico:

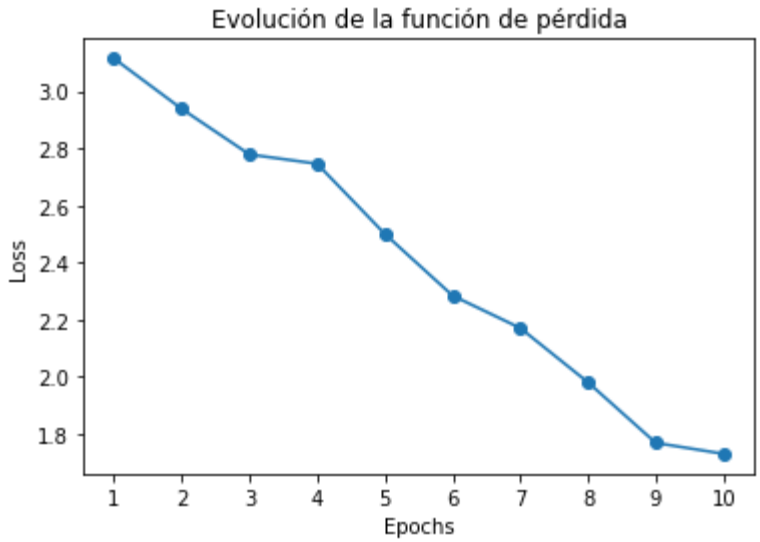


Ilustración 24: Evolución de pérdida en experimento 4

Durante la evaluación del modelo con los datos de validación obtenemos una precisión del 47.06%, un F1-Score del 44.72% y una pérdida de 1.91.

3.6.2 Ejecución con CPU vs Ejecución con GPU

En el caso de nuestro cuarto experimento, podemos observar los resultados de los tiempos en los siguientes gráficos:

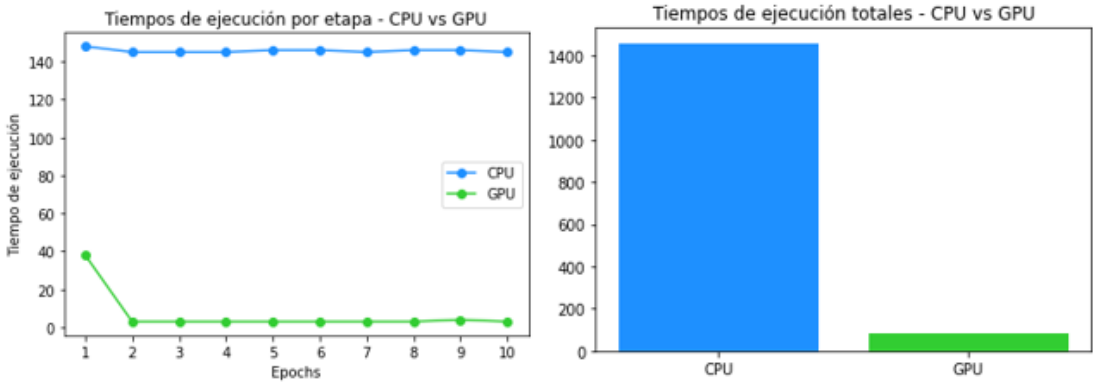


Ilustración 25: Comparación de tiempos en experimento 4

Para este experimento nos encontramos en la arquitectura de CPU tiempos por etapa de aproximádamente 145 segundos, algo más bajos que en los anteriores experimentos ya que aquí extraemos menos filtros en cada capa, y en la arquitectura basada en GPU nos encontramos en la primera etapa con un tiempo de 38 segundos y en el resto de 3 segundos.

Si comparamos los tiempos totales, nos encontramos que la arquitectura basada en CPU tarda en la fase de entrenamiento un total de 1457.12 segundos y la arquitectura basada en GPU un total de 82.78 segundos, lo que es un 5.68% del tiempo total empleado por la CPU.

3.7 Experimento 5

3.7.1 Configuración de modelo y resultados

Para esta prueba estableceremos en 32 el número de filtros a extraer en la primera capa de convolución y 64 filtros en la segunda. En este caso la topología y los parámetros a aprender serían los siguientes:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 32)      2432
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 32)      0
_________________________________________________________________
Capa_conv_2 (Conv2D)         (None, 122, 122, 64)      51264
_________________________________________________________________
Capa_Pool_2 (MaxPooling2D)   (None, 61, 61, 64)        0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 238144)            0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                5477335
=================================================================
Total params: 5,531,031
Trainable params: 5,531,031
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 26: Topología de red neuronal experimento 5

Obtenemos durante el entrenamiento una precisión del 56.21% y una F1-Score del 56.33%, podemos ver la evolución de las métricas durante el entrenamiento en la siguiente gráfica:

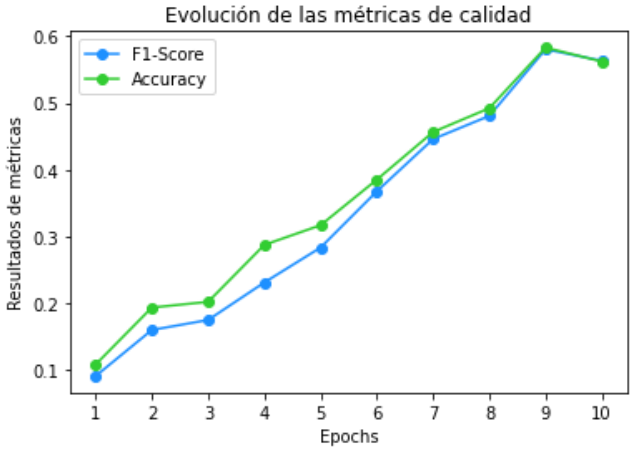


Ilustración 27: Evolución de métricas de calidad en experimento 5

El valor obtenido para la pérdida en la fase de entrenamiento es 1.61, en la siguiente gráfica vemos un resumen de la evolución durante el entrenamiento:

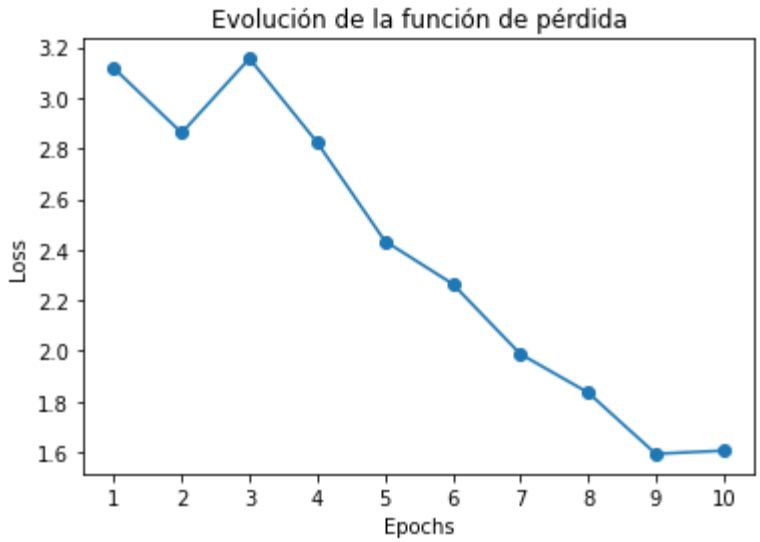


Ilustración 28: Evolución de pérdida en experimento 5

Con los datos de prueba obtenemos una precisión del 46.13%, un F1-Score del 45.22% y una pérdida del 1.8953, los cuales siguen sin ser aun buenos resultados.

3.7.2 Ejecución con CPU vs Ejecución con GPU

En el caso del quinto experimento, podemos observar los resultados de los tiempos en los siguientes gráficos:

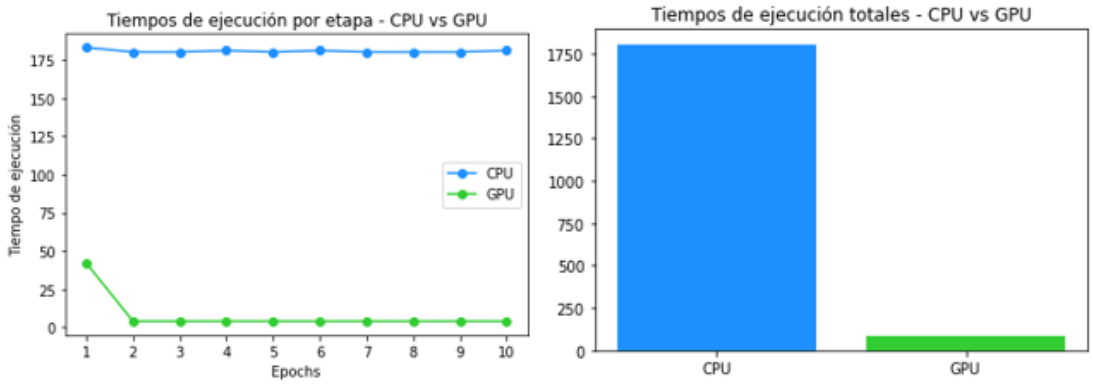


Ilustración 29: Comparación de tiempos en experimento 5

En este caso, si nos fijamos en el tiempo que ha tomado la arquitectura basada en CPU por cada etapa, vemos que dichos tiempos se encuentran alrededor de los 180 segundos mientras que, los tiempos empleados por la arquitectura basada en GPU, se encuentran en torno a los 4 segundos, salvo en la primera etapa que son 46 segundos.

En el caso de los tiempos totales de entrenamiento, la arquitectura basada en CPU emplea un total de 1805.85 segundos mientras que la arquitectura basada en GPU tarda

un total de 82.78 segundos, es decir, el tiempo de la GPU es únicamente un 4.58% del tiempo de la CPU.

3.8 Experimento 6

3.8.1 Configuración de modelo y resultados

En este caso extraeremos en cada capa de convolución 64 filtros de tamaño 5x5, quedando la topología y el número de parámetros a aprender como se muestra a continuación:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 64)      4864
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 64)      0
_________________________________________________________________
Capa_conv_2 (Conv2D)         (None, 122, 122, 64)      102464
_________________________________________________________________
Capa_Pool_2 (MaxPooling2D)   (None, 61, 61, 64)        0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 238144)            0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                5477335
=================================================================
Total params: 5,584,663
Trainable params: 5,584,663
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 30: Topología de red neuronal experimento 6

Obtenemos una precisión del 61.80% y una F1-Score del 61.05%, podemos ver la evolución de las métricas en la siguiente gráfica:

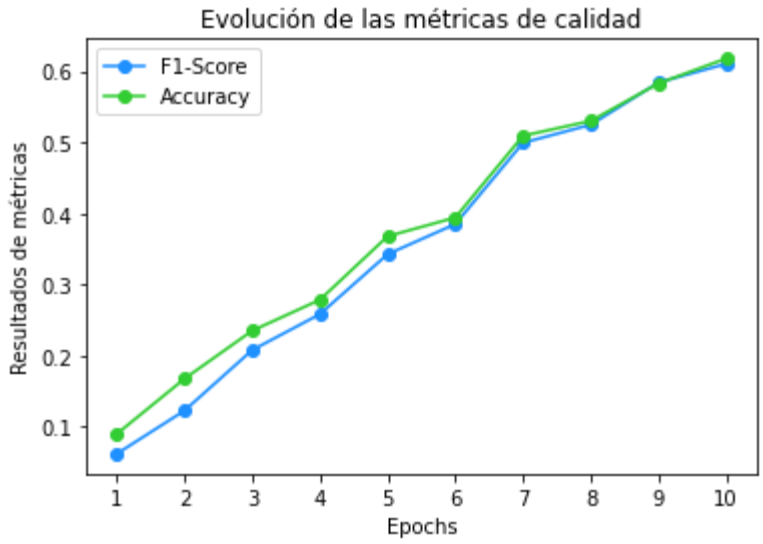


Ilustración 31: Evolución de métricas de calidad en experimento 6

En cuanto al valor de pérdida obtenido, nos devuelve el valor 1.44. Podemos ver un resumen de la evolución a lo largo del entrenamiento en el siguiente gráfico:

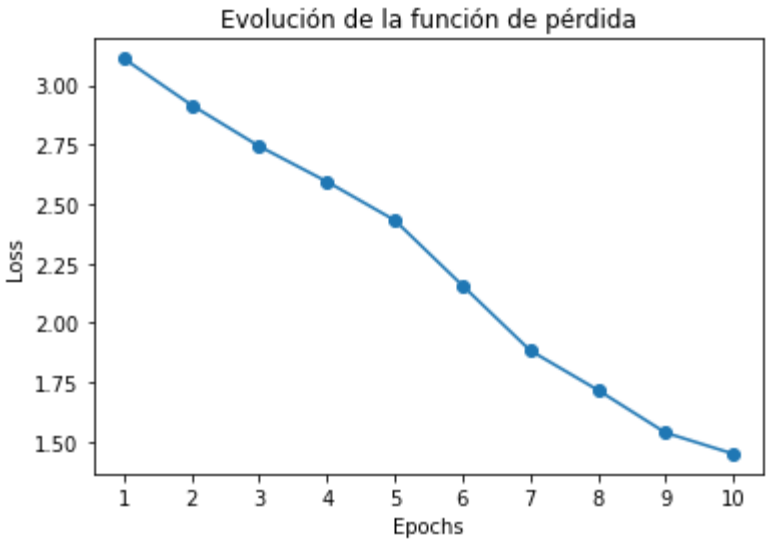


Ilustración 32: Evolución de pérdida en experimento 6

Cuando procedemos a evaluar nuestro modelo con los datos de validación, obtenemos una precisión del 46.44%, un F1-Score del 43.83% y una pérdida de 1.81.

Este resultado aun no es lo que esperamos, hasta que no alcancemos un valor superior al 50% de precisión con los datos de validación, no consideraremos el modelo como resultado aceptable.

3.8.2 Ejecución con CPU vs Ejecución con GPU

En el caso del sexto experimento, podemos observar los resultados de los tiempos en los siguientes gráficos:

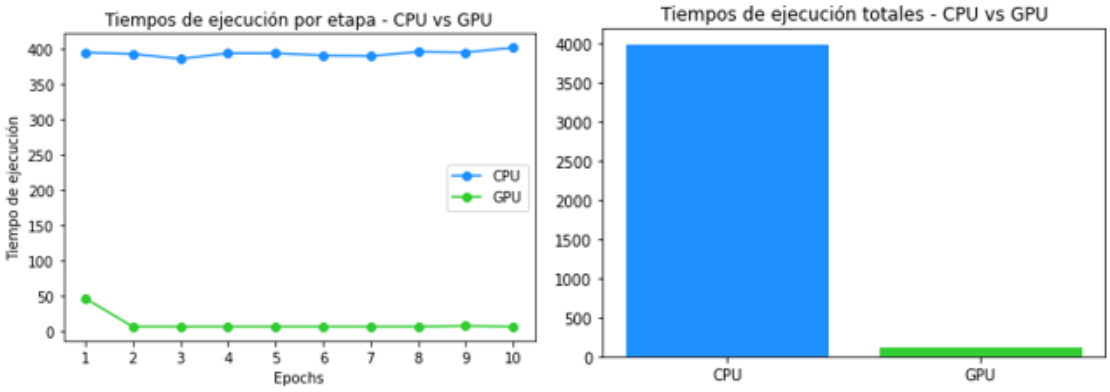


Ilustración 33: Comparación de tiempos en experimento 6

En este caso vemos que los tiempos de cada etapa de entrenamiento en la arquitectura de CPU son alrededor de 395 segundos mientras que para la arquitectura basada en

GPU obtenemos tiempos para cada etapa de 6 segundos, salvo en la primera que son 46 segundos.

Si comparamos los tiempos totales vemos que a la arquitectura basada en CPU el entrenamiento le ha llevado un total de 3982.98 segundos, mientras que a la arquitectura basada en GPU le ha tomado solo 104.14 segundos, es decir, solo un 2.61% del tiempo que le ha llevado a la CPU.

3.9 Comparión de tiempos de ejecución CPU vs GPU

Tras ejecutar los experimentos con la GPU y la CPU, vemos que los tiempos de ejecución evolucionan de manera positiva al ejecutar nuestro modelo con una arquitectura basada en GPU, obteniendo tiempos de ejecución similares y muy bajos al emplear dicha arquitectura.

Debemos anotar que, como es lógico, cuantos más filtros queramos extraer o mayor sea nuestro modelo, mayor número de parámetros deberá aprender nuestra red neuronal, lo que se traduce en un mayor coste computacional durante el entrenamiento y un mayor tiempo dedicado a dicho entrenamiento.

Un ejemplo muy destacable es el caso del experimento 6, el modelo empleado ha tardado un total de 66 minutos con la arquitectura basada en CPU, y con la arquitectura basada en GPU únicamente ha dedicado poco más de un minuto al entrenamiento.

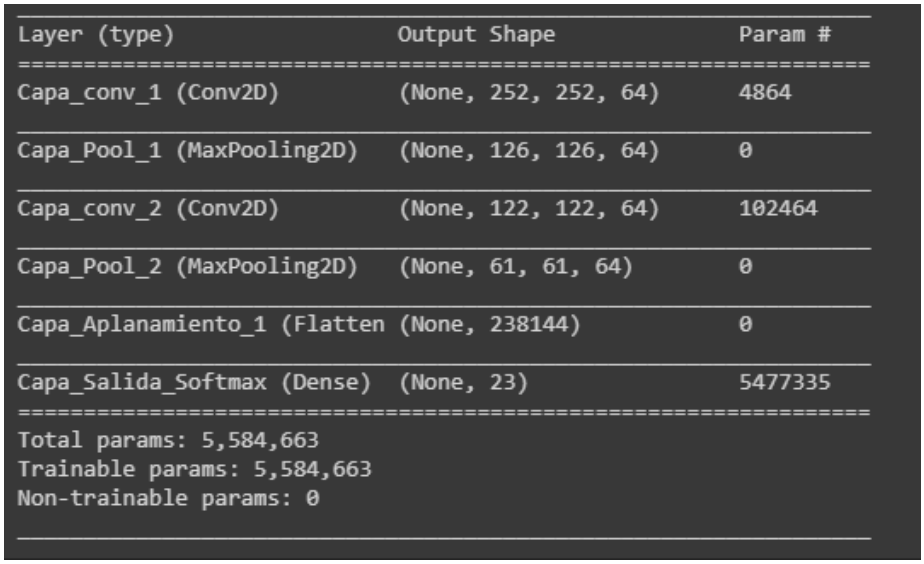
Cabe mencionar que el tiempo también es un factor muy importante a tener en cuenta a la hora de decantarnos por un modelo u otro, por ejemplo, otros experimentos como el 4 y el 5, en sus ejecuciones con CPU, han alcanzado valores en sus métricas de calidad similares a los analizados en el experimento 6, también en su ejecución con CPU, pero han tardado menos de la mitad del tiempo en entrenarse, por lo que podríamos decantarnos por ellos y utilizar este criterio en función de si los valores de las medidas de calidad del modelo son similares.

Por todo ello, dado que hemos visto el gran descenso del tiempo de entrenamiento empleando GPU, en los siguientes experimentos continuaremos con la arquitectura basada en GPU de manera más que justificada.

3.10 Experimento 7

3.10.1 Configuración de modelo y resultados

En este experimento, y en el resto de aquí en adelante salvo que se mencione lo contrario, continuaremos con nuestra capa cuya topología tenía dos capas de convolución de 64 filtros cada una y dos capas de *pooling*. A continuación, dejamos el resumen de la topología para mayor claridad del lector:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Salida_Softmax (Dense)	(None, 23)	5477335

Total params: 5,584,663
Trainable params: 5,584,663
Non-trainable params: 0

Ilustración 34: Topología de red neuronal experimento 7

Hemos observado que, tras diferentes ejecuciones del mismo modelo con la misma configuración de hiperparámetros, en ocasiones obtenemos diferentes resultados en las métricas de calidad, esto en cierto modo tiene sentido debido a la inicialización aleatoria de los pesos de nuestra red.

Con el objetivo de intentar que en cada ejecución del mismo modelo obtengamos resultados similares, vamos a incrementar las etapas de aprendizaje de nuestro modelo a 40.

Para evitar excedernos en las etapas de entrenamiento, vamos a incorporar a nuestro modelo lo que se denomina un *callback*. Un *callback* de Keras [18] es una clase que posee un conjunto de métodos llamados en varias etapas de entrenamiento, pruebas y predicciones. Pueden ser de gran utilidad para tener visibilidad de los estados internos y las estadísticas del modelo durante el entrenamiento.

Con el fin de no excedernos en las etapas de entrenamiento emplearemos un *callback* denominado *EarlyStopping*. Con este *callback* le indicaremos a nuestro modelo que se fije en el resultado de la función de pérdida durante cada etapa del entrenamiento y, en caso de que no mejore en un número definido de etapas, detendrá el entrenamiento para que no empeore nuestro modelo entrenándolo más etapas de la cuenta. En este caso, vamos a definir un total de 6 etapas para que el *callback* tenga como “paciencia” durante el entrenamiento.

Tras entrenar nuestro modelo obtenemos una precisión del 99.92% y un F1-Score del 99.92%. Podemos ver la evolución de las métricas durante el entrenamiento en la siguiente gráfica:

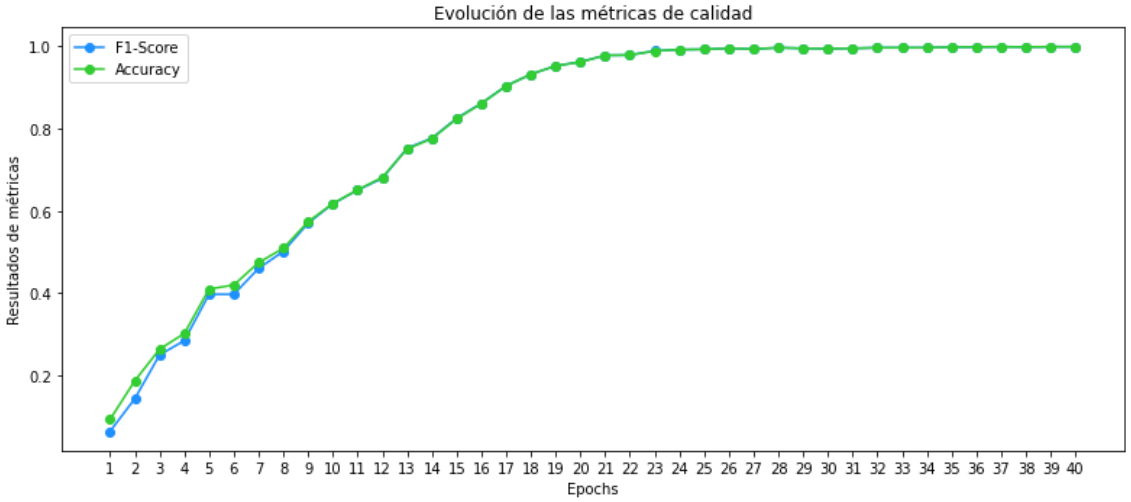


Ilustración 35: Evolución de métricas de calidad en experimento 7

En cuanto a la función de pérdida, obtenemos un 0.0148 tras el entrenamiento. En la siguiente gráfica podemos observar la evolución de la función de perdida en cada etapa a lo largo del entrenamiento:

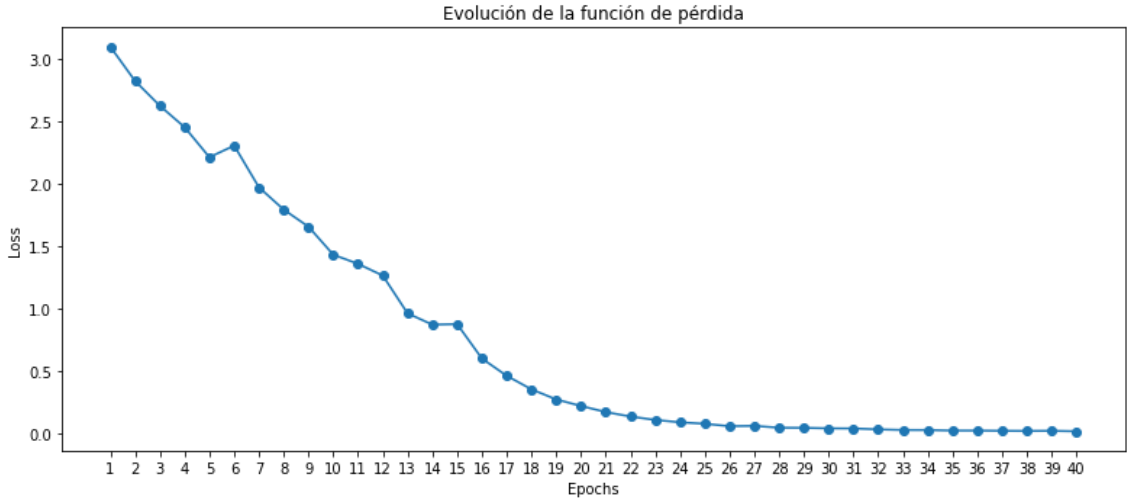


Ilustración 36: Evolución de pérdida en experimento 7

Cuando probamos nuestro modelo con los datos de validación obtenemos una precisión del 56.35% y un F1-Score del 57.64%, en este caso ya nos encontramos por encima del 50% de precisión, es decir, nuestro modelo ya es mejor que un clasificador trivial.

También podemos observar que existe una amplia diferencia, cerca de un 40%, entre la F1-Score obtenida durante la fase de entrenamiento y la obtenida en la fase de validación. Esto nos indica que nuestro modelo muestra un cierto nivel de sobreajuste.

Para comprobar si la configuración de 40 etapas era adecuada o nos faltaban más etapas para que el modelo pudiera aprender lo máximo posible, hemos realizado tres pruebas más con el mismo modelo, pero en este caso hemos establecido 60 etapas para cada prueba, los resultados obtenidos son los siguientes:

- **Prueba 1**: Nuestro modelo detiene el entrenamiento en la etapa 55 debido a que no sufre mejoras, obteniendo el valor mínimo de pérdida en la etapa 49. En este caso los resultados obtenidos con los datos de validación son de una pérdida de 1.99, una precisión del 57.28% y una F1-Score del 58.44%. Como podemos observar, se mejoran ligeramente los resultados del modelo.

- **Prueba 2**: En este caso se completan las 60 etapas, obteniendo el valor mínimo de pérdida en la etapa 60, pero, si nos fijamos en los resultados obtenidos con los datos de validación, podemos observar que en este caso tenemos una pérdida de 2.04, una precisión del 57.28% y una F1-Score del 57.55%.

- **Prueba 3**: En este caso, también se completan las 60 etapas, pero el valor mínimo de pérdida lo obtenemos en la etapa 59. Los resultados obtenidos en este caso son de una precisión del 56.35%, una F1-Score del 56.93% y una pérdida de 2.02.

Como hemos visto, los mejores resultados los hemos obtenido con 40 y con 49 etapas por ello, vamos a establecer las etapas en 50 de aquí en adelante, salvo algún cambio.

También podemos observar en este experimento que hemos cumplido dos de los objetivos que teníamos propuestos, por un lado, ya hemos superado el umbral del 50% en la precisión y F1-Score de nuestro modelo, por lo que ya es mejor que un clasificador trivial, y, por otro lado, aumentando el número de etapas de entrenamiento de nuestro modelo hemos conseguido obtener en cada ejecución del modelo resultados ligeramente diferentes, pero mucho más homogéneos que en los experimentos anteriores.

3.11 Experimento 8

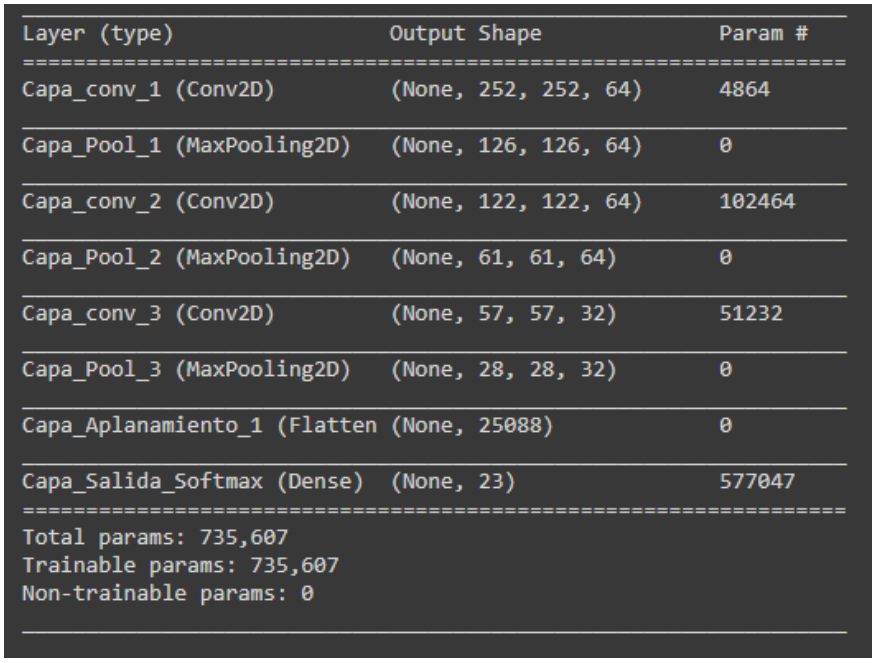
3.11.1 Configuración de modelo y resultados

Antes de continuar integrando mejoras en los hiperparámetros para aumentar la eficacia de nuestro modelo y reducir el sobreajuste, vamos a tratar de introducir una tercera capa de convolución y otra de *pooling*.

Haremos dos pruebas con la capa de convolución, primero probaremos a extraer 32 filtros de 5x5 y después 64 filtros 5x5, la función de activación en ambas será ReLU.

En la capa de *pooling* emplearemos una ventana de 2x2.

Comenzamos observando los resultados utilizando la capa de convolución en la que extraemos 32 filtros. En este caso, nos encontramos con la siguiente topología y número de parámetros:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 32)	51232
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 32)	0
Capa_Aplanamiento_1 (Flatten	(None, 25088)	0
Capa_Salida_Softmax (Dense)	(None, 23)	577047

Total params: 735,607
Trainable params: 735,607
Non-trainable params: 0

Ilustración 37: Topología 1 de red neuronal experimento 8

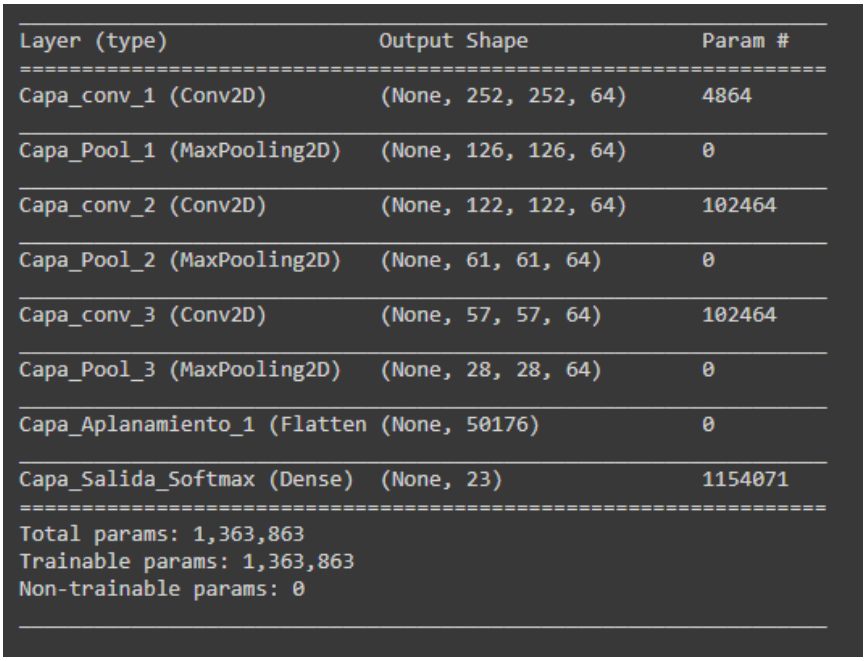
Para estas pruebas, dado que hemos añadido más capas, hemos decidido establecer 80 etapas de aprendizaje junto con el *EarlyStopping* definido en el anterior experimento con una "paciencia" de 6 etapas. Los resultados obtenidos durante las pruebas son los siguientes:

- **Prueba 1**: En esta primera prueba nuestro entrenamiento se detiene en la etapa 77 con una pérdida en entrenamiento de 0.0115, una precisión del 99.84% y una F1-Score del 99.84%.
  Una vez lo sometemos a los datos de validación obtenemos una precisión del 55.73% y una F1-Score del 57.01%, valores que no están mal, pero, si nos fijamos en el valor de la función de pérdida, observamos que obtenemos un valor de 3.46, un valor demasiado elevado.

- **Prueba 2**: En este caso, nos encontramos con que el entrenamiento se detiene en la etapa 35, donde únicamente obtiene una precisión del 66.07% y una F1-Score del 65.53%, con una pérdida de 1.3893.
  Con los datos de validación obtenemos un valor para la función de pérdida de 2.229, una precisión del 42.11% y una F1-Score del 43.75%.
  Como vemos, estos resultados no son muy buenos en comparación con lo que teníamos.
- **Prueba 3**: En esta tercera prueba, nos encontramos con que la fase de entrenamiento finaliza en la etapa 79 con una pérdida de 0.0101, una precisión del 99.84% y una F1-Score del 99.84%.
  Cuando ejecutamos el modelo con los datos de validación nos encontramos con una precisión del 53.87% y una F1-Score del 54.79%, valores que son mejores que un clasificador trivial, pero, como nos pasaba en la primera prueba, el resultado de la función de pérdida es de 3.55 el cual, es muy elevado.

En principio, esta primera arquitectura con una tercera capa de convolución de 32 filtros y una tercera capa de *pooling* no nos termina de convencer.

Vamos a realizar los experimentos ahora con una tercera capa de convolución en la que extraeremos 64 filtros y una tercera capa de *pooling*. Podemos ver la topología y el número de parámetros en la siguiente imagen:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 64)	102464
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 50176)	0
Capa_Salida_Softmax (Dense)	(None, 23)	1154071

Total params: 1,363,863
Trainable params: 1,363,863
Non-trainable params: 0

Ilustración 38: Topología 2 de red neuronal experimento 8

Para estas pruebas también emplearemos 80 etapas y un *EarlyStopping* con una "paciencia" de 6 etapas. Los resultados obtenidos durante las pruebas son los siguientes:

- **Prueba 1**: En esta primera prueba, el entrenamiento se detiene en la etapa 44 con una pérdida de 1.55, una precisión del 64.75% y una F1-Score del 64.69%. Cuando aplicamos nuestro modelo sobre los datos de validación nos encontramos con unos resultados poco favorables. En concreto, obtenemos un valor de pérdida de 2.003, una precisión del 43.65% y una F1-Score del 42.35%. Como podemos observar, nos encontramos con un modelo con una calidad por debajo del 50%.

- **Prueba 2**: En esta prueba el entrenamiento se detiene en la etapa 59 con una pérdida de 0.0124, una precisión del 99.84% y una F1-Score del 99.84%.
  Con los datos de validación nos encontramos unas métricas que pudieran ser aceptables, en este caso una precisión del 54.49% y una F1-Score del 55.58%, pero obtenemos una pérdida de 3.56, un valor que es demasiado elevado.

- **Prueba 3**: En esta última prueba, nos encontramos con que el modelo finaliza en la etapa 79 con una precisión del 99.84%, una F1-Score del 99.84% y una pérdida del 0.0105.
  Cuando enfrentamos al modelo a los datos de validación, nos encontramos unos resultados que siguen sin ser demasiado favorables. En concreto, obtenemos una precisión del 53.25%, una F1-Score del 54.50% y una pérdida del 3.15, la cual sigue siendo elevada.

Tras realizar las pruebas con la tercera capa de convolución de 64 filtros obtenemos resultados poco favorables.

A rasgos generales, los resultados conseguidos añadiendo una tercera capa de convolución, tanto de 32 filtros como de 64 filtros, y una tercera capa de *pooling*, no están a la altura de los obtenidos en el anterior experimento con únicamente dos capas de convolución y de *pooling*, ya que, a pesar de obtener resultados similares en cuanto a las métricas de calidad, los valores obtenidos en la función de pérdida son demasiado altos.

Por ello, decidimos continuar con la topología de nuestra red neuronal empleada en el experimento 7, ya que, no obteniendo mejores resultados y ni siquiera resultados iguales, sería un desperdicio de recursos de cómputo y de tiempo innecesario.

3.12 Experimento 9

3.12.1 Configuración de modelo y resultados

Continuaremos con la topología utilizada en el experimento 7, es decir, emplearemos la red neuronal con dos capas de convolución en las que extraemos 64 filtros respectivamente, y dos capas de *pooling*.

En este experimento vamos a realizar diversas pruebas trabajando con el parámetro *learning rate* de nuestro optimizador "*sgd*".

Los algoritmos de gradiente descendiente multiplican la magnitud del gradiente, que es un vector con una dirección y una magnitud, por un escalar conocido como *learning rate* que determina el siguiente valor del parámetro. [6]

Por ejemplo, si la magnitud del gradiente es de 2.5 y el *learning rate* es de 0.01, entonces nuestro algoritmo de gradiente descendiente seleccionará el siguiente punto a 0.025 de distancia del punto previo.

Para estas pruebas modificaremos el valor de nuestro *learning rate* con los siguientes valores: 0.0085, 0.009, 0.0095, 0.01, 0.0105, 0.011 y 0.0115. Debemos anotar que, por defecto, el valor del optimizador "*sgd*" es de 0.01.

También debemos comentar que estas pruebas las dividiremos en dos, por un lado, hemos realizado las pruebas modificando los valores del *learning rate* empleando 50 etapas para el entrenamiento y, por otro lado, hemos realizado estas mismas variaciones del *learning rate* pero con 60 etapas por si fueran necesarias más etapas para aprender de manera más adecuada al modificar este valor.

A continuación, vamos a mostrar los resultados obtenidos durante la ejecución de 50 etapas de entrenamiento:

- ***Learning rate* de 0.0085**: En la fase de entrenamiento obtenemos una pérdida de 0.0126, una precisión del 99.92% y una F1-Score del 99.92%.

Durante la fase de validación obtenemos una pérdida de 2.00, una precisión del 56.35% y una F1-Score del 56.90%.

- ***Learning rate* de 0.009**: En la fase de entrenamiento obtenemos una pérdida de 0.012, una precisión del 99.92% y una F1-Score del 99.92%.
  Durante la fase de validación obtenemos una pérdida 1.9705, una precisión del 56.66% y una F1-Score del 57.32%.

- ***Learning rate* de 0.0095**: En la fase de entrenamiento obtenemos una pérdida de 0.0107, una precisión del 99.92% y una F1-Score del 99.92%.
  En la fase de validación conseguimos una valor de pérdida de 2.0087, una precisión del 56.97% y una F1-Score del 58.27%.

- ***Learning rate* de 0.01**: Durante el entrenamiento obtenemos una pérdida de 0.01, una precisión del 99.84% y una F1-Score del 99.84%.
  Cuando enfrentamos al modelo a los datos de validación obtenemos una pérdida de 2.07, una precisión del 57.28% y una F1-Score del 58.26%.

- ***Learning rate* de 0.0105**: Tras el entrenamiento obtenemos una pérdida de 0.0096, una precisión del 99.92% y una F1-Score del 99.92%.
  Durante la fase de validación los resultados obtenidos son de 2.06 para la pérdida, 56.97% para la precisión y un 58.17% para la F1-Score.

- ***Learning rate* de 0.011**: Durante el entrenamiento obtenemos una pérdida de 0.011, una precisión del 99.84% y una F1-Score del 99.92%.
  En la fase de validación nos encontramos con buenos resultados, para la función de pérdida obtenemos un valor de 2.025, una precisión del 57.59% y un valor para la F1-Score del 58.65%.

- ***Learning rate* de 0.0115**: Durante la fase de entrenamiento obtenemos una pérdida de 0.009, una precisión del 99.92% y una F1-Score del 99.92%.
  Durante la fase de validación se obtiene una pérdida de 2.115, una precisión del 57.28% y una F1-Score del 57.79%.

A continuación, mostraremos los resultados obtenidos durante la variación del *learning rate* con un entrenamiento de 60 etapas:

- ***Learning rate* de 0.0085**: En la fase de entrenamiento obtenemos una pérdida de 0.0093, una precisión del 99.92% y una F1-Score del 99.92%.
  En la fase de entrenamiento obtenemos una pérdida de 2.196, una precisión del 55.73% y una F1-Score del 56.75%.

- ***Learning rate* de 0.009**: Para la fase de entrenamiento obtenemos una pérdida de 0.0108, una precisión del 99.84% y una F1-Score del 99.84%.
  En la fase de validación obtenemos un valor de pérdida de 2.0628, una precisión del 56.35% y una F1-Score del 57.18%.

- ***Learning rate* de 0.0095**: Durante la fase de entrenamiento obtenemos una pérdida de 0.0069, una precisión del 99.92% y una F1-Score del 99.92%.

En la fase de validación obtenemos un valor para la función de pérdida de 2.248, una precisión del 53.25% y una F1-Score del 54.60%.

- ***Learning rate* de 0.01**: En la fase de entrenamiento obtenemos una pérdida de 0.0076, una precisión del 99.92% y una F1-Score del 99.92%.
  Durante la fase de validación obtenemos una pérdida de 2.223, una precisión del 56.04% y una F1-Score del 56.68%.

- ***Learning rate* de 0.0105**: Durante el entrenamiento obtenemos una pérdida de 0.0081, una precisión del 99.92% y una F1-Score del 99.92%.
  En la fase de validación obtenemos una pérdida de 2.066, una precisión del 57.89% y una F1-Score del 58.33%

- ***Learning rate* de 0.011**: En la fase de entrenamiento obtenemos una pérdida de 0.0077, una precisión del 99.92% y una F1-Score del 99.92%.
  En la fase de validación obtenemos una pérdida de 2.246, una precisión del 54.80% y una F1-Score del 56.05%.

- ***Learning rate* de 0.0115**: Durante la fase de entrenamiento obtenemos una pérdida de 0.0108, una precisión del 99.84% y una F1-Score del 99.84%.
  En la fase de validación obtenemos una pérdida de 2.21, una precisión del 56.66% y una F1-Score del 57.45%.

A rasgos generales, observando los resultados obtenidos durante las pruebas, no hemos generado malos resultados, aunque tampoco hemos mejorado demasiado los valores que teníamos previamente.

En cualquier caso, podemos destacar algunos de los resultados obtenidos: por la parte de las pruebas con 60 etapas, destacamos el *learning rate* de 0.0105 y, para la parte de pruebas de 50 etapas, destacamos los *learning rates* de 0.0095, 0.01 y 0.011.

En los siguientes gráficos presentamos una comparativa de las funciones de pérdida, la precisión y la F1-Score obtenidos con los datos de validación en las pruebas con los 4 *learning rates* destacados:

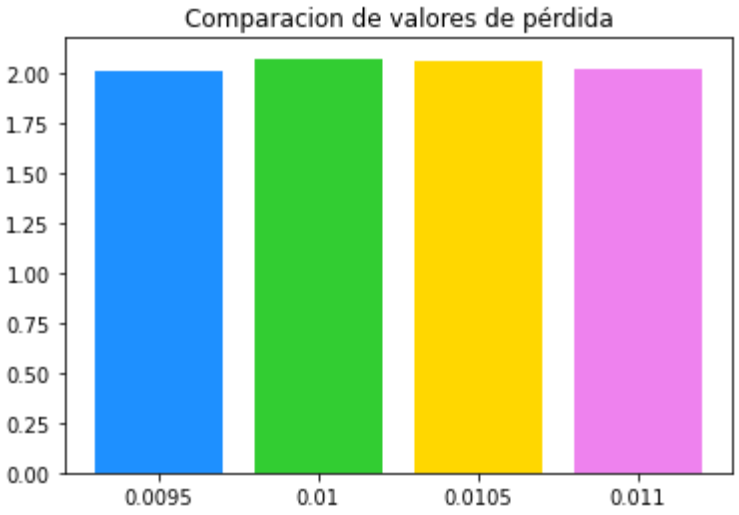


Ilustración 39: Comparación de valores de pérdida experimento 9

Si nos fijamos en los cuatro valores obtenidos en las funciones de pérdida, observamos que todos ellos son muy similares, aunque podríamos destacar los colores azul y violeta que se corresponden con los valores de *learning rate* 0.0095 y 0.011, respectivamente.

En el siguiente gráfico podemos ver una comparativa de los valores obtenidos en la precisión:

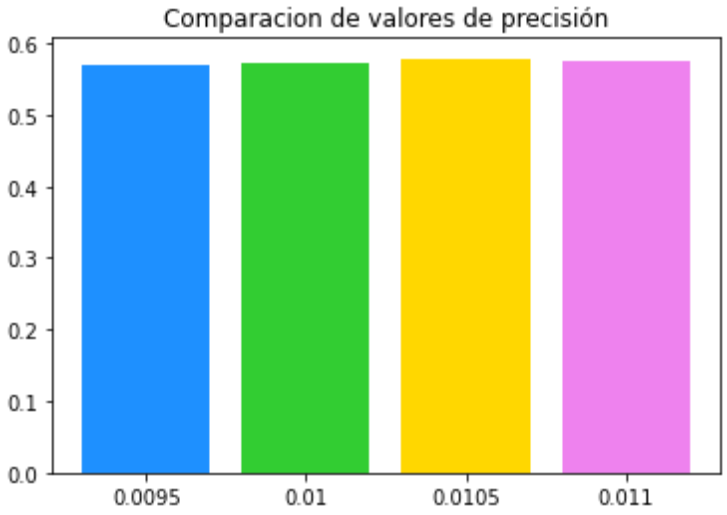


Ilustración 40: Comparación de valores de precisión experimento 9

Como podemos observar, en este caso las precisiones obtenidas son muy similares entre ellas, pero podemos destacar la amarilla, correspondiente al *learning rate* 0.0105, como la más alta con un 57.89% y la violeta, correspondiente al *learning rate* 0.011, como la segunda precisión más alta con un 57.59%.

En el siguiente gráfico encontramos una comparativa de las F1-Score obtenidas durante la etapa de validación:

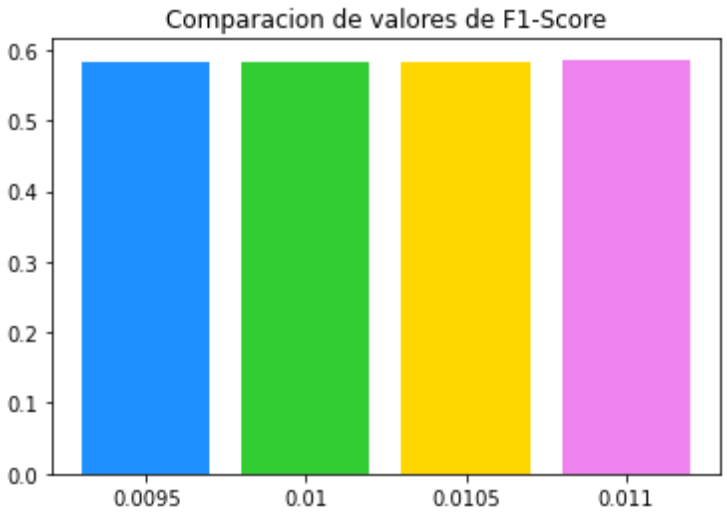


Ilustración 41: Comparación de valores de F1-Score experimento 9

En el caso de los valores para la F1-Score, observamos que prácticamente son iguales los cuatro valores, pero si nos fijamos en la columna violeta, correspondiente al *learning*

rate 0.011, es ligeramente más alta. En concreto, como hemos visto previamente, posee una F1-Score del 58.65%.

En general, hemos visto que todos los resultados obtenidos son buenos y muy similares entre ellos. A pesar de ser similares, hemos visto que existen pequeños matices que nos pueden hacer decantarnos por uno u otro, en concreto, nos decantamos por el *learning rate* de 0.011 ya que, como hemos visto, posee un valor menor en cuanto a la función de pérdida y un valor mayor en cuanto a la F1-Score.

En los siguientes experimentos, salvo que se mencione lo contrario, continuaremos con *learning rate* de 0.011.

3.13 Experimento 10

3.13.1 Configuración de modelo y resultados

En este experimento veremos si somos capaces de mejorar la calidad de nuestro modelo añadiendo un número mayor de filtros a extraer en nuestra segunda capa de convolución.

Con ello, nuestra red neuronal quedaría de la siguiente manera en cuanto a topología y parámetros a extraer:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Capa_conv_1 (Conv2D)         (None, 252, 252, 64)      4864
_________________________________________________________________
Capa_Pool_1 (MaxPooling2D)   (None, 126, 126, 64)      0
_________________________________________________________________
Capa_conv_2 (Conv2D)         (None, 122, 122, 128)     204928
_________________________________________________________________
Capa_Pool_2 (MaxPooling2D)   (None, 61, 61, 128)       0
_________________________________________________________________
Capa_Aplanamiento_1 (Flatten (None, 476288)            0
_________________________________________________________________
Capa_Salida_Softmax (Dense)  (None, 23)                10954647
=================================================================
Total params: 11,164,439
Trainable params: 11,164,439
Non-trainable params: 0
_________________________________________________________________
```

Ilustración 42: Topología de red neuronal experimento 10

Durante la primera prueba vamos a probar nuestra red con 128 filtros en la segunda capa de convolución y con 50 etapas de entrenamiento.

Tras el entrenamiento, obtenemos una precisión del 66.23%, una F1-Score del 64.44% y una pérdida de 1.30.

Cuando sometemos a nuestra red a los datos de validación, obtenemos una pérdida de 1.86, una precisión del 45.51% y una F1-Score del 46.33%.

Podemos ver en la siguiente gráfica la evolución de la precisión y la F1-Score durante el entrenamiento:

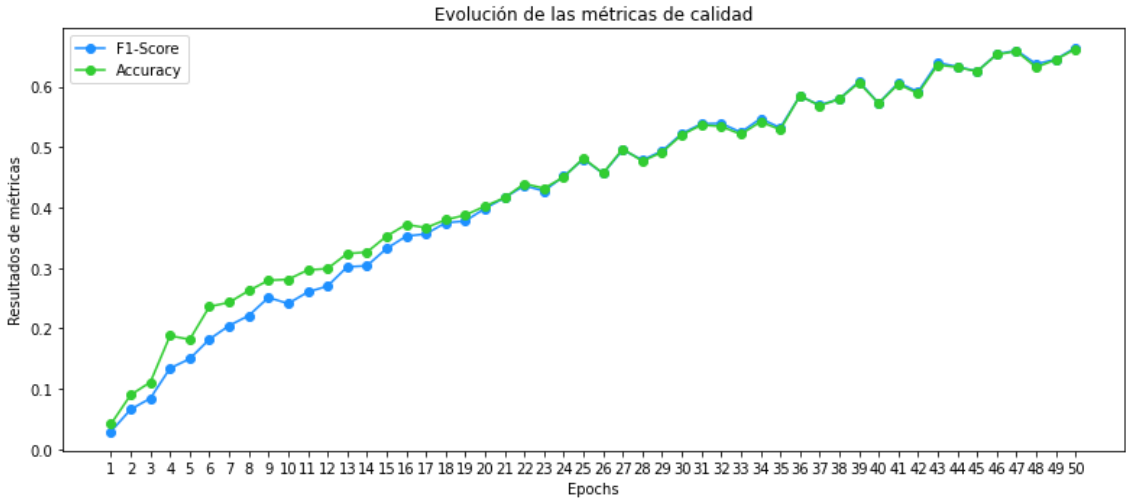


Ilustración 43: Evolución 1 de métricas de calidad en experimento 10

En la siguiente gráfica podemos observar la evolución de la función pérdida durante el entrenamiento:

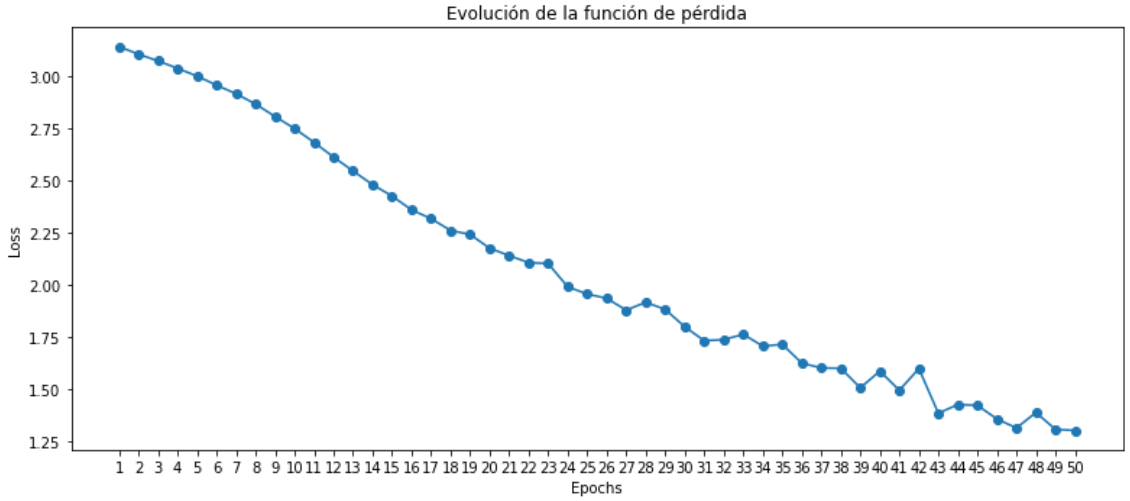


Ilustración 44: Evolución 1 de pérdida en experimento 10

Como podemos observar, no llegamos a un punto en el que se estabilicen los valores, tanto de las métricas de calidad como de la función de pérdida. Esto nos puede llevar a pensar que quizás necesitemos establecer un mayor número de etapas a nuestra red durante el entrenamiento, lo cual tendría sentido ya que, al tener un mayor número de filtros, necesitará más tiempo de aprendizaje.

Además, como hemos visto, los resultados obtenidos con los datos de validación no son para nada aceptables.

En nuestra segunda prueba estableceremos el número de etapas en 120 para darle más tiempo a nuestro modelo de aprender.

Los resultados obtenidos tras el entrenamiento son de un 0.60 de pérdida, del 87.03% de precisión y del 87.23% para la F1-Score.

En cuanto a los datos de validación, los resultados obtenidos son de una precisión del 52.01%, de una F1-Score del 53.33% y una pérdida de 1.79.

Vemos que estos resultados son algo mejores que los obtenidos en la primera prueba, pero no mejoran lo que ya habíamos conseguido en otros experimentos.

Vamos a realizar una tercera prueba en la que estableceremos 200 etapas de entrenamiento con el fin de observar si mejora la calidad del modelo.

En este caso, obtenemos tras el entrenamiento una precisión del 98.45% y una F1-Score del 98.45%, además de una pérdida de 0.13.

En el siguiente gráfico podemos ver la evolución de las métricas de calidad a lo largo del entrenamiento:

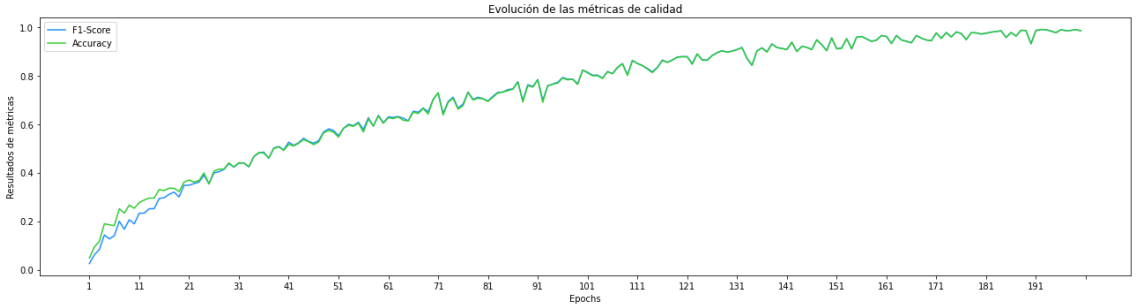


Ilustración 45: Evolución 2 de métricas de calidad en experimento 10

En el siguiente gráfico se puede observar el desarrollo de la función de perdida durante el entrenamiento:

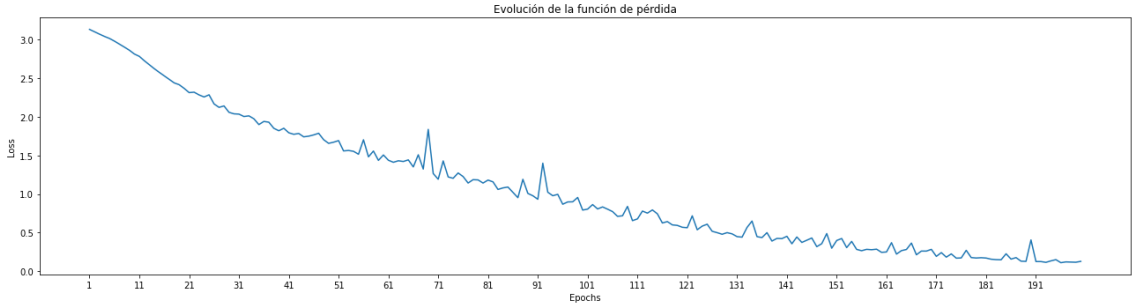


Ilustración 46: Evolución 2 de pérdida en experimento 10

Los resultados obtenidos con los datos de validación son de una precisión del 50.77%, de una F1-Score del 51.43% y de una pérdida de 2.08.

Como vemos, los resultados obtenidos extrayendo más filtros no son demasiado buenos ni mejoran los resultados previos.

Además, como podemos observar en las gráficas, ya no merece la pena añadir más etapas al experimento.

3.14 Experimento 11

3.14.1 Configuración de modelo y resultados

Como hemos visto, nuestro modelo presenta cierto nivel de sobreajuste a los datos de entrenamiento, es decir, no aprende a generalizar todo lo que debería. Esto es muy habitual en redes neuronales cuando el conjunto de datos para entrenar la red no es muy grande, y se produce debido a que en los datos de entrenamiento puede existir cierto ruido que después no estará presente en los datos de validación.

Para eliminar el nivel de sobreajuste, una técnica muy empleada en *deep learning* es añadir capas de *dropout* a nuestro modelo.

Cuando añadimos capas de *dropout* a nuestro modelo, lo que estamos realizando en la práctica es desactivar un porcentaje, previamente establecido, de neuronas aleatorias en nuestra red neuronal. Con esto conseguimos que las neuronas no "memoricen" las entradas que recibe el modelo, sino que entorpecemos el aprendizaje para evitar de este modo el sobreajuste. [19]

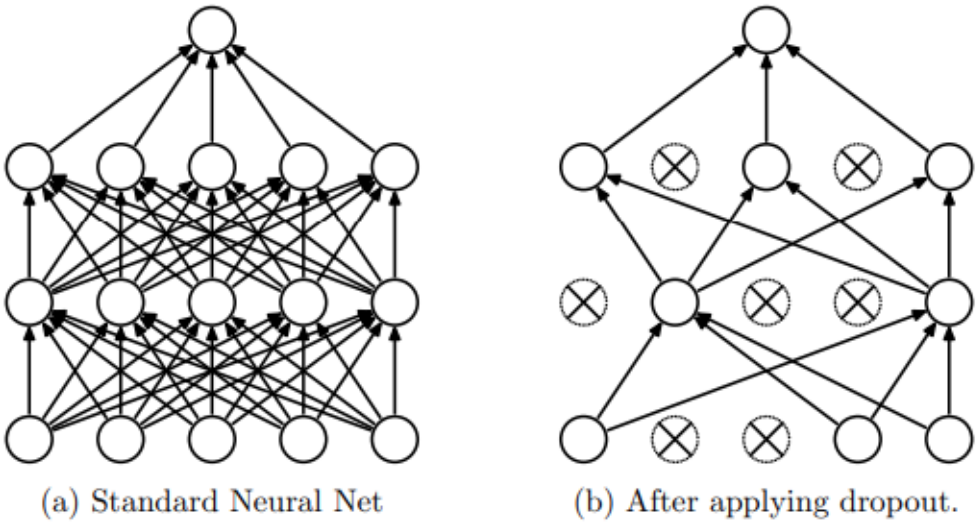
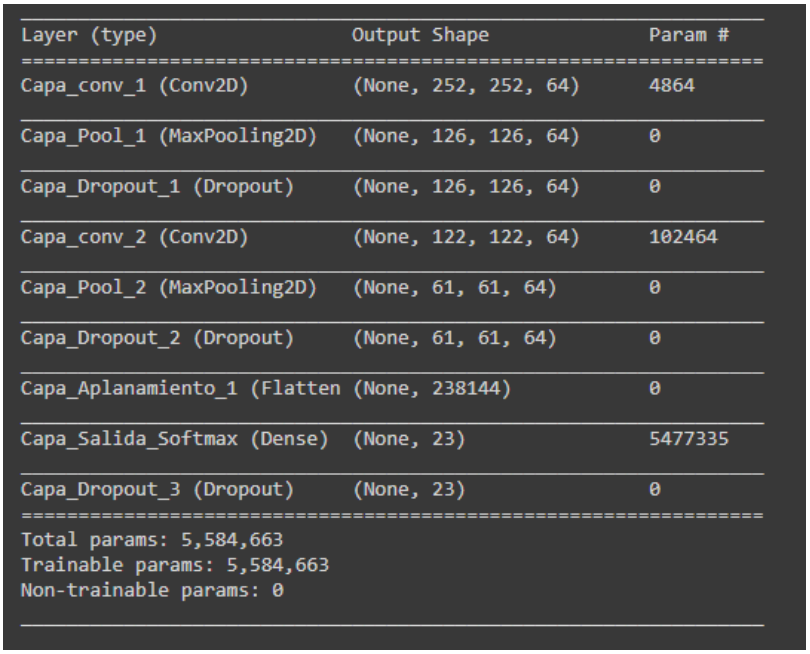


Ilustración 47: Ejemplo de aplicación de Dropout [20]

Para las pruebas en este experimento añadiremos tres capas de *dropout* a las que haremos referencia como "primera capa de *dropout*", "segunda capa de *dropout*" y "tercera capa de *dropout*" para una compresión más sencilla por parte del lector. Las capas siempre estarán ubicadas en la misma posición, en caso de no mencionar ningún valor para dicha capa, querrá decir que esa prueba no emplearemos dicha capa de *dropout*.

A continuación, mostramos una imagen de como quedarían ubicadas las tres capas de *dropout* que mencionábamos:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Salida_Softmax (Dense)	(None, 23)	5477335
Capa_Dropout_3 (Dropout)	(None, 23)	0

Total params: 5,584,663
Trainable params: 5,584,663
Non-trainable params: 0

Ilustración 48: Topología de red neuronal experimento 11

Las pruebas realizadas son las siguientes:

- En la primera prueba únicamente emplearemos una capa de *dropout*, en este caso utilizaremos la tercera capa de *dropout* con un valor de 0.25, lo cual quiere decir que un 25% de las neuronas se desactivaran durante cada etapa del entrenamiento. También estableceremos 50 etapas de entrenamiento y una paciencia en el *early stopping* de 6 etapas.
  Con los datos de entrenamiento obtenemos una precisión del 75.13%, una F1-Score del 74.84% y una pérdida de 3.99.
  Tras la ejecución con los datos de validación, obtenemos una precisión del 57.28%, una F1-Score del 58.22% y una pérdida del 1.73.
  En esta primera prueba vemos como ha descendido el sobreajuste y obtenemos un valor similar en cuanto a las métricas de calidad con respecto a otros experimentos al ejecutar el modelo con los datos de validación.

- En esta prueba utilizaremos únicamente la tercera capa de *dropout* con un valor de 0.25, una paciencia de 9 etapas para el early stopping y establecemos 50 etapas de entrenamiento.
  Tras el entrenamiento, obtenemos una pérdida de 4.49, una precisión del 72.20% y una F1-Score del 72.50%.
  Con los datos de validación nos encontramos con una pérdida de 1.81, una precisión del 56.66% y una F1-Score del 57.12%.

- En esta prueba empleamos la tercera capa de *dropout* con un valor de 0.25, establecemos 50 etapas de entrenamiento y un *early stopping con una paciencia* de 12 etapas.
  En este caso, tras el entrenamiento obtenemos una pérdida de 3.71, una precisión del 76.94% y una F1-Score del 77.17%.
  Tras el paso de los datos de validación obtenemos una precisión del 56.66%, una F1-Score del 57.17% y una pérdida de 1.88.

En este caso obtenemos resultados similares a los anteriores.

- En la siguiente prueba vamos a emplear dos de las tres capas de *dropout*, en concreto la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor también de 0.25. Además, establecemos 50 etapas para el entrenamiento y una paciencia de 9 etapas.
  Tras el entrenamiento obtenemos una pérdida de 4.24, una precisión del 73.60% y una F1-Score del 73.94%.
  Con los datos de validación nos encontramos con un valor para la función de pérdida de 1.73, una precisión del 56.35% y una F1-Score del 56.96%.

- En este caso vamos a emplear dos de las tres capas de *dropout*, en concreto la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor también de 0.35. Además, establecemos 50 etapas para el entrenamiento y una paciencia de 12 etapas.
  Con los datos de entrenamiento obtenemos una pérdida de 5.86, una precisión del 63.66% y una F1-Score del 64.34%.
  Con los datos de validación obtenemos una pérdida de 1.84, una precisión del 58.20% y una F1-Score del 58.99%
  En este caso obtenemos uno de los mejores valores con los datos de validación hasta el momento.

- En esta prueba vamos a emplear dos de las tres capas de *dropout*, en concreto, la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor de 0.4. Además, establecemos 75 etapas para el entrenamiento y una paciencia de 12 etapas.
  Con los datos de entrenamiento obtenemos una pérdida de 6.15, una precisión del 61.26% y una F1-Score del 61.71%.
  Con los datos de validación obtenemos una pérdida de 2.12, una precisión del 52.32% y una F1-Score del 53.66%.
  En este caso, los resultados obtenidos no son demasiado favorables.

- Para esta prueba vamos a utilizar dos de las tres capas de *dropout*, en concreto la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor también de 0.4. Además, establecemos 60 etapas para el entrenamiento y una paciencia de 12 etapas.
  Con los datos de entrenamiento obtenemos una pérdida de 6.11, una precisión del 60.93% y una F1-Score del 61.01%.
  Con los datos de validación obtenemos una pérdida de 2.24, una precisión del 52.13% y una F1-Score del 52.56%.

- En la siguiente prueba vamos a utilizar la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor también de 0.35. Además, establecemos 60 etapas para el entrenamiento y una paciencia de 12 etapas. En este caso, vamos a probar en nuestras capas de convolución filtros de tamaño 3x3 en lugar de los filtros de tamaño 5x5 que estábamos empleando hasta ahora.
  Con los datos de entrenamiento obtenemos una pérdida de 5.68, una precisión del 64.75% y una F1-Score del 65.19%.

Con los datos de validación obtenemos una pérdida de 1.82, una precisión del 55.73% y una F1-Score del 57.11%.

- Para esta prueba vamos a utilizar la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor también de 0.35. En este caso, establecemos 80 etapas para el entrenamiento y una paciencia de 12 etapas. En esta prueba vamos a emplear de nuevo en nuestras capas de convolución filtros de tamaño 3x3.
  Con los datos de entrenamiento obtenemos una pérdida de 5.06, una precisión del 65.14% y una F1-Score del 65.55%.
  Con los datos de validación obtenemos una pérdida de 1.93, una precisión del 54.49% y una F1-Score del 55.54%.

- En la siguiente prueba vamos a utilizar la primera capa de *dropout* con un valor de 0.25 y la segunda capa de *dropout* con un valor también de 0.25. Además, establecemos 120 etapas para el entrenamiento y una paciencia de 15 etapas. De aquí en adelante, salvo que mencionemos lo contrario, volveremos a emplear en las capas de convolución con filtros de tamaño 5x5.
  Con los datos de entrenamiento obtenemos una pérdida de 3.96, una precisión del 75.39% y una F1-Score del 75.84%.
  Con los datos de validación obtenemos una pérdida de 1.96, una precisión del 56.66% y una F1-Score del 56.63%.

- En esta prueba vamos a emplear las tres capas de dropout, en concreto, la primera capa de *dropout* con un valor de 0.25, la segunda capa de *dropout* con un valor también de 0.25 y la tercera capa de *dropout* con un valor de 0.5. Emplearemos 50 etapas de entrenamiento con una paciencia de 6 etapas.
  Con los datos de entrenamiento obtenemos una pérdida de 8.53, una precisión del 34.08% y una F1-Score del 34.66%.
  Con los datos de validación obtenemos una pérdida de 1.97, una precisión del 43.65% y una F1-Score del 42.35%.

- Para la siguiente prueba volvemos a utilizar las tres capas de dropout, en concreto, la primera capa de *dropout* con un valor de 0.25, la segunda capa de *dropout* con un valor también de 0.25 y la tercera capa de *dropout* con un valor de 0.25. Emplearemos 120 etapas de entrenamiento con una paciencia de 15 etapas.
  Con los datos de entrenamiento obtenemos una pérdida de 3.96, una precisión del 75.39% y una F1-Score del 75.84%.
  Con los datos de validación obtenemos una pérdida de 1.96, una precisión del 56.61% y una F1-Score del 56.64%.

A continuación, se muestran los resultados obtenidos en estas pruebas y el umbral a superar del 58.65%:

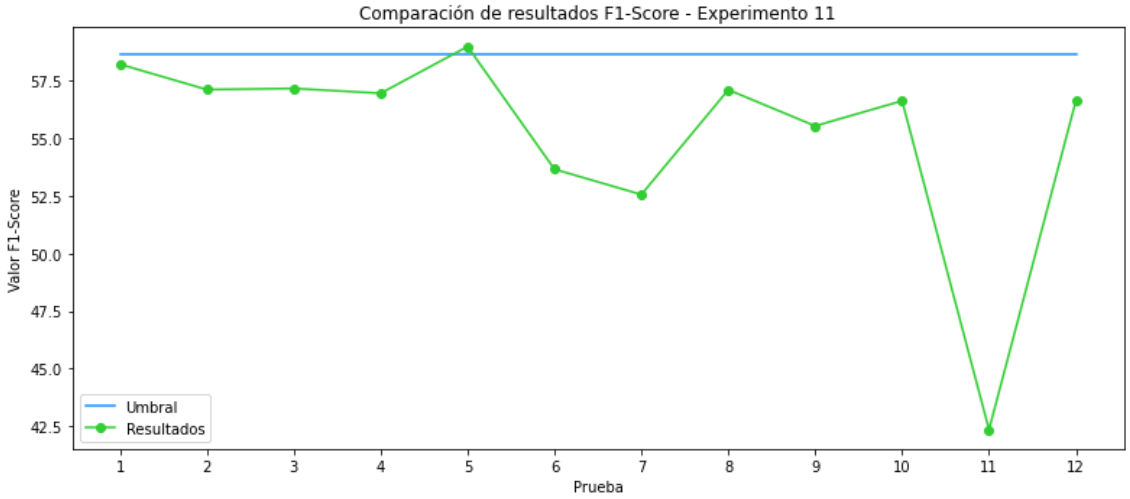


Ilustración 49: Comparación de resultados F1-Score experimento 11

Como podemos observar, solo en una de las pruebas hemos conseguido superar el umbral, concretamente con una F1-Score del 58.99%. Este valor lo conseguíamos con la configuración de dos capas *dropout*, en concreto empleábamos la segunda capa de *dropout* con un valor de 0.25 y la tercera capa de *dropout* con un valor de 0.35.

A rasgos generales, tras estas pruebas es cierto que hemos conseguido reducir el sobreajuste en nuestro modelo empleando las capas de *dropout*, pero observando los resultados obtenidos en nuestras métricas de calidad, no hemos sido capaces de mejorar nuestro modelo ni siquiera en un uno por ciento.

Por lo tanto, no vamos a emplear ninguna de las arquitecturas presentadas en este experimento ya que no conseguimos mejoras notables.

3.15 Experimento 12

3.15.1 Configuración de modelo y resultados

En este experimento vamos a incorporar más capas densas después de la capa de aplanamiento y antes de la capa de salida de nuestra red.

Con las capas de convolución lo que hace nuestro modelo es detectar aquellas características presentes en nuestras imágenes y que las pueden diferenciar unas de otras. Estas capas convolucionales generan unos filtros con dichas características que se pasan a las capas densas, las cuales se encargan de clasificar los filtros y decidir cuáles pueden ser útiles y cuáles no.

En una primera prueba incorporamos una nueva capa densa entre la capa de aplanamiento y la capa de salida. Esta nueva capa densa contendrá un total de 128 neuronas y una función de activación ReLU.

La topología y el número de parámetros quedaría como se muestra en la siguiente imagen:

Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Densa_1 (Dense)	(None, 128)	30482560
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 30,592,855
Trainable params: 30,592,855
Non-trainable params: 0

Ilustración 50: Topología 1 de red neuronal experimento 12

En este caso los resultados obtenidos tras el entrenamiento son de una pérdida de 0.01, una precisión del 99.84% y una F1-Score del 99.84%. En la siguiente gráfica podemos observar la evolución de las métricas de calidad durante el entrenamiento:

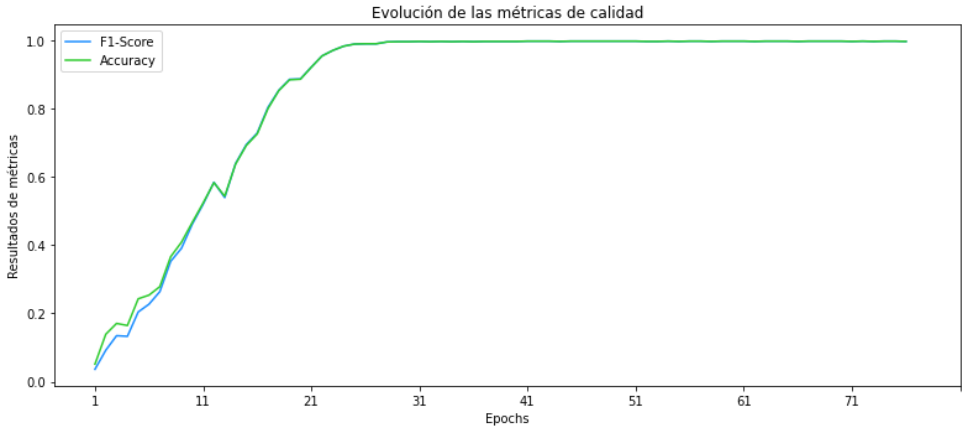


Ilustración 51: Evolución de métricas de calidad en experimento 12

En la siguiente gráfica se puede observar la evolución de la función de pérdida durante el entrenamiento:

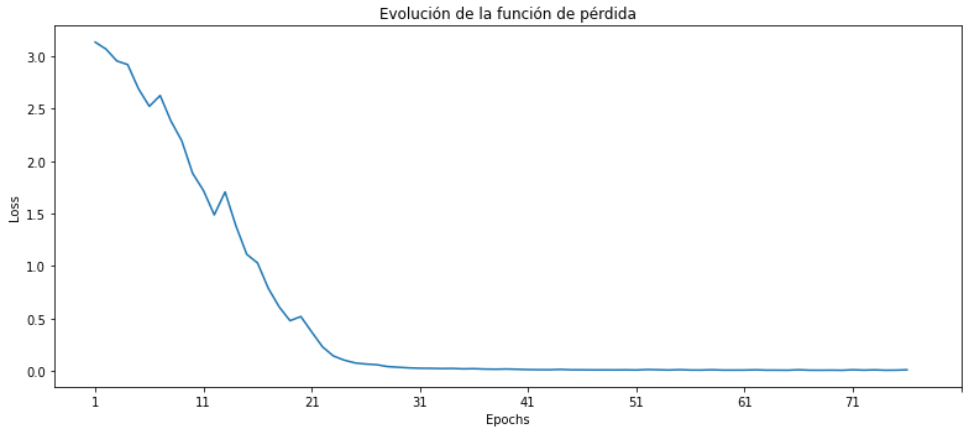


Ilustración 52: Evolución de pérdida en experimento 12

Al ejecutar nuestro modelo con los datos de validación, obtenemos una pérdida de 2.18, una precisión del 56.97 y una F1-Score del 57.74%.

Como vemos, en este caso no conseguimos mejorar los resultados de las métricas que ya teníamos previamente, en concreto, empeoran.

En nuestra siguiente prueba, tendremos dos capas densas entre la capa de aplanamiento y la de salida. La capa densa 1 tendrá un total de 512 neuronas y la capa densa 2 tendrá 128 neuronas, ambas tendrán una función de activación ReLU.

La topología y el número de parámetros de nuestra red neuronal quedaría como se muestra en la siguiente imagen:

Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_conv_3 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Densa_1 (Dense)	(None, 512)	121930240
Capa_Densa_2 (Dense)	(None, 128)	65664
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 122,106,199
Trainable params: 122,106,199
Non-trainable params: 0

Ilustración 53: Topología 2 de red neuronal experimento 12

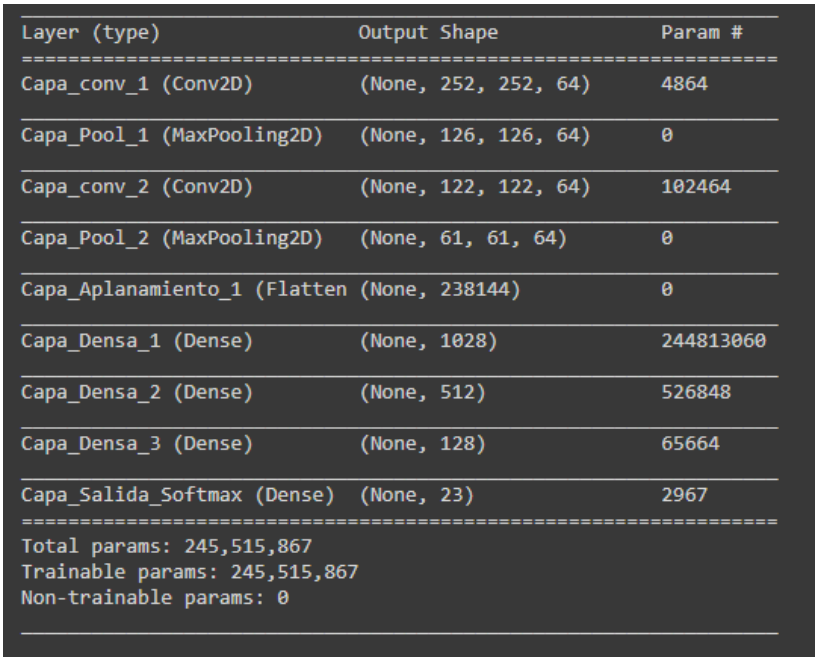
Los resultados obtenidos tras el entrenamiento son de una pérdida de 0.007, una precisión del 99.92% y una F1-Score del 99.91%.

Al pasarle a nuestro modelo los datos de validación obtenemos una pérdida de 2.02, una precisión del 56.35% y una F1-Score del 56.88%.

De nuevo, los resultados obtenidos no mejoran nuestros resultados previos.

En la siguiente prueba, emplearemos tres capas densas situadas entre la capa de aplanamiento y la capa de salida. La capa densa uno cuenta con un total de 1028 neuronas, la capa densa dos cuenta con 512 neuronas y la capa densa tres cuenta con 128 neuronas, las tres capas emplean una función de activación ReLU.

La topología y el número total de parámetros a entrenar puede visualizarse en la siguiente imagen:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Densa_1 (Dense)	(None, 1028)	244813060
Capa_Densa_2 (Dense)	(None, 512)	526848
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 245,515,867
Trainable params: 245,515,867
Non-trainable params: 0

Ilustración 54: Topología 3 de red neuronal experimento 12

Tras el entrenamiento obtenemos unos resultados de una precisión del 99.84%, una F1-Score del 99.84% y una pérdida de 0.005.

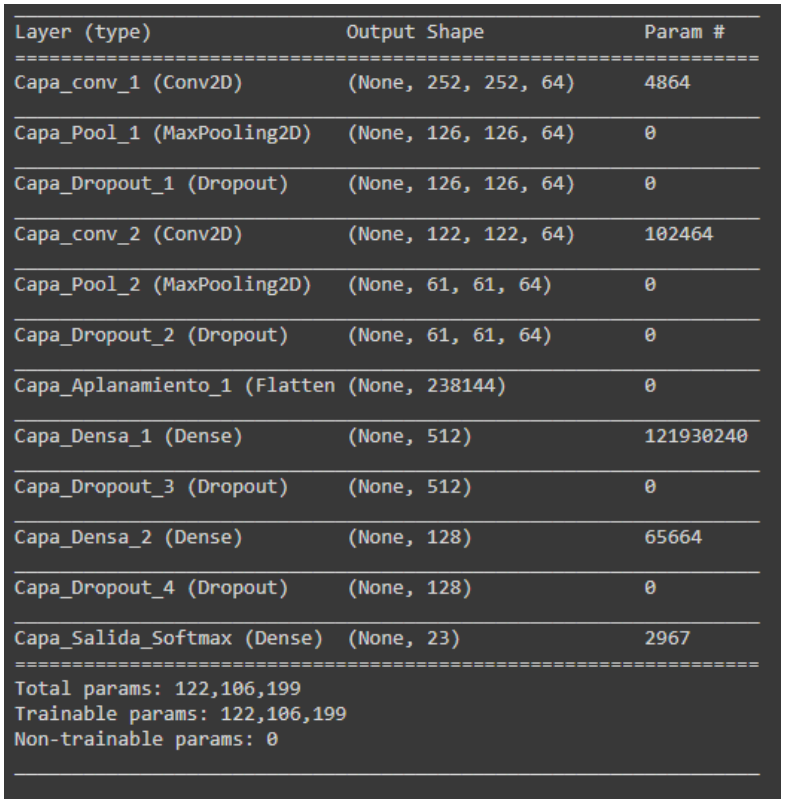
Con los datos de validación nos encontramos con una pérdida de 2.13, una precisión del 57.59% y una F1-Score del 58.17%.

Hemos visto que, empleando únicamente capas densas adicionales, nuestro modelo no mejora mucho más de lo que habíamos conseguido previamente. Por ello, ahora vamos a mezclar capas densas adicionales con capas de *dropout*.

En esta primera prueba vamos a emplear una capa de *dropout* después del primer par de capas de convolución y *pooling*, con un valor de 0.25, y una segunda capa de *dropout* después del segundo par de capas de convolución y pooling, también con un valor de 0.25.

Además, añadiremos dos capas densas entre la capa de aplanamiento y la de salida, la capa densa uno tendrá un total de 512 neuronas y la capa densa dos tendrá 128 neuronas, ambas emplearán una función de activación ReLU. Después de cada una de las dos capas densas mencionadas, añadiremos una capa de *dropout* con un valor de 0.5 en ambos casos.

Podemos visualizar en la siguiente imagen la topología de la red y el número de parámetros a entrenar:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Densa_1 (Dense)	(None, 512)	121930240
Capa_Dropout_3 (Dropout)	(None, 512)	0
Capa_Densa_2 (Dense)	(None, 128)	65664
Capa_Dropout_4 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 122,106,199
Trainable params: 122,106,199
Non-trainable params: 0

Ilustración 55: Topología 4 de red neuronal experimento 12

Estableceremos el número de etapas de entrenamiento en 140 y la paciencia para la parada en 10 etapas.

Tras el entrenamiento obtenemos una pérdida de 0.079, una precisión del 98.37% y una F1-Score del 98.37%.

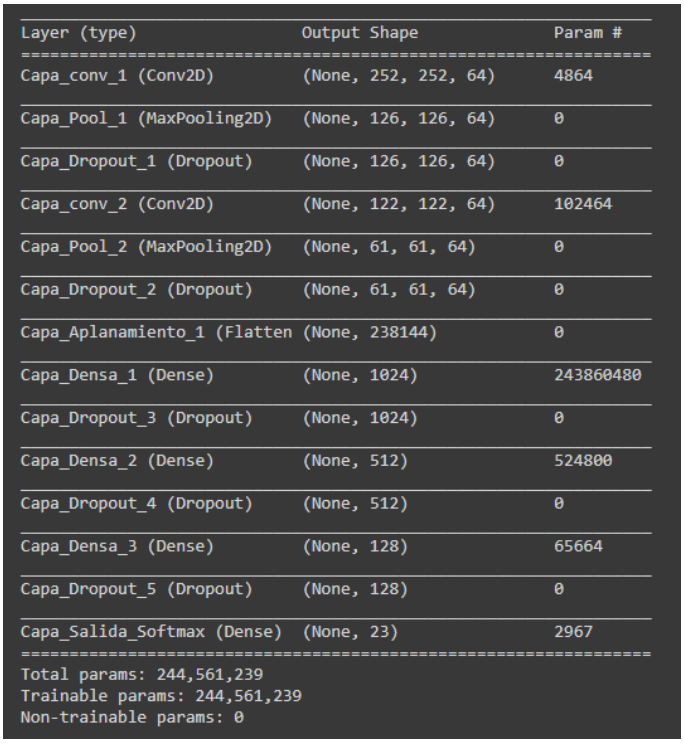
Con los datos de validación obtenemos una pérdida de 1.72, una precisión del 62.54% y una F1-Score del 62.09%.

Como vemos, con estos nuevos cambios hemos conseguido mejorar nuestro modelo de manera notable.

En esta siguiente prueba vamos a emplear una capa de *dropout* después del primer par de capas de convolución y *pooling*, con un valor de 0.25, y una segunda capa de *dropout* después del segundo par de capas de convolución y pooling, también con un valor de 0.25.

Además, añadiremos tres capas densas entre la capa de aplanamiento y la de salida, la capa densa uno tendrá un total de 1024 neuronas, la capa densa dos tendrá 512 neuronas y la capa densa tres tendrá en total 128 neuronas, las tres capas densas emplearán una función de activación ReLU. Después de cada una de las tres capas densas mencionadas, añadiremos una capa de *dropout* con un valor de 0.5 en cada caso.

Podemos observar en la siguiente imagen la topología de la red y el número de parámetros a entrenar:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 238144)	0
Capa_Densa_1 (Dense)	(None, 1024)	243860480
Capa_Dropout_3 (Dropout)	(None, 1024)	0
Capa_Densa_2 (Dense)	(None, 512)	524800
Capa_Dropout_4 (Dropout)	(None, 512)	0
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Dropout_5 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 244,561,239
Trainable params: 244,561,239
Non-trainable params: 0

Ilustración 56: Topología 5 de red neuronal experimento 12

Estableceremos el número de etapas de entrenamiento en 200 y la paciencia para la parada en 12 etapas. En este caso emplearemos un tamaño de *batch* en el entrenamiento de 150.

Tras el entrenamiento obtenemos una pérdida de 0.11, una precisión del 96.97% y una F1-Score del 96.98%.

Con los datos de validación obtenemos una pérdida de 1.74, una precisión del 61.61% y una F1-Score del 61.12%.

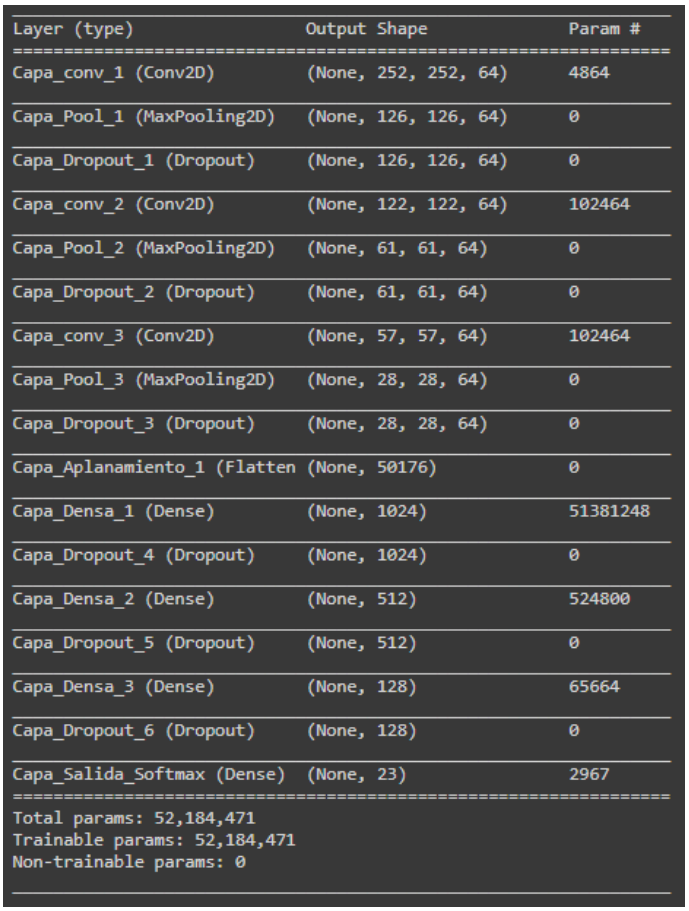
Como podemos ver, con esta configuración de la red neuronal obtenemos mejores resultados que en anteriores experimentos.

En esta siguiente prueba vamos a añadir un nueva capa de convolución de 64 filtros de 5x5 y con función de activación ReLU, y una capa de *pooling* de filtros de 2x2.

También vamos a emplear una capa de *dropout* después del primer par de capas de convolución y *pooling*, con un valor de 0.25, una segunda capa de *dropout* después del segundo par de capas de convolución y pooling, también con un valor de 0.25, y una tercera capa de *dropout* después del tercer par de capas de convolución y *pooling*, con un valor de 0.25.

Además, añadiremos tres capas densas entre la capa de aplanamiento y la de salida, la capa densa uno tendrá un total de 1024 neuronas, la capa densa dos tendrá 512 neuronas y la capa densa tres tendrá en total 128 neuronas, las tres capas densas emplearán una función de activación ReLU. Después de cada una de las tres capas densas mencionadas, añadiremos una capa de *dropout* con un valor de 0.5 en cada caso.

Podemos ver en la siguiente imagen la topología de la red y el número de parámetros a entrenar:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 64)	102464
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 64)	0
Capa_Dropout_3 (Dropout)	(None, 28, 28, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 50176)	0
Capa_Densa_1 (Dense)	(None, 1024)	51381248
Capa_Dropout_4 (Dropout)	(None, 1024)	0
Capa_Densa_2 (Dense)	(None, 512)	524800
Capa_Dropout_5 (Dropout)	(None, 512)	0
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Dropout_6 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 52,184,471
Trainable params: 52,184,471
Non-trainable params: 0

Ilustración 57: Topología 6 de red neuronal experimento 12

Estableceremos el número de etapas de entrenamiento en 300 y la paciencia para la parada temprana en 12 etapas. En este caso emplearemos un tamaño de *batch* en el entrenamiento de 150.

Tras el entrenamiento obtenemos una pérdida de 0.23, una precisión del 92.86% y una F1-Score del 92.83%.

Con los datos de validación obtenemos una pérdida de 1.64, una precisión del 63.78% y una F1-Score del 63.48%.

En esta prueba vemos que volvemos a mejorar los resultados que habíamos venido obteniendo previamente.

En la siguiente prueba vamos a añadir un nueva capa de convolución de 64 filtros de 5x5 y función de activación ReLU, y una capa de *pooling* de filtros de 2x2, es decir, tendremos tres capas de convolución y tres capas de *pooling*.

También vamos a emplear una capa de *dropout* después del primer par de capas de convolución y *pooling*, con un valor de 0.35, una segunda capa de *dropout* después del segundo par de capas de convolución y pooling, también con un valor de 0.35, y una tercera capa de *dropout* después del tercer par de capas de convolución y *pooling*, con un valor de 0.35.

Además, añadiremos tres capas densas entre la capa de aplanamiento y la de salida, la capa densa uno tendrá un total de 1024 neuronas, la capa densa dos tendrá 512 neuronas y la capa densa tres tendrá en total 128 neuronas, las tres capas densas emplearán una función de activación ReLU. Después de cada una de las tres capas densas mencionadas, añadiremos una capa de *dropout* con un valor de 0.6 en el caso de las dos primeras y con un valor de 0.5 en el caso de la última.

Podemos observar en la siguiente imagen la topología de la red y el número de parámetros a entrenar:

Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 64)	102464
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 64)	0
Capa_Dropout_3 (Dropout)	(None, 28, 28, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 50176)	0
Capa_Densa_1 (Dense)	(None, 1024)	51381248
Capa_Dropout_4 (Dropout)	(None, 1024)	0
Capa_Densa_2 (Dense)	(None, 512)	524800
Capa_Dropout_5 (Dropout)	(None, 512)	0
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Dropout_6 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 52,184,471
Trainable params: 52,184,471
Non-trainable params: 0

Ilustración 58: Topología 7 de red neuronal experimento 12

Estableceremos el número de etapas de entrenamiento en 300 y la paciencia para la parada en 12 etapas. En este caso emplearemos un tamaño de *batch* en el entrenamiento de 150.

Tras el entrenamiento obtenemos una pérdida de 0.90, una precisión del 70.89% y una F1-Score del 70.82%.

Con los datos de validación obtenemos una pérdida de 1.53, una precisión del 59.13% y una F1-Score del 58%.

En este caso, los resultados obtenidos no son tan buenos como en las anteriores pruebas y se asemejan a los de anteriores experimentos. Estos resultados nos hacen pensar que pudiera ser que la red neuronal no haya tenido las suficientes etapas de entrenamiento para aprender lo suficiente por ello, vamos a realizar otra prueba con la misma configuración.

En este caso estableceremos el número de etapas de entrenamiento en 430 y la paciencia para la parada en 12 etapas. Emplearemos un tamaño de *batch* en el entrenamiento de 150.

Después del entrenamiento obtenemos una pérdida de 0.23, una precisión del 92.55% y una F1-Score del 92.50%.

Con los datos de validación obtenemos una pérdida de 1.82, una precisión del 61.92% y una F1-Score del 61.64%.

Tras aumentar las etapas de entrenamiento, obtenemos mejores resultados que en la anterior ejecución de esta configuración de red neuronal.

En esta siguiente prueba vamos a añadir un nueva capa de convolución de 64 filtros de 5x5 y función de activación ReLU, y una capa de *pooling* de filtros de 2x2.

También vamos a emplear una capa de *dropout* después del primer par de capas de convolución y *pooling*, con un valor de 0.25, una segunda capa de *dropout* después del segundo par de capas de convolución y pooling, también con un valor de 0.25, y una tercera capa de *dropout* después del tercer par de capas de convolución y *pooling*, con un valor de 0.25.

Además, añadiremos tres capas densas entre la capa de aplanamiento y la de salida, la capa densa uno tendrá un total de 1024 neuronas, la capa densa dos tendrá 512 neuronas y la capa densa tres tendrá en total 128 neuronas, las tres capas densas emplearán una función de activación ReLU. Después de cada una de las tres capas densas mencionadas, añadiremos una capa de *dropout* con un valor de 0.5 en cada caso.

Podemos ver en la siguiente imagen la topología de la red y el número de parámetros a entrenar:

Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 64)	102464
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 64)	0
Capa_Dropout_3 (Dropout)	(None, 28, 28, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 50176)	0
Capa_Densa_1 (Dense)	(None, 1024)	51381248
Capa_Dropout_4 (Dropout)	(None, 1024)	0
Capa_Densa_2 (Dense)	(None, 512)	524800
Capa_Dropout_5 (Dropout)	(None, 512)	0
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Dropout_6 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Ilustración 59: Topología 8 de red neuronal experimento 12

Si el lector recuerda, ya hemos empleado esta configuración de red neuronal en pruebas anteriores, en este caso, vamos a establecer un total de 360 etapas de entrenamiento, una paciencia de parada de 12 etapas y un tamaño de *batch* de 150.

Tras el entrenamiento obtenemos una pérdida de 0.14, una precisión del 96.04% y una F1-Score del 96.05%.

Con los datos de validación obtenemos una pérdida de 1.82, una precisión del 65.33% y una F1-Score del 65.33%.

En este caso como vemos, los resultados obtenidos han evolucionado positivamente.

En la siguiente prueba, vamos a añadir un nueva capa de convolución de 64 filtros de 5x5 y función de activación ReLU, y una capa de *pooling* de filtros de 2x2.

También vamos a emplear una capa de *dropout* después del primer par de capas de convolución y *pooling*, con un valor de 0.25, una segunda capa de *dropout* después del segundo par de capas de convolución y pooling, también con un valor de 0.25, y una tercera capa de *dropout* después del tercer par de capas de convolución y *pooling*, con un valor de 0.25.

Además, añadiremos tres capas densas entre la capa de aplanamiento y la de salida, la capa densa uno tendrá un total de 1024 neuronas, la capa densa dos tendrá 512 neuronas y la capa densa tres tendrá en total 128 neuronas, las tres capas densas emplearán una función de activación ReLU. Después de cada una de las tres capas densas mencionadas, añadiremos una capa de *dropout* con un valor de 0.55 en cada caso.

Podemos ver en la siguiente imagen la topología de la red y el número de parámetros a entrenar:

Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 64)	102464
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 64)	0
Capa_Dropout_3 (Dropout)	(None, 28, 28, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 50176)	0
Capa_Densa_1 (Dense)	(None, 1024)	51381248
Capa_Dropout_4 (Dropout)	(None, 1024)	0
Capa_Densa_2 (Dense)	(None, 512)	524800
Capa_Dropout_5 (Dropout)	(None, 512)	0
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Dropout_6 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Ilustración 60: Topología 9 de red neuronal experimento 12

Esta vez estableceremos el número de etapas de entrenamiento en 400 y la paciencia para la parada en 12 etapas. Emplearemos un tamaño de *batch* en el entrenamiento de 150.

Tras el entrenamiento obtenemos una pérdida de 0.19, una precisión del 94.25% y una F1-Score del 94.26%.

Con los datos de validación obtenemos una pérdida de 1.76, una precisión del 64.71% y una F1-Score del 64.39%.

En este caso, los resultamos que obtenemos son muy buenos en comparación con los de anteriores experimentos.

A continuación, vamos a mostrar una gráfica en la que visualizaremos los resultados obtenidos en este experimento y cuáles de ellos superan el umbral de F1-Score máxima conseguida hasta el momento, 58.99%:

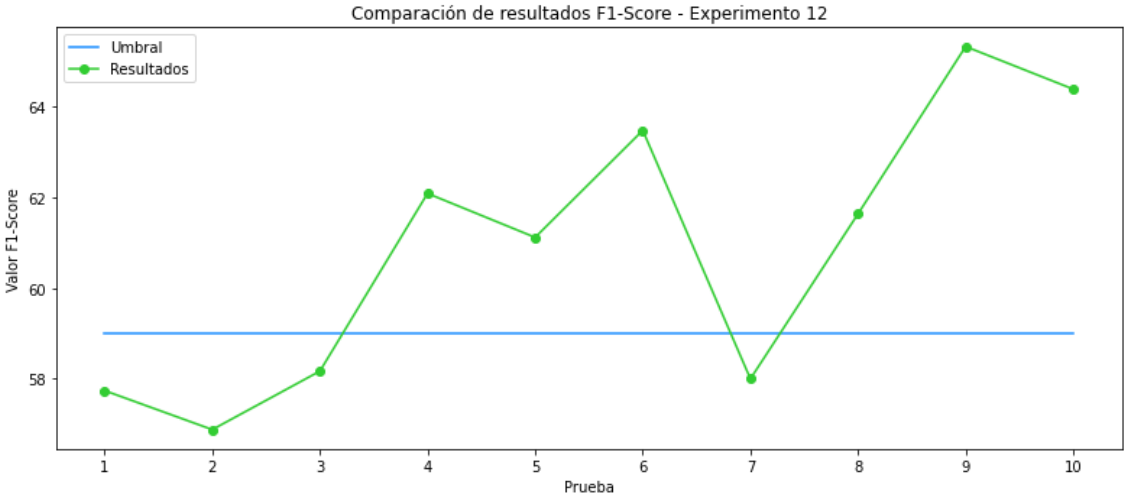


Ilustración 61: Comparación de resultados F1-Score experimento 12

A rasgos generales, vemos que la mayoría de las pruebas que hemos realizado en este experimento superan el umbral que teníamos hasta ahora. En concreto, llegamos a conseguir más de un 6% de mejoría.

Podemos concluir con que emplear capas densas unidas con capas de tipo *dropout* ha sido una buena mejora en nuestro modelo, consiguiendo el mejor de los resultados hasta el momento, un 65.33%.

3.16 Experimento 13

3.16.1 Configuración de modelo y resultados

En este último experimento vamos a realizar una técnica muy empleada en el mundo del *machine learning* cuando poseemos un conjunto de datos pequeño, dicha técnica es el aumento de datos o *data augmentation*.

Como hemos comentado y como su propio nombre indica, esta técnica consiste en aumentar el número de imágenes de nuestro conjunto de datos. Para ello, se emplean técnicas de transformación de imágenes como pueden ser hacer zoom sobre una región de la imagen, escalar la imagen, rotar la imagen o modificar el canal del color, entre otros.

En nuestro caso, hemos implementado una función que, dado un conjunto de datos y sus etiquetas, le aplicará una serie de transformaciones, previamente establecidas, y nos devolverá dichas imágenes transformadas de manera aleatoria y sus correspondientes etiquetas.

Vamos a aplicar dichas transformaciones y aumento de datos únicamente al conjunto de datos de entrenamiento, es decir, a un total de 1288 imágenes que se corresponden con el 80% del total del conjunto de datos.

Únicamente aplicamos el aumento de datos al conjunto de entrenamiento ya que nos interesa que nuestra red neuronal sea capaz de identificar más situaciones, pero no se lo aplicamos al conjunto de validación ya que queremos que nuestro modelo prediga sobre imágenes reales.

Cabe mencionar que no podemos añadir imágenes de manera infinita, ya que estaríamos desvirtuando nuestro modelo, es decir, comenzaría a aprender más sobre otras imágenes que no son los casos de estudio y podría comenzar a ofrecernos peores resultados.

En primer lugar, los aumentos que realicemos serán tres:

- Variaciones en el brillo de cara a obtener imágenes con mayor brillo, variando entre un 0% y un 30% el brillo.
- Variaciones en el brillo de cara a obtener imágenes mas oscuras, variando entre un 0% y un 20% el brillo.
- Variaciones en la posición de las imágenes aplicándoles una rotación de 0 grados a 20 grados.

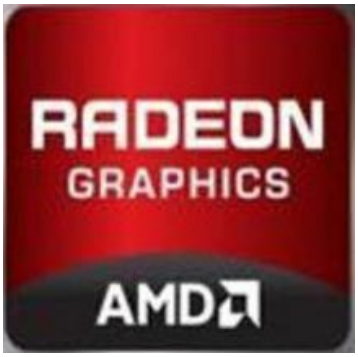


Ilustración 62: Imagen original sin aplicar técnicas de data augmentation

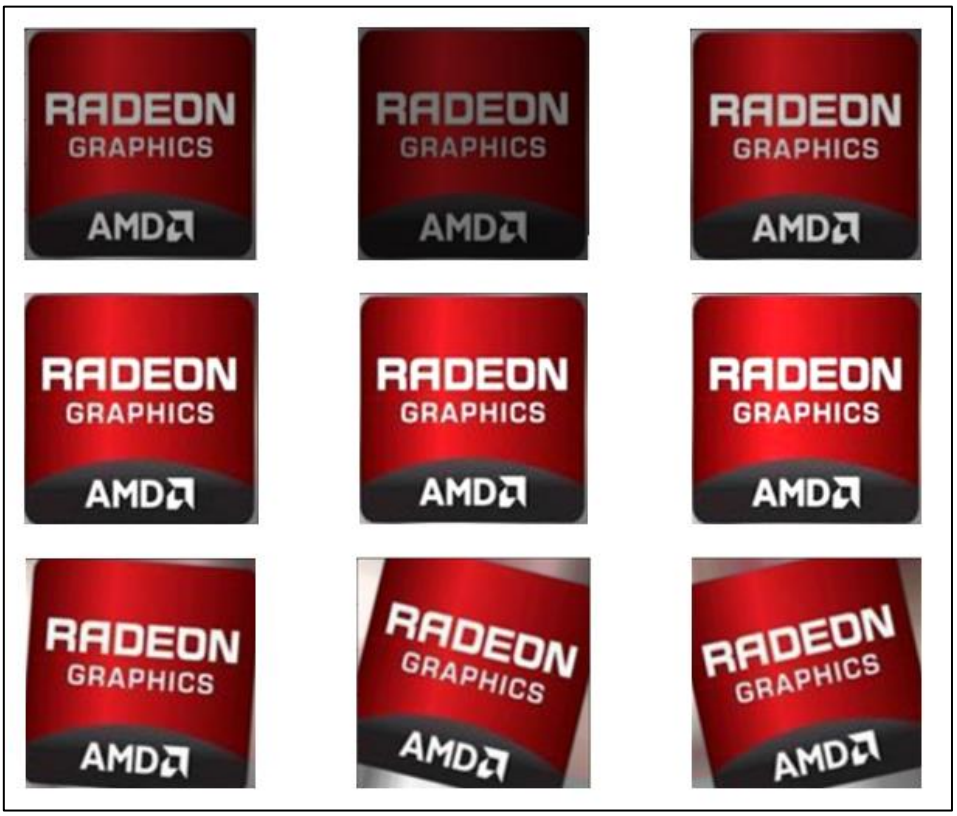
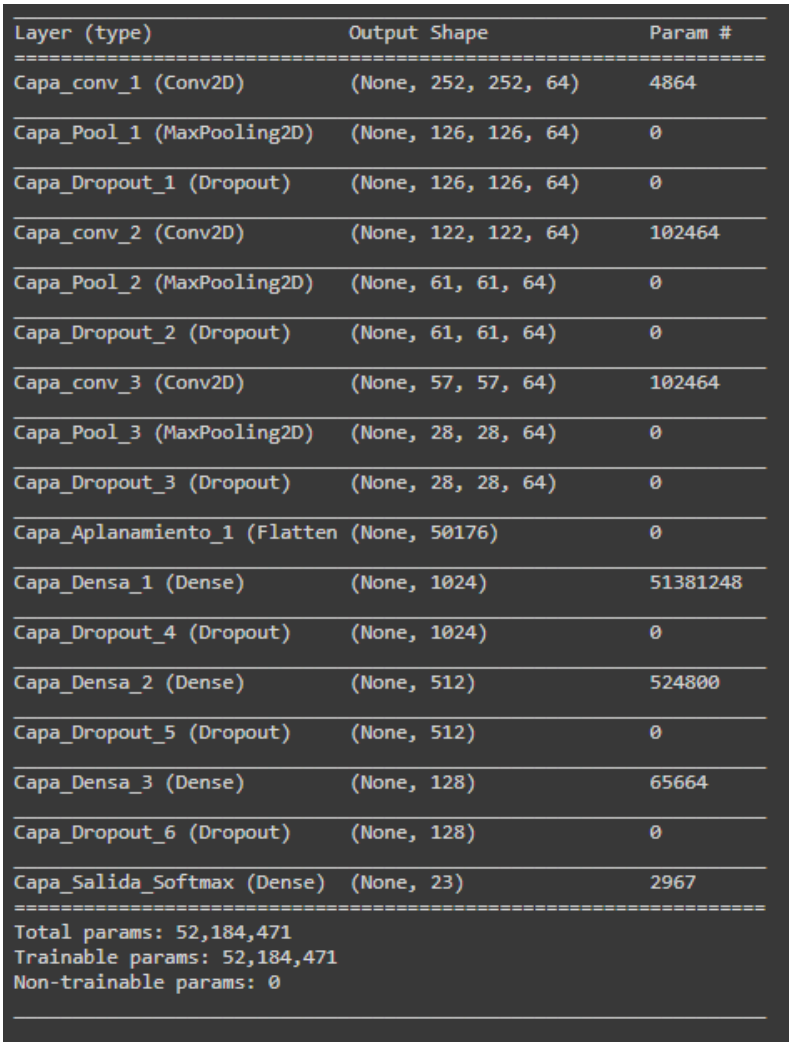


Ilustración 63: Ejemplos de aplicación de técnicas de data augmentation

En la imagen anterior se muestran algunos ejemplos de las transformaciones aplicadas sobre las imágenes para realizar el *data augmentation*. En la primera fila encontramos tres ejemplos de transformación en el brillo para obtener imágenes más oscuras, el la segunda fila encontramos ejemplos de transformaciones para realizar las imágenes más claras, y en la tercera fila nos encontramos con tres ejemplos de transformación en los que rotamos la imagen.

Con estas transformaciones pasamos de tener 1288 imágenes para el entrenamiento a 5152 imágenes.

En este experimento partimos de la red neuronal lograda en el experimento 12, la cual tenía la siguiente topología y número de parámetros a entrenar:



Layer (type)	Output Shape	Param #
Capa_conv_1 (Conv2D)	(None, 252, 252, 64)	4864
Capa_Pool_1 (MaxPooling2D)	(None, 126, 126, 64)	0
Capa_Dropout_1 (Dropout)	(None, 126, 126, 64)	0
Capa_conv_2 (Conv2D)	(None, 122, 122, 64)	102464
Capa_Pool_2 (MaxPooling2D)	(None, 61, 61, 64)	0
Capa_Dropout_2 (Dropout)	(None, 61, 61, 64)	0
Capa_conv_3 (Conv2D)	(None, 57, 57, 64)	102464
Capa_Pool_3 (MaxPooling2D)	(None, 28, 28, 64)	0
Capa_Dropout_3 (Dropout)	(None, 28, 28, 64)	0
Capa_Aplanamiento_1 (Flatten	(None, 50176)	0
Capa_Densa_1 (Dense)	(None, 1024)	51381248
Capa_Dropout_4 (Dropout)	(None, 1024)	0
Capa_Densa_2 (Dense)	(None, 512)	524800
Capa_Dropout_5 (Dropout)	(None, 512)	0
Capa_Densa_3 (Dense)	(None, 128)	65664
Capa_Dropout_6 (Dropout)	(None, 128)	0
Capa_Salida_Softmax (Dense)	(None, 23)	2967

Total params: 52,184,471
Trainable params: 52,184,471
Non-trainable params: 0

Ilustración 64: Topología de red neuronal experimento 13

En la primera prueba no hacemos modificaciones a la red mencionada y la entrenamos con el conjunto de datos aumentado para observar si recibimos mejorías en los resultados.

Los resultados obtenidos durante el entrenamiento son de una pérdida de 0.05, una precisión del 98.87% y una F1-Score del 98.87%.

Con los datos de validación obtenemos una pérdida de 2.86, una precisión del 65.63% y una F1-Score del 65.24%.

Como vemos, no conseguimos mejoras en nuestro modelo.

En la segunda prueba establecemos el número de etapas de entrenamiento en 180, el tamaño del *batch* de entrenamiento lo establecemos en 100 y la paciencia para la parada la establecemos en 8 etapas.

Tras el entrenamiento obtenemos una pérdida de 0.04, una precisión del 89.72% y una F1-Score del 98.72%.

Con los datos de validación obtenemos una pérdida de 2.31, una precisión del 67.18% y una F1-Score del 66.61%.

En este caso obtenemos cierta mejora en los resultados.

En la siguiente prueba reducimos el número de etapas de entrenamiento, estableciéndolas en 100 etapas.

En esta ocasión tras el entrenamiento obtenemos una pérdida de 0.08, una precisión del 97.81% y una F1-Score del 97.81%.

Con los datos de validación nos encontramos con una pérdida de 2.28, una precisión del 66.87% y una F1-Score del 66.27%.

En la próxima prueba aumentaremos el número de etapas de entrenamiento estableciéndolas en 115 etapas.

Después del entrenamiento obtenemos una pérdida de 0.06, una precisión del 98.49% y una F1-Score del 98.49%.

Con los datos de validación obtenemos una pérdida de 2.40, una precisión del 66.87% y una F1-Score del 66.46%.

En esta prueba vamos a cambiar los generadores de imágenes sintéticas para el aumento del conjunto de datos de entrenamiento:

- Generaremos imágenes con una variación en el brillo entre el 0% y el 40% para hacerlas más claras. Además, les aplicaremos una rotación de entre 0 grados y 20 grados.
- Generaremos imágenes con una variación en el brillo entre el 0% y el 40% para hacerlas más oscuras. Además, les aplicaremos una rotación de entre 0 grados y 20 grados.

En este caso, pasamos a tener un total de 3864 imágenes para el entrenamiento.

Para el entrenamiento continuamos con 115 etapas, un tamaño de *batch* de 100 y una paciencia para la parada temprana de 8 etapas.

En el entrenamiento obtenemos una pérdida de 0.14, una precisión del 95.57% y una F1-Score del 95.57%.

Con los datos de validación nos encontramos con una pérdida de 1.87, una precisión del 68.42% y una F1-Score del 67.84%.

En la siguiente prueba vamos a aumentar el número de etapas de entrenamiento, lo configuraremos en 130 etapas.

Tras el entrenamiento obtenemos una pérdida de 0.09, una precisión del 97.05% y una F1-Score del 97.05%.

Con los datos de validación obtenemos una pérdida de 2.17, una precisión del 68.11% y una F1-Score del 68.39%. El resultado obtenido con esta configuración es realmente bueno, en comparación con los valores de los experimentos iniciales.

Para esta prueba volvemos a aumentar el número de etapas de entrenamiento. En este caso, establecemos en 150 el número de etapas para el entrenamiento.

Después del entrenamiento obtenemos una pérdida de 0.07, una precisión del 98.16% y una F1-Score del 98.16%.

Con los datos de validación nos encontramos con una pérdida de 2.19, una precisión del 68.11% y una F1-Score del 68.14%.

En la siguiente prueba probaremos a añadir más filtros a la primera capa de convolución, en concreto, 128 filtros de tamaño 5x5. También, vamos a aumentar el número de etapas de entrenamiento, estableciéndolas en un total de 200 etapas.

Los resultados obtenidos tras el entrenamiento son de una pérdida de 0.07, una precisión del 98.01% y una F1-Score del 98.01%

Con los datos de validación obtenemos una pérdida de 2.32, una precisión del 66.87% y una F1-Score del 66.99%.

Los resultados obtenidos en esta prueba no son malos, pero no mejores que los que ya teníamos.

En esta prueba vamos a incrementar el número de filtros a extraer en la primera y la segunda capa de convolución, en total vamos a extraer en cada una 128 filtros de tamaño 5x5. Incrementaremos el número de etapas de entrenamiento a 215 etapas.

Tras el entrenamiento obtenemos una pérdida de 0.06, una precisión del 98.55% y una F1-Score del 98.55%.

Con los datos de validación obtenemos una pérdida de 2.56, una precisión del 65.63% y una F1-Score del 65.49%.

En la siguiente prueba reduciremos el número de etapas de entrenamiento a un total de 150 etapas. Continuaremos con la configuración de 128 filtros en la primera y segunda capa de convolución.

Tras el entrenamiento obtenemos una pérdida de 0.07, una precisión del 98.19% y una F1-Score del 98.19%.

Con los datos de validación obtenemos una pérdida de 2.29, una precisión del 67.49% y una F1-Score del 67.11%.

Para nuestra próxima prueba de nuevo estableceremos la primera y segunda capa de convolución en 64 filtros y la tercera capa la estableceremos en 32 filtros de 5x5.

En este caso, establecemos el número de etapas de entrenamiento en 125 y el tamaño del *batch* en 80.

Tras el entrenamiento obtenemos una pérdida de 0.17, una precisión del 94.77% y una F1-Score del 94.77%

Con los datos de validación nos encontramos con una pérdida 2.39, una precisión del 63.47% y una F1-Score del 63.06%.

En la siguiente prueba continuamos con la configuración de 64 filtros en la primera y segunda capa de convolución, y con 32 filtros en la tercera capa de convolución. En este caso aumentamos el número de etapas de entrenamiento estableciéndolas en 155.

Durante el entrenamiento obtenemos una pérdida de 0.09, una precisión del 97.52% y una F1-Score del 97.52%.

Con los datos de validación obtenemos una pérdida de 2.34, una precisión del 66.56% y una F1-Score del 66.21%.

A continuación, vamos a ver una gráfica con los resultados obtenidos en este experimento y el umbral a superar con las métricas de calidad, con respecto a lo que habíamos conseguido en experimentos previos:

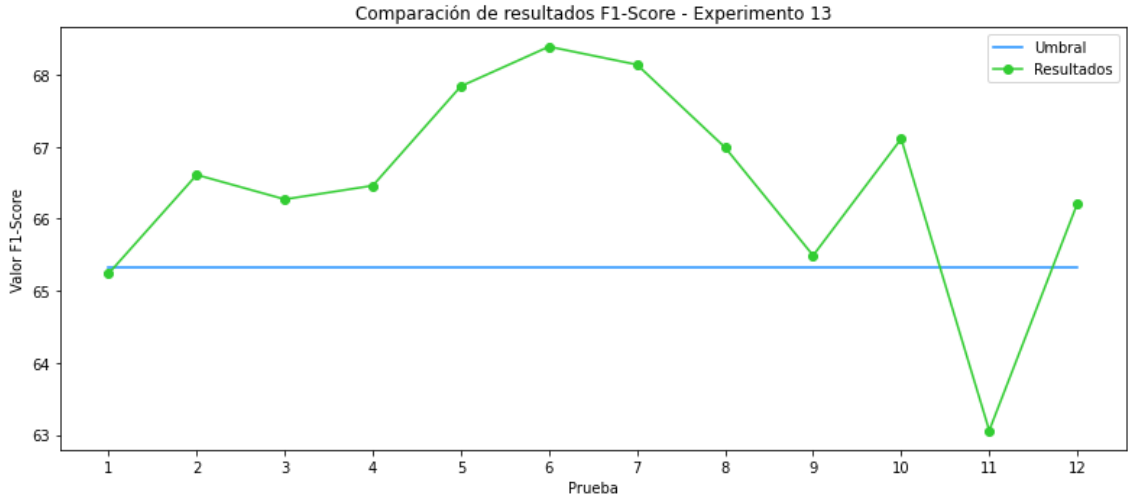


Ilustración 65: Comparación de resultados F1-Score experimento 13

A rasgos generales, en la gran mayoría de pruebas que hemos realizado en este experimento, conseguimos mejorar el valor que teníamos previamente de F1-Score.

En concreto, llegamos a conseguir una F1-Score del 68.39%, un valor que es muy bueno.

Capítulo 4. RESULTADOS Y CONCLUSIONES

4.1 Resultados logrados

4.1.1 Modelos conseguidos

Tras diversos experimentos, hemos logrado evolucionar un modelo simple, con resultados pobres, en un modelo con resultados muy positivos desde el punto de vista de los objetivos planteados.

En la siguiente gráfica podemos ver un resumen de los mejores resultados conseguidos, en lo referido a la métrica F1-Score, en cada experimento con el fin de ver la evolución de nuestro modelo:

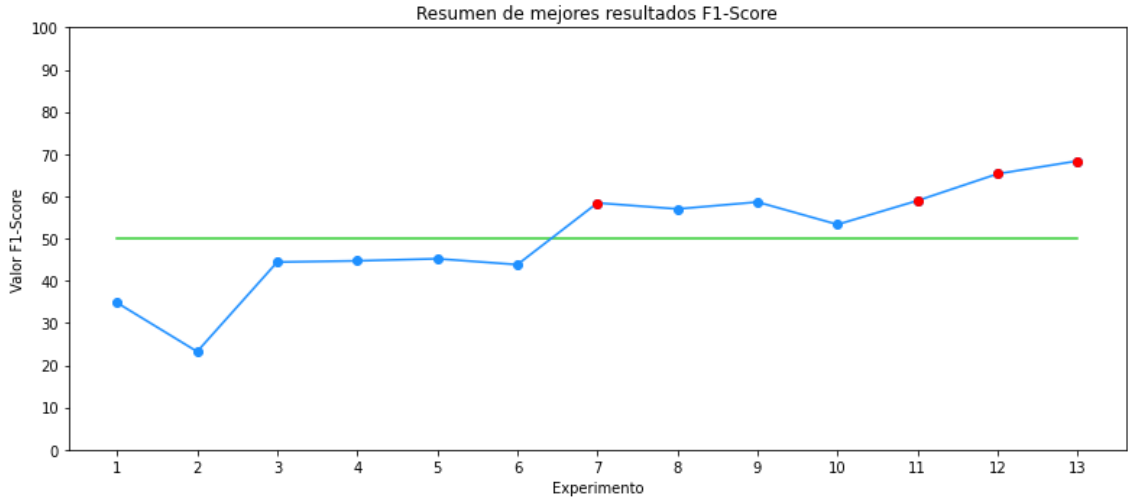


Ilustración 66: Resumen final de mejores resultados F1-Score

En la gráfica nos encontramos, por un lado, con una línea verde que delimita el 50% de F1-Score, valor que teníamos como objetivo superar, y, por otro lado, nos encontramos con los puntos que indican los mejores resultados de F1-Score conseguidos en cada experimento, marcando en rojo aquellos puntos pertenecientes a experimentos en los que tenemos algún logro que destacar.

Como vemos, partimos de un clasificador en el que obtenemos una F1-Score del 34.78% y durante los seis primeros experimentos nos mantenemos por debajo del 50%, es decir, resultados nada aceptables.

A partir del séptimo experimento, conseguimos al fin superar el umbral de 50%, con una buena cifra como es un 58.65%, esto lo conseguimos gracias a aumentar el número de etapas de entrenamiento de nuestro modelo. Durante cinco experimentos nos mantenemos en valores similares entre el 53% y el 59%, consiguiendo en el experimento 11 una F1-Score del 58.99%.

A partir del experimento 12 conseguimos superar el umbral del 60%, obteniendo una F1-Score del 65.33%. Esto lo logramos combinando más capas densas con capas de tipo *dropout*.

Finalmente, en el experimento 13, alcanzamos una F1-Score del 68.39% gracias a mezclar el modelo logrado en el experimento 12 con *data augmentation*.

La topología finalmente elegida la podemos ver resumida en la siguiente imagen:

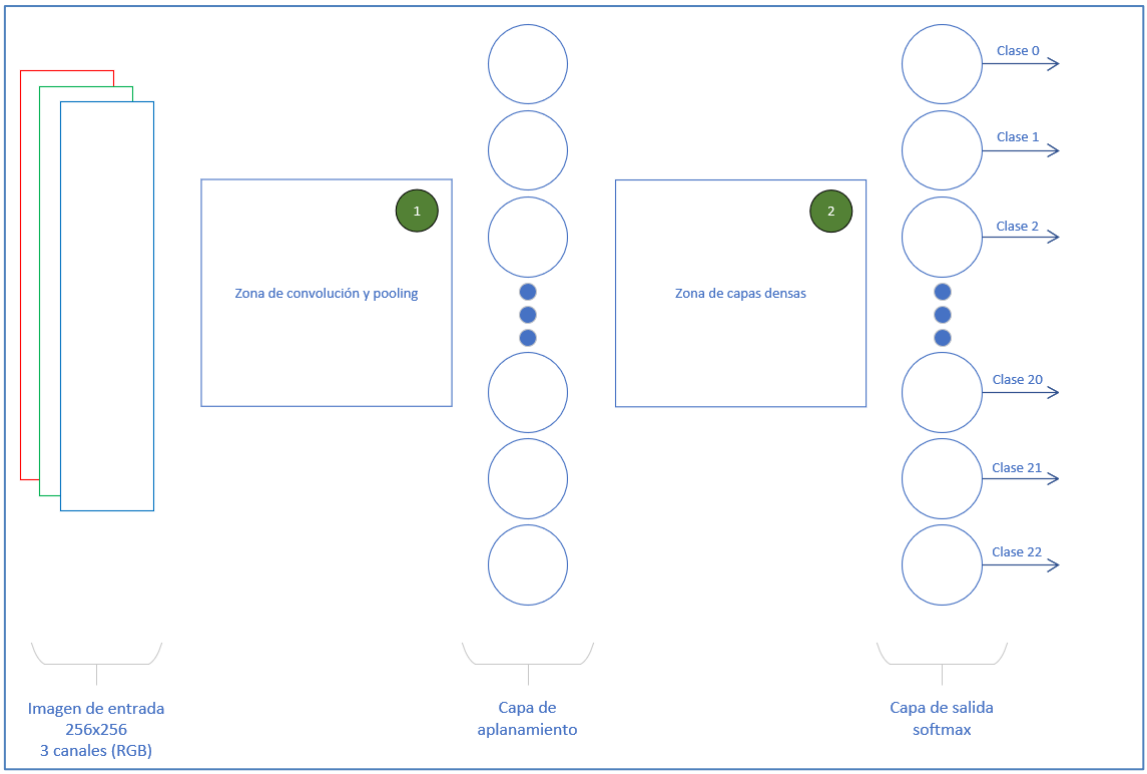


Ilustración 67: Topología final elegida

Si explicamos la imagen mostrada, la topología contiene las siguientes capas:

1. Nos encontramos con una entrada de 256x256 píxeles y tres canales de colores (RGB).
2. Una zona de capas de convolución y *pooling* que posteriormente veremos en detalle.
3. Una capa de aplanamiento.
4. Una zona de capas densas que veremos con mayor detalle posteriormente.
5. Una capa de salida softmax que devuelve un vector de probabilidades.

En la zona de convolución nos encontramos con tres pares de capas de convolución y *pooling*:

- Las capas de convolución extraen 64 filtros de tamaño 5x5 y emplean una función de activación ReLU.
- Las capas de *pooling* son de tipo MaxPooling y emplean una máscara de dimensión 2x2.

Después de cada par de capas de convolución y *pooling* nos encontramos una capa de dropout con un factor de 0.25.

En la siguiente imagen podemos ver de manera aumentada la zona de capas de convolución y *pooling*:

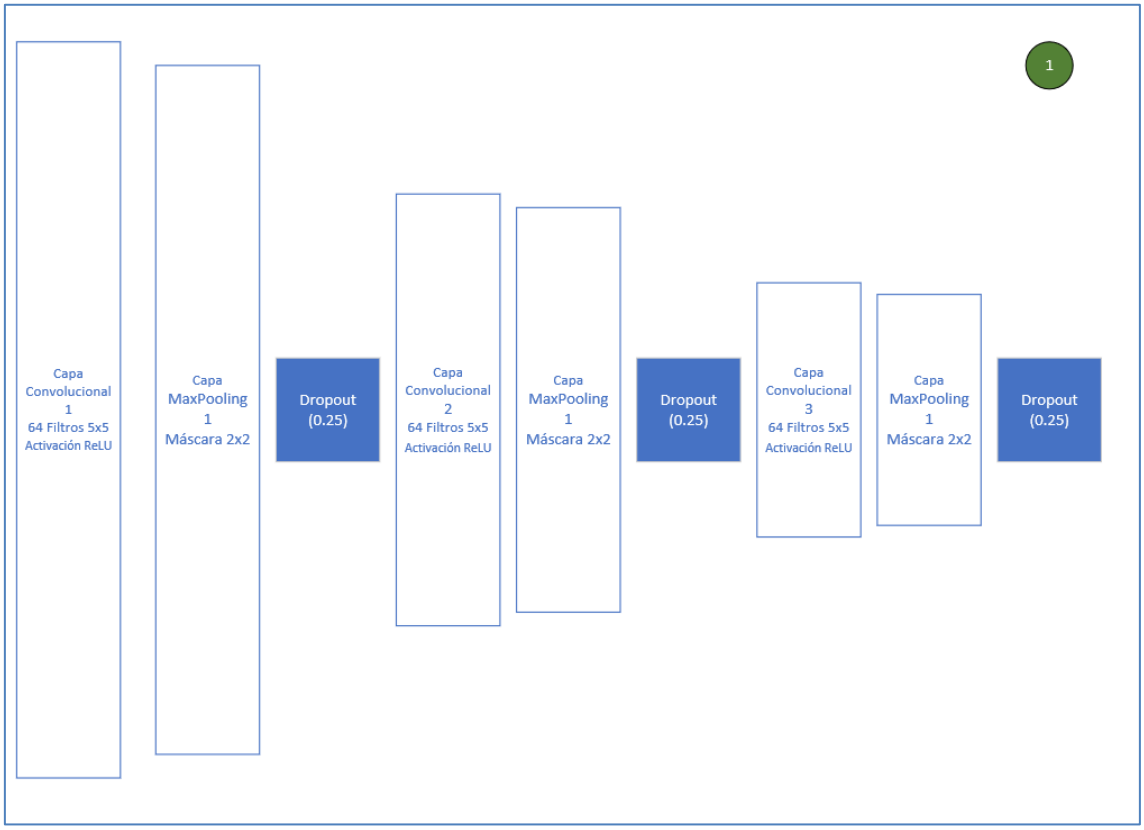


Ilustración 68: Topología final elegida - Zona de capas convolucionales y pooling

Si nos centramos en la zona de capas densas nos encontramos con la siguiente configuración de capas:

1. Capa densa de 1024 neuronas.
2. Capa densa de 512 neuronas.
3. Capa densa de 128 neuronas.

Todas las capas emplean una función de activación ReLU.

Después de cada capa densa nos encontramos una capa de tipo dropout con un factor de 0.5.

En la siguiente imagen podemos ver de manera aumentada la zona de capas densas:

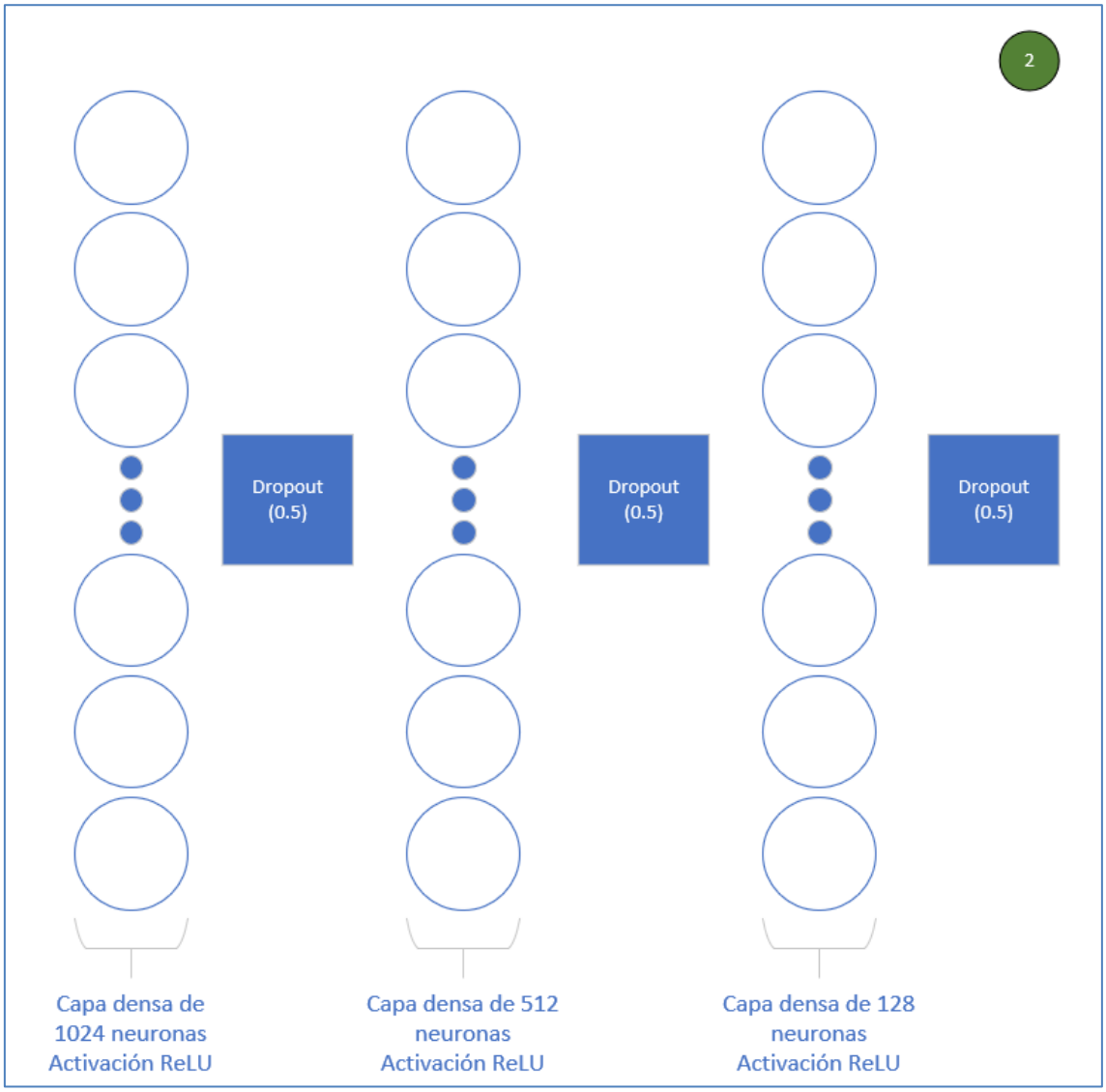


Ilustración 69: Topología final elegida - Zona de capas densas

Cabe mencionar que, finalmente, las técnicas de dropout empleadas con esta configuración de red neuronal fueron las siguientes:

- Un primer generador de imágenes que aplica una variación en el brillo entre el 0% y el 40% para hacerlas más claras. Además, les aplicaremos una rotación de entre 0 grados y 20 grados.
- Un segundo generador de imágenes que aplica una variación en el brillo entre el 0% y el 40% para hacerlas más oscuras. Además, les aplicaremos una rotación de entre 0 grados y 20 grados.

En último lugar, resulta interesante analizar en que clases acierta más y en cuales se equivoca en mayor grado nuestro modelo. Para ello, hemos generado la matriz de confusión para el modelo presentado y podemos verla en la siguiente imagen:

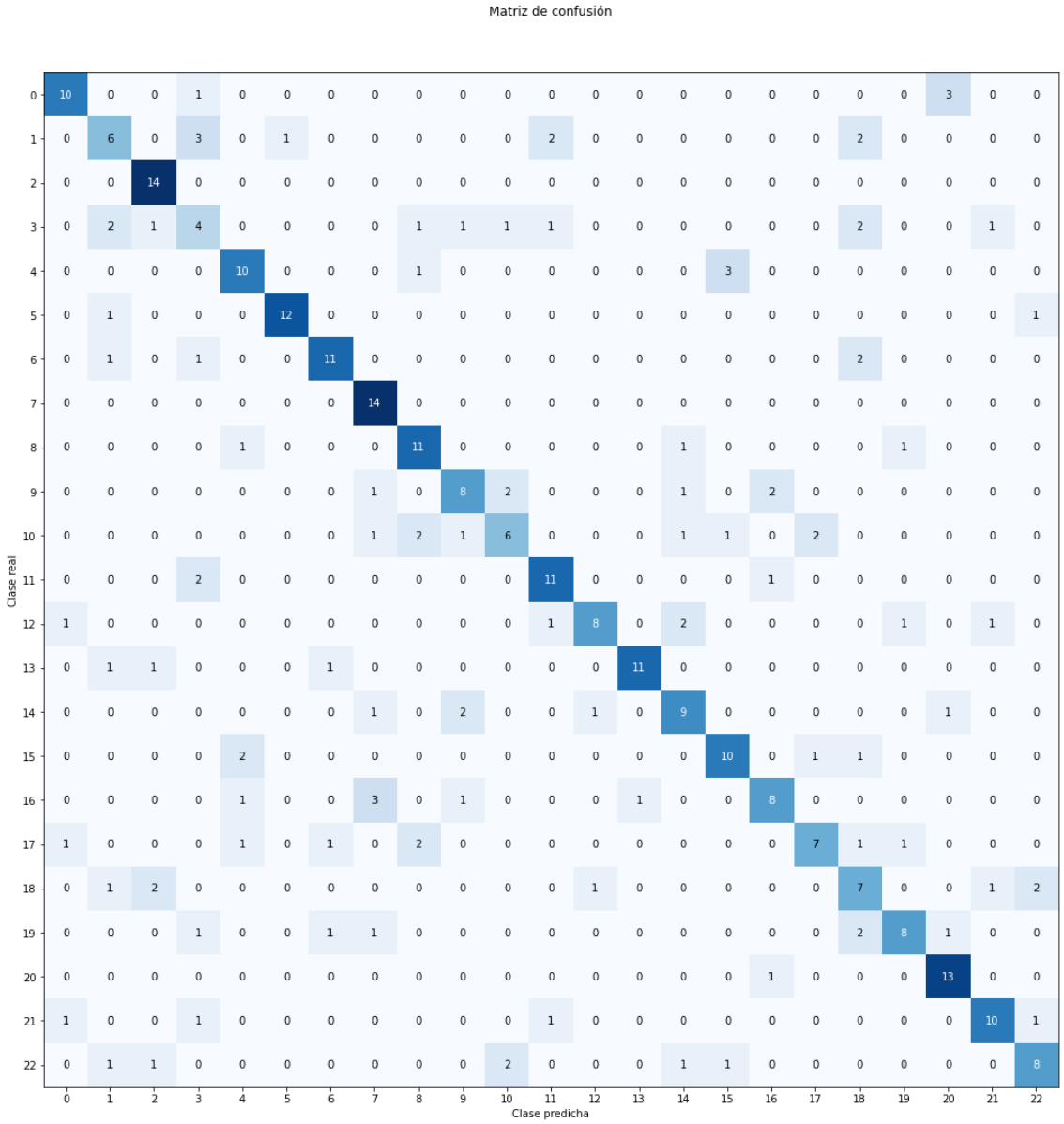


Ilustración 70: Matriz de confusión del modelo final

Si analizamos la matriz de confusión por porcentaje de acierto nos encontramos con lo siguiente:

- **Igual o mayor al 90%**: Nos encontramos con las clases 2 , 7 y 20 (D-Link, KitKat y Tic Tac).
- **Entre 70 % y 90%**: Nos encontramos con las clases 0, 4, 5, 6, 8, 11, 13, 15 y 21 (AMD, Domino's Pizza, Hellman's, IBM, LG, Milka, Nestea, Pepsi y Universal).
- **Entre 50% y 70%**: En esta franja encontramos las clases 9, 12, 14, 16, 17, 18, 19 y 22 (Lipton, Monster, Pac-Man, Pizza Hut, Red Bull, Samsung, Sony y Dell).
- **Por debajo del 50%**: Nos encontramos con las clases 1, 3 y 10 (Aquafina, Disney, McDonald's).

De manera resumida, las clases con las que mejor se comporta nuestro algoritmo, con un 100% de acierto, son las clases D-Link y KitKat, y la clase con la que peor se comporta, con un acierto del 28.50% (4 aciertos de 14 muestras), es con la clase Disney.

Esto tiene cierto nivel de lógica y coherencia ya que la clase Disney es una de la que mas ruido y distorsión tiene.

4.1.2 Tiempo y costes

Otro punto interesante que hemos analizado en este trabajo es el factor tiempo. Concretamente, hemos analizado el tiempo que tardamos en ejecutar determinados modelos de redes neuronales con una arquitectura CPU y con una arquitectura GPU.

Si realizamos un promedio de los tiempos obtenidos, en total tardamos de media en ejecutar con una arquitectura CPU 33 minutos, mientras que si ejecutamos con una arquitectura GPU tardamos tan solo 1.5 minutos, es decir, es prácticamente 22 veces más rápido ejecutar con GPU que con CPU un modelo de redes neuronales.

Cuando nos trasladamos a un entorno empresarial o productivo el factor tiempo es muy importante, ya que nos puede suponer grandes beneficios económicos o grandes pérdidas.

Supongamos que poseemos una empresa y tenemos contratado a un empleado que cobra 25€ de media a la hora como informático programando y ejecutando modelos de *deep learning*.

Al mismo tiempo, supongamos que, mientras este empleado esta ejecutando su modelo, no puede realizar otras tareas de programación porque tiene el sistema ocupado. En este caso, si la empresa cuenta con una arquitectura CPU, el empleado mientras ejecuta el modelo le estaría costando un total 13.75€, mientras que si posee una arquitectura GPU, únicamente le costaría 0.63€.

Visto de esta manera parece una diferencia no muy grande, pero si lo llevamos a, por ejemplo, 100 ejecuciones y lo visualizamos en una gráfica vemos muy claramente el margen económico:

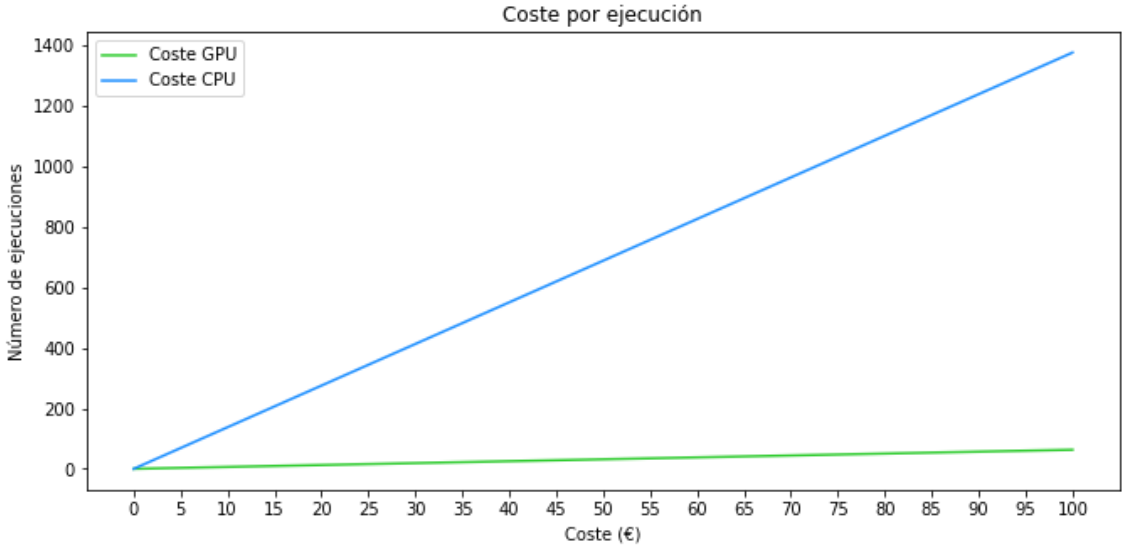


Ilustración 71: Ejemplo de coste por 100 ejecuciones

Observamos que, mientras el coste por ejecución con CPU se dispara tras 100 ejecuciones, el coste con GPU permanece prácticamente inmóvil. En concreto, nos deja una diferencia de más de 1300€ tras 100 ejecuciones.

Ahora, vamos a revisar cuanto nos costaría una unidad de cómputo en Google Cloud con CPU y con GPU. Cabe mencionar que, de cara a las estimaciones, cogeremos máquinas de características lo más similares posibles.

Una máquina con CPU nos costaría alrededor de 0.19€ por hora, mientras que una máquina con GPU nos costaría 0.35€ por hora. [21][22]

Si empleamos estos precios para calcular cuanto nos costaría ejecutar nuestros modelos con el tiempo promedio calculado, nos quedaría lo siguiente:

- Ejecutar el modelo con CPU nos llevaría 33 minutos, por lo que nos costaría un total de 0.11€.
- Ejecutar el modelo con GPU no llevaría 1.5 minutos, por lo que nos costaría un total de 0.009€, es decir, menos de 1 céntimo.

Podemos ver de nuevo la evolución de los precios tras 100 ejecuciones, pero en este caso con los costes en Google Cloud:

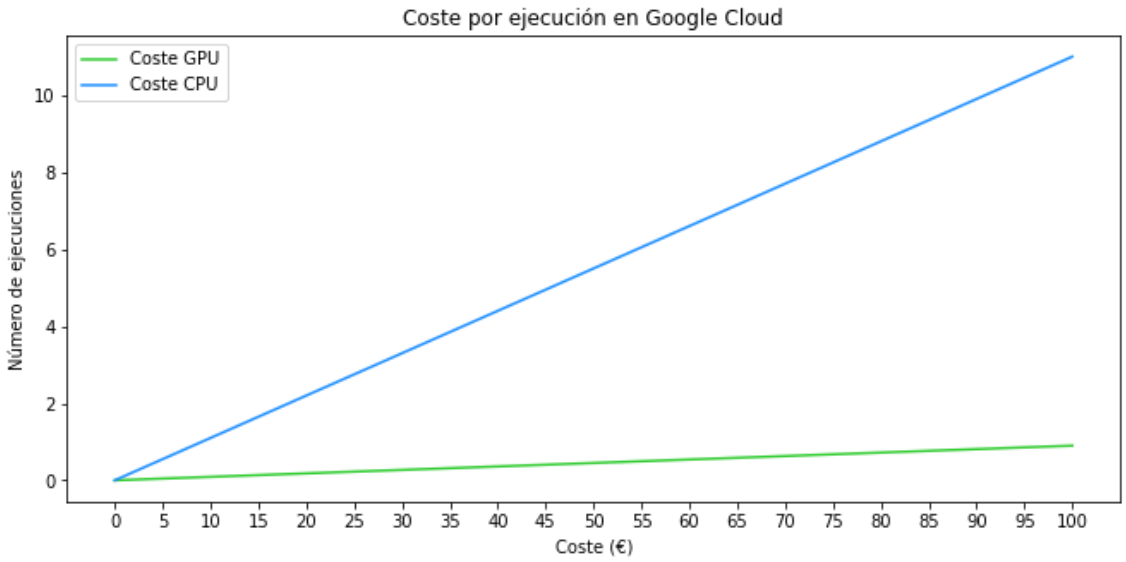


Ilustración 72: Ejemplo de coste por 100 ejecuciones en Google Cloud

En este caso, vemos como se dispara de nuevo el coste con CPU, aunque ahora tenemos un margen más pequeño de cerca de 10€ debido al coste del cloud.

Como vemos, dependiendo de la situación en la que nos encontremos programando, conviene invertir más en un tipo de tecnología que en otro. En este caso, cuando se trata de redes neuronales, conviene invertir en entornos de cómputo capacitados con GPU aunque sean un poco más caros, ya que estaremos ahorrando una enorme cantidad de dinero para la empresa y de tiempo para los trabajadores.

4.2 Conclusiones

En este trabajo hemos partido de cero y hemos visto que la disciplina del *deep learning* es un mundo muy amplio y lleno de muchas posibilidades que explorar.

Existe una infinitud de configuraciones que realizar en nuestra red neuronal y, a veces, todo aquello que puede ser conveniente para resolver un problema, puede no darnos resultados positivos.

Hemos comprobado que, cuando inicias el camino en *deep learning*, se trata de ir explorando conceptos y configuraciones hasta dar con una adecuada. En este caso, hemos llegado a una configuración con un resultado del 68.39%, es decir, un resultado que supera de manera amplia nuestro objetivo propuesto de superar un 50%.

Poco a poco, según se vaya adquiriendo experiencia en este campo, será más fácil configurar modelos de redes neuronales sin necesidad de experimentar de manera tan amplia con las capas de nuestra red.

Para concluir, también hemos visto que es importante en muchas ocasiones invertir más en tecnología en lugar de movernos en la opción más barata, ya que, a la larga, nos puede suponer un gran ahorro económico para nuestro negocio.

4.3 Trabajos futuros

Tras finalizar este trabajo, conviene pensar en que podemos hacer en el futuro, es decir, que mejoras hacer en nuestro trabajo o donde aplicarlo. Vamos a mencionar una serie de modificaciones o aplicaciones a realizar:

- Eliminar imágenes de determinadas clases que incorporan ruido a nuestro algoritmo.
- Emplear técnicas de *transfer learning* con el objetivo de mejorar la calidad de nuestro modelo.
- Emplear técnicas adicionales de *data augmentation*.
- Ampliar el conjunto de logotipos a reconocer o incrementar el número de imágenes de las clases actuales.
- Combinar nuestro clasificador de logotipos con un reconocedor de objetos en video.

Como vemos, hemos mencionado algunas de las posibles modificaciones a realizar, pero dentro del mundo del *machine learning* las posibilidades son infinitas.

BIBLIOGRAFÍA

[1] L. J. Aguilar, *Big Data: Análisis de grandes volúmenes de datos en organizaciones*. Marcombo.

[2] "Tipos de aprendizaje automático." [Online]. Available: https://medium.com/soldai/tipos-de-aprendizaje-autom%C3%A1tico-6413e3c615e2. [Accessed: 2021]

[3] S. Raschka and V. Mirjalili, *Python Machine Learning*. Marcombo.

[4] "Aprendizaje automático por refuerzo." [Online]. Available: https://www.aprendemachinelearning.com/aprendizaje-por-refuerzo/. [Accessed: 2021]

[5] "Redes neuronales." [Online]. Available: https://sites.google.com/site/mayinteligenciartificial/unidad-4-redes-neuronales. [Accessed: 2021]

[6] J. Torres, *Python Deep Learning: Introducción práctica con Keras y TensorFlow 2*. Marcombo.

[7] "A Comprehensive Guide to Convolutional Neural Networks." [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. [Accessed: 2021]

[8] "¿Cómo funcionan las Convolutional Neural Networks?" [Online]. Available: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/. [Accessed: 2021]

[9] M. Lutz, *Python Pocket Reference*. O'Reilly.

[10] M. Lutz, *Learning Python*. O'Reilly.

[11] "TensorFlow." [Online]. Available: https://www.tensorflow.org/?hl=es-419. [Accessed: 2021]

[12] A. Audevart, K. Banachewicz, and L. Massaron, *Machine Learning Using TensorFlow Cookbook*. Packt Publishing.

[13] A. Gulli, A. Kapoor, and S. Pal, *Deep Learning with TensorFlow 2 and Keras*. Packt Publishing.

[14] "What is Colaboratory?" [Online]. Available: https://colab.research.google.com/. [Accessed: 2021]

[15] "Multi-Class Metrics: F1-Score." [Online]. Available: https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1. [Accessed: 2021]

[16] “Classification: Accuracy.” [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy. [Accessed: 2021]
[17] “TensorFlow: Categorical Crossentropy.” [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy. [Accessed: 2021]

[18] “Tensorflow: Callbacks.” [Online]. Available: https://www.tensorflow.org/guide/keras/custom_callback?hl=es. [Accessed: 2021]

[19] “Técnicas de Regularización Básicas para Redes Neuronales.” [Online]. Available: https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4. [Accessed: 2021]

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” 2014.

[21] “Precios CPU Google Cloud.” [Online]. Available: https://cloud.google.com/compute/all-pricing. [Accessed: 2021]

[22] “Precios GPU Google Cloud.” [Online]. Available: https://cloud.google.com/compute/gpus-pricing#gpus-pricing. [Accessed: 2021]