Universidad Zaragoza
1542

Trabajo Fin de Grado

DeepMyco: Aprendizaje profundo aplicado a la micología
DeepMyco: Deep Learning applied to micology

Autor

Marcos Martínez Galindo

Director/es

Dra. Dª Piedad Garrido Picazo
Dr. D. Julio A. Sangüesa Escorihuela

Escuela Universitaria Politécnica de Teruel

2019

Resumen

Uno de los mayores encantos de la provincia de Teruel es su diversidad micológica, de la que se aprovechan miles de personas cada año. Pero no todo el mundo sabe distinguir las diferentes especies de setas, algo realmente difícil. Por ello, la idea de una aplicación móvil que reconozca la especie con tan sólo tomar una fotografía es muy interesante.

Las redes neuronales están ganando mucho protagonismo en la industria dada su gran capacidad para aprender a reconocer patrones en los datos. Uno de los campos donde se utilizan esta clase de algoritmos es en la visión por computador, campo cada vez más en uso y en gran evolución. Una de las principales funciones es el reconocimiento de objetos en imágenes, o clasificación de imágenes. Con tan sólo ver la imagen, la red neuronal es capaz de reconocer los objetos que hay en ella, si ha sido bien entrenada. Sin embargo, entrenar bien una red neuronal para que reconozca objetos no es tarea fácil.

Esto, unido a la dificultad de diferenciar a simple vista una seta, hacen del reconocimiento de setas en imágenes con redes neuronales un desafío considerable. En este proyecto, se ha realizado un estudio de los principales bancos de datos, del funcionamiento de estos algoritmos y del rendimiento de algunos de los diferentes algoritmos preentrenados del mercado. Con ello, se ha conseguido desarrollar un modelo con una precisión del 82 %, diferenciando entre 7 especies de la provincia de Teruel, que ha sido embebido en una aplicación móvil desarrollada para tal fin.

Palabras claves Redes neuronales, Aprendizaje profundo, Aprendizaje transferido, reconocimiento setas, setas, micología.

Abstract

One of the greatest charms of the province of Teruel is its mycological diversity, which thousands of people take advantage of each every year. But not everyone knows to distinguish different species of mushrooms, something really difficult. Therefore, the idea of a mobile application that recognizes the species just with taking a picture is very interesting.

Neural Networks (NN) are gaining a lot of limelight in the industry thanks to their great capacity to learn pattern recognition of the data. One of the areas where this kind of algorithms are frequently used is in computer vision, field increasingly in use and evolution. The main functions are object recognition in images, or image classification. With just the image, the NN is able to recognize the objects which are in it, as long as it is well trained. However, to train well a NN to recognize objects is not an easy task.

This, attached to the difficulty of differentiating a mushroom at a glance, makes mushroom recognition by means of images a considerable challenge. In this project, it has been done an essay of the main data banks, the functionality of these algorithms and the performance of some of the different pre-trained algorithms of the market. With that, it has been developed a model with 82 % of accuracy, differentiating among 7 species of the province of Teruel, which has been embedded in a mobile application developed for such purpose.

Keywords Neural networks, Deep Learning, Transfer Learning, Mushroom recognition, Mushroom, Micology.

Tabla de contenidos

Índice de figuras

Índice de tablas

Agradecimientos

Agradezco al Centro Europeo de Empresas e Innovación de Aragón, S.A. (C.E.E.I Aragón), y en particular a D. Antonio Martínez (Gerente de Proyectos), no sólo la ayuda prestada durante el desarrollo de este proyecto a nivel de infraestructura sino por enseñarme también el enfoque empresarial, como posible emprendedor, que podría darle en el futuro. Dicha colaboración pudo llevarse a cabo gracias al convenio de prácticas académicas externas gestionado por Universa entre el C.E.E.I Aragón y la Universidad de Zaragoza (UZ).

1. Introducción

1.1. Motivación

A día de hoy la Inteligencia Artificial (IA) está en boca de todos, desde coches autonómos hasta software de detección de células cancerígenas, el mundo está lleno de aplicaciones de IA. A veces incluso sin que la gente se dé cuenta. Por ello, hay multitud de técnicas de IA diferentes, que sumadas al desconocimiento de la población, hace que la definición de IA sea muy ambigüa e incluso a veces desconcertante.

Una de las principales áreas de IA es el aprendizaje automático. El aprendizaje automático se puede entender como la capacidad de aprender de una máquina de forma autónoma. Y es en ello en lo que se va a fundamentar este trabajo de fin de grado (TFG) para resolver el problema que se presenta a continuación.

Todos los años miles de personas van a los montes para la época de recolección de setas, algo que puede ser muy peligroso si se desconocen los diferentes tipos de setas que existen, pues hay algunas muy parecidas que pueden ser deliciosas para el paladar o fatales para la salud. Es por esto que surje la necesidad de desarrollar una IA capaz de distinguir y clasificar las setas de la provincia.

Así el proyecto se centra en realizar una investigación sobre diferentes métodos de aprendizaje automático para poder realizar un clasificador con una buena precisión.

Este TFG comienza con una introducción, dónde se cuenta brevemente la historia de las redes neuronales y se explica su funcionamiento. Posteriormente se comentan los objetivos del trabajo y se hace un estudio sobre el estado del arte, tanto de la tecnología como de las soluciones al problema que ya han sido desarrolladas. Seguidamente se realiza una búsqueda de datos para poder clasificar los tipos de setas, se procesan los datos obtenidos y se pasa a estudiar y desarrollar diferentes técnicas de aprendizaje automático. Una vez implementados los diferentes algoritmos se realiza un análisis de los resultados para poder elegir la mejor solución. Con la solución elegida se desarrolla una APP móvil para el uso del sistema elegido. Finalmente se presentan unas conclusiones y el trabajo futuro a llevar a cabo para la mejora del proyecto.

1.2. Historia de las redes neuronales

Para entrar en contexto se debe conocer la historia de la IA o, al menos, de las redes neuronales, la tecnología en la que se centra el TFG.

Cuando se habla de redes neuronales se puede remontar su origen a 1943, cuando Warren McCulloch y Walter Pitts realizaron su primer trabajo sobre IA, por el que fueron considerados los autores del primer trabajo de este tema.[1]

En este trabajo propusieron un modelo que intentaba simular la estructura cerebral, en el que un conjunto de neuronas, denominado red neuronal, se conectaba e interactuaba con otra red neuronal, sugiriendo que algunas redes neuronales bien definidas podrían aprender. Para ello se basaron en la estructura cerebral y en el funcionamiento de las neuronas, así como en la lógica proposicional de Russell y Whitehead y en la teoría de la computación de Turing.[1], [2]

En el ámbito del aprendizaje automático se debe nombrar a Karl Lashley, el cuál destacó que el proceso de aprendizaje es un proceso distribuido, y no local a una determinada área del cerebro como se pensaba hasta entonces[2]. Pero por encima de Lashley se debe nombrar a un alumno suyo, Donald Hebb, quién determinó una de las reglas de aprendizaje más usadas y que sigue vigente a día de hoy; el aprendizaje hebbiano. En ella D. Hebb explicó los procesos de aprendizaje desde un punto de vista psicológico, según el cual el aprendizaje ocurría cuando se realizaban una serie de cambios en la neurona, es decir, que la conectividad del cerebro cambia en el proceso de aprendizaje y se crean asociaciones neuronales nuevas.[1], [2]

D. Hebb partió de los trabajos del español Santiago Ramón y Cajal, afirmando que, conforme se repite la activación de una neurona sobre otra a través de la sinapsis, la efectividad de ésta se incrementa.[2]

Siguiendo el estudio de McCulloch y Pitts, Rosenblatt presentó, en 1958, una aproximación al reconocimiento de patrones mediante la invención del perceptrón, una máquina capaz de aprender. El trabajo de Rosenblatt continuó en la década de 1960, primero con las contribuciones de Widrow, que presentó el sistema ADALINE con un algoritmo de aprendizaje sencillo, denominado LMS (Least Mean Square), con el que se podía aprender de forma más precisa y rápida que la de los perceptrones de aquel entonces. [1], [2], [3]

En 1969 surgió el trabajo de Minsky y Papert, *Perceptrons*, que se quedó estancado durante 10 años al ver reducidos sus fondos a prácticamente nada. En su trabajo se manifestaron las limitaciones de los perceptrones, demostrando que un perceptrón sólo puede resolver problemas linealmente separables[1], [2].

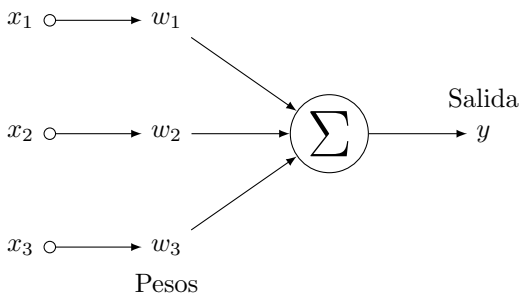
Después del libro de Minsky y Papert no hubo ningún avance significativo referente a las redes neuronales, hasta que en 1982 J. Hopfield publica un trabajo clave para el resurgimiento de las mismas. En él, desarrolla la idea del uso de una función de energía para comprender la dinámica de una red neuronal recurrente con uniones sinápticas simétricas, permitiendo sólo salidas bipolares (0 ó 1)[2].

Junto al trabajo de Hopfield el más importante y que más repercusión tuvo para la "resurrección" de las redes neuronales fue el de Rumelhart, Hinton y Williams en 1986. En él, toman de partida el trabajo de Paul Werbos de 1974 y desarrollan el algoritmo de aprendizaje de retropropagación (back-propagation) para redes neuronales multicapa. A partir de 1986 el número de trabajos sobre redes neuronales ha aumentado exponencialmente y a día de hoy las redes neuronales son un mecanismo de IA bastante utilizado.[2]

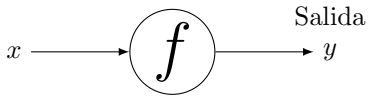
1.3. Funcionamiento de las redes neuronales

Una red neuronal es, como su nombre indica, una red de neuronas. Por ello si se quiere conocer el funcionamiento de una red neuronal primero se debe conocer el funcionamiento de una neurona.

1.3.1. Funcionamiento de una neurona



(a) Perceptrón de Minsky y Papert



(b) Función matemática

Figura 1: Comparación del perceptrón y una función.

Una neurona es la evolución del perceptrón desarrollado por Rosenblatt y, al igual que él, se puede entender como una función matemática. Como se puede apreciar en las figuras 1a y 1b, el perceptrón y una función matemática son muy parecidos, prácticamente no se nota diferencia alguna, ambas reciben varios datos de entrada y tras una serie de operaciones devuelven un valor de salida.

Esto es muy similar al funcionamiento del cerebro. Una neurona recibe varios estímulos procedentes de otras neuronas mediante sus dentritas y, después de una serie de reacciones químicas, emite una señal por su axoma, que a su vez reciben otras neuronas por sus dentritas.

Siguiendo con el perceptrón, la única diferencia observable es que además de las entradas y las salidas hay algunas otras variables que no están en la función matemática. En cada entrada se puede ver una variable ω, conocida como peso, más adelante se verá para qué sirve.

Como se ha dicho anteriormente una neurona es la evolución del perceptrón, cuya estructura no difiere mucho de la del perceptrón y puede observarse en la Figura 1.

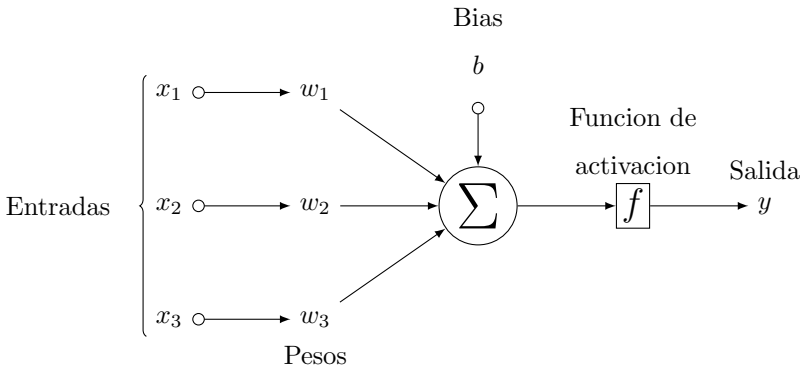


Figura 2: Neurona artificial

Como se observa en la figura 2, además de los pesos que se habían visto en el perceptrón, se ven componentes nuevos. Por un lado se observan dos funciones, la función de activación (f) y una función de suma (Σ) y, por otro lado, se tiene una nueva entrada, conocida como *bias*. Todos estos componentes son los componentes básicos de una neurona.

Básicamente se puede decir que una función es un modelo de regresión lineal, es decir, una suma ponderada de las entradas sumado a la bias. Si a la salida de esta suma ponderada se le aplica la función de activación se tendrá la salida de la neurona. Hay diversos tipos de funciones de activación (sigmoide, tanh...) que se verán más adelante.

Escrito en términos matemáticos, una neurona podría verse representada de la siguiente forma:

$$y = f(\sum_{i=1}^{n} x_i \omega_i + b) \tag{1}$$

Pero entonces, ¿qué diferencia a la neurona de una función matemática? La respuesta, como se ha dicho al principio del trabajo, es la capacidad de aprender. Cuando se entrena la red, o la neurona en este caso, se busca una combinación de pesos y de la bias para que dadas unas entradas la salida sea la esperada. Conforme se entrena la red, se van ajustando los valores de los pesos y de la bias y, al final del entrenamiento, se tendrá la combinación que más se ajusta para resolver el problema para el que se ha entrenado.

Se pueden intentar solucionar problemas aritmético-lógicos básicos con una neurona, como por ejemplo, aislar los valores de las puertas *AND*, *OR* y *XOR*. Se van a resolver estos problemas para poder entender la finalidad de una neurona. Se va a suponer que la función de activación es una función escalonada. Esta función devolverá 1 si supera un umbral determinado y 0 en caso contrario, en el ejemplo siguiente se supone que ese umbral es 0, de forma que, si el resultado es positivo devolverá 1.

$$f(output) = \begin{cases} 1, & \text{if } output \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Recordando que:

$$output = \sum_{i=1}^{n} x_i \omega_i$$

Los problemas a resolver son los siguientes:

- Puerta *AND*:

$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{y}$
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 1: Tabla de verdad puerta AND

Una posible solución sería: $\omega_1 = 4$, $\omega_2 = 5$ y $b = -6$.

$$output = 4x_1 + 5x_2 + -6$$

Si se resuelve con los diferentes valores se verá que cumple con la tabla de verdad.

- Puerta *OR*:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 2: Tabla de verdad puerta OR

Para la puerta *OR* se puede encontrar la siguiente combinación: $w_1 = 3$, $w_2 = 3$ y $b = -2$

$$output = 3x_1 + 3x_2 + -2$$

- Puerta *XOR*:

Por último la puerta *XOR*. No se puede encontrar ninguna combinación que satisfaga la tabla de

x_1	x_2	y
0	0	1
0	1	0
1	0	0
1	1	1

Tabla 3: Tabla de verdad puerta XOR

verdad. Esto es porque la *XOR* no es un problema linealmente separable, es decir, no se puede encontrar una línea recta que separe los valores positivos de los negativos en un gráfico.

Por ejemplo en la puerta *AND* se pueden encontrar muchas rectas, un ejemplo se puede ver en la Figura 3. De la misma forma ocurre con la OR, figura 4.

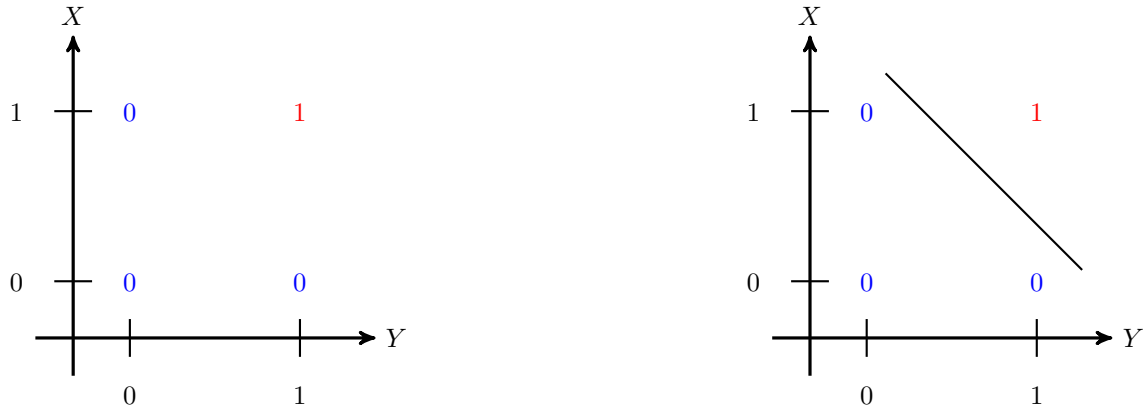


Figura 3: Puerta AND.

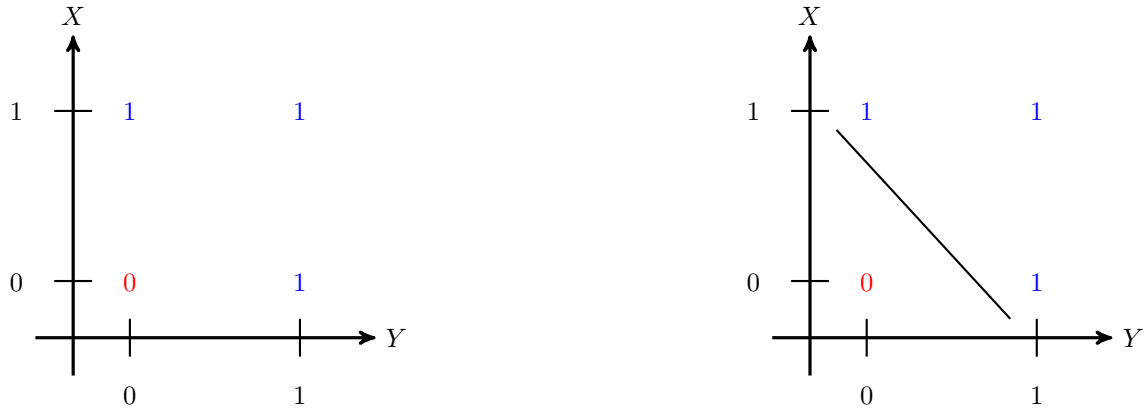


Figura 4: Puerta OR.

Pero no se puede encontrar una línea recta que separe los 1 de los 0 en el problema de la XOR, Figura 5.

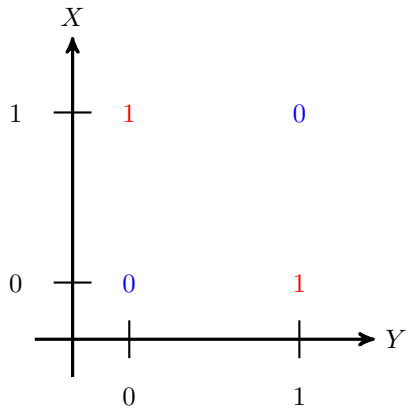


Figura 5: Puerta XOR

1.3.2. Conectando neuronas

Como se ha visto, con una neurona no se puede resolver el problema de la XOR. Éste es el problema que apuntaron Minsky y Papert en su trabajo *Perceptrons* y que dejó sin financiación al mundo de las redes neuronales durante décadas.

Para solucionarlo basta con añadir otra neurona. Una se encargará de separar linealmente la puerta *OR* y otra la puerta *AND*.

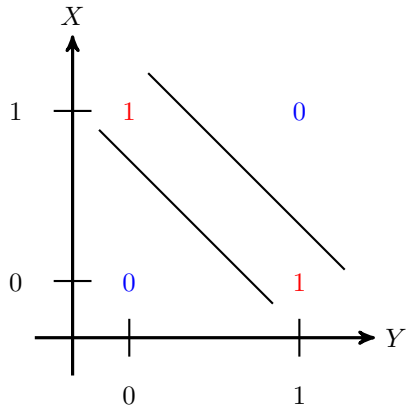


Figura 6: Puerta XOR separada linealmente

Así uniendo dos neuronas se consiguió resolver el problema, lo que indica que uniendo neuronas se pueden resolver problemas más complejos.

Para tratar todas las neuronas, las redes neuronales se organizan en capas. Una capa es un conjunto de neuronas que no se relacionan entre sí, y en la que pueden haber tantas neuronas como se quiera. Como mínimo tiene que haber dos capas en una red neuronal, una de entrada y una de salida, pero se pueden añadir entre medio tantas capas como sea necesario. Estas capas intermedias se conocen como capas ocultas. Las salidas de las neuronas de una capa se transmiten como entradas a las neuronas de la capa siguiente, cada una con sus pesos y bias. En la figura 7 se puede ver la estructura de una red neuronal. En ella se puede observar como hay varias capas y cada una de ellas tiene distinto número de neuronas.

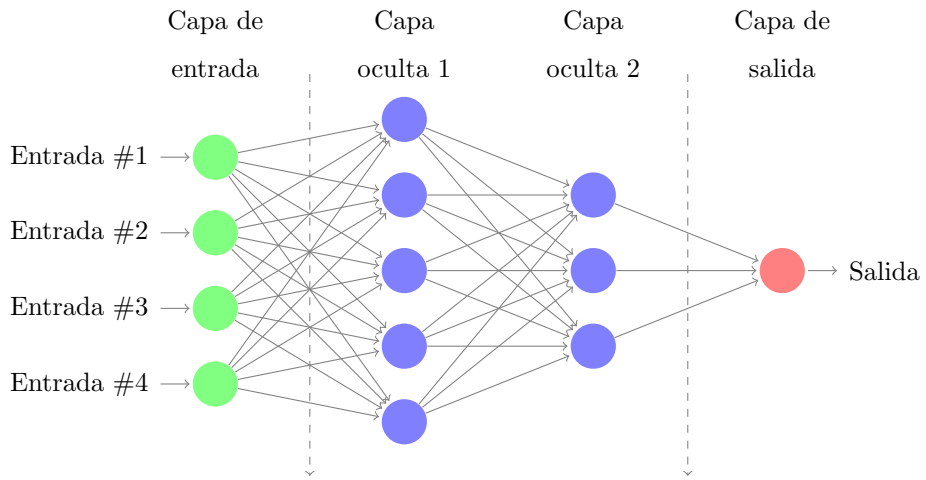


Figura 7: Estructura de una red neuronal.

Hay que destacar que cuántas más capas y más neuronas tenga la red, no siempre será mejor el resultado, aunque sí que será mayor el tiempo de entrenamiento y de ejecución, por lo que hay que buscar un punto medio, donde se obtengan buenos resultados en un tiempo aceptable, teniendo en cuenta además los recursos de los que se dispone.

Algo que sí se puede saber es cuántas neuronas deben tener la primera y la última capa, es decir, la capa de entrada y la capa de salida. La capa de entrada tendrá que tener tantas neuronas como variables tenga el problema y la de salida tantas neuronas como soluciones tenga el problema. En el ejemplo de las puertas lógicas harán falta dos neuronas en la capa de salida, una para cada variable,

y una neurona en la capa de salida, para el resultado. Se pueden poner más capas y neuronas entre medio, aunque a veces es inútil poner entre medio muchas capas ocultas o muchas neuronas, ya que la única diferencia será el gasto de recursos y tiempo.

Así pues una red neuronal aprende actualizando sus pesos y bias, buscando la mejor combinación para que, dada una entrada, la salida sea la esperada. Para buscar esa combinación primero se busca una función que cuantifique el resultado, generalmente teniendo en cuenta la distancia de la salida esperada a la salida real de la red. A esto se le conoce como la función de coste o función de pérdida. Más adelante se verá cómo seleccionar una función de pérdida.

Esta función nos dice el coste dado unos pesos ω y unas bias b. El número de entradas viene definido por n, el vector que contiene las entradas por x y a es el vector con las salidas de la red dadas las entradas. y es la propia red.

Para minimizar el coste se pueden usar distintos algoritmos de optimización, como por ejemplo el descenso de gradiente. Estos algoritmos de optimización tratan de buscar el punto mínimo de una función, aplicado a la función de coste, buscando minimizar el error de la red. No se va a entrar en detalles matemáticos de cómo funcionan estos algoritmos, puesto que no es el objetivo de este TFG.

Así el entrenamiendo de la red neuronal consistirá en ir variando los pesos y bias siguiendo un algoritmo de optimización, de manera que se minimice la función de coste, es decir, el error de la red. Pero la actualización de los pesos puede ser lenta y costosa, pues hay que volver a recorrer toda la red neuronal, y no siempre sale bien. Es por esto que además de por las limitaciones de hardware, las redes neuronales años atrás no constaban de muchas capas, por el coste de entrenamiento.

1.3.3. Backpropagation

Como se ha visto en la sección anterior, Rumelhart, Hinton y Williams en 1986 desarrollaron el algoritmo *Backpropagation* o retropropagación, un algoritmo de aprendizaje para redes neuronales multicapa. Este algoritmo se basa en la idea de "volver hacia atrás" en el aprendizaje. Es decir, en lugar de actualizar todos los pesos y bias, cada iteración, el algoritmo vuelve hacia atrás intentando entender qué neuronas han tenido más peso a la hora de decidir la salida. De esta forma el algoritmo no recorre ni actualiza todos los pesos de la red, sino que sólo actualiza los pesos y bias que han tenido importancia en la salida obtenida.

Para lograrlo el algoritmo de *backpropagation* computa no sólo el error de la red, sino también el error δ, un error intermedio que se computa en cada neurona de cada capa, para poder relacionarlo

después con el coste de la red.

El modo de proceder del algoritmo *backpropagation* es el siguiente:

- Para cada capa (l) calcula z, que viene dada por:

$$z^l = w^l a^{l-1} + b^l \tag{2}$$

  w^l son los pesos de la capa.

  a^{l-1} es la activación de la capa anterior.

  b^l es la bias de la capa. Y actualiza la activación calculando la función sigmoide:

$$a^l = \sigma(z^l) \tag{3}$$

- Después calcula el error delta de la última capa:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{4}$$

  Donde L es la última capa, σ es la función sigmoide y C es el coste.

- Por último, "vuelve hacia atrás" con el error, calculando los errores delta de las demás capas.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{5}$$

  Donde $l = L-1, L-2, \ldots, 2$

Una vez haya llegado a la primera capa ya habría acabado el algoritmo y el gradiente de la función de coste vendría dado por:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{6}$$

y por:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{7}$$

Así es como calcula el algoritmo de *backpropagation* el gradiente de la función de pérdida para un ejemplo de entrenamiento, pero en la práctica se suelen tomar grupos de entradas de entrenamiento para calcular el gradiente de la función de pérdida. Para ello aparecen nuevos algoritmos, algunos modificaciones de algoritmos que ya se han visto, como por ejemplo el descenso de gradiente estocástico (en inglés Stocasthic Gradient Descent (SGD)).

2. Objetivos

Como se ha dicho anteriormente, el objetivo de este TFG es realizar una aplicación móvil capaz de distinguir entre algunas especies de setas de la provincia de Teruel, así como el análisis y la comparación de distintas técnicas de aprendizaje automático, como el aprendizaje profundo o el aprendizaje transferido.

Por ello primero se va a desarrollar una red neuronal sin uso de ninguna librería externa. Después se verá como construir un clasificador de aprendizaje automático con librerías externas, aprendizaje profundo y con aprendizaje transferido.

Por último, se compararán los resultados de los modelos obtenidos. La lista completa de los modelos a desarrollar/comparar:

- Modelo desarrollado sin uso de librerías externas.
- Modelo desarrollado con uso de *Sklearn*.
- Modelo desarrollado con uso de *Keras*.
- Modelo de *Transfer-Learning* con diferentes modelos pre-entrenados, (*VGG*, *MobileNet*, *ResNet50*, *InceptionV3*)

Tanto el modelo desarrollado sin uso de librerías externas, como el desarrollado con *sklearn* y con *keras* contarán únicamente con capas densas para poder realizar una comparación entre ellos. Debido al coste computacional solo tendrán una capa oculta con 15 neuronas. La comparación incluirá: precisión, tiempo de entrenamiento, tamaño del modelo (guardado en disco).

Por otro lado se compararán los modelos de aprendizaje transferido, *Transfer-Learning*, con los que sí que se espera conseguir un buen resultado. Para estos modelos se usará un clasificador que conste de:

- Capa oculta de 256 neuronas
- 3 capas ocultas de 100 neuronas cada una
- Capa oculta de 7 neuronas

La comparación será igual a la vista anteriormente, pero teniendo en cuenta que el tiempo de entrenamiento no será de toda la red, pues se reutilizarán los pesos de las capas ya entrenadas y no se re-entrenarán.

3. Estado del arte

3.1. Clasificación de imágenes

El tratamiento con imágenes es muy costoso computacionalmente, por lo que no ha sido hasta las últimas décadas cuando se han visto importantes avances en este campo.

El incremento de la capacidad de cómputo, así como de la calidad de las cámaras, ha hecho posible el desarrollo de arquitecturas y algoritmos para el reconocimiento o la clasificación de imágenes, como las redes neuronales convolucionales.

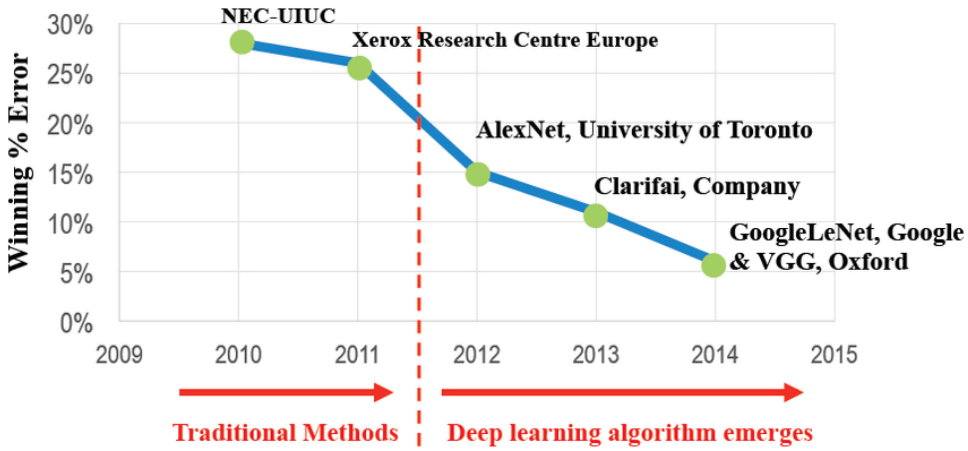


Figura 8: Resultados de la competición de ImageNet.[12]

Desde 2010 se celebra cada año la competición *ILSVRC* (*ImageNet Large Scale Visual Recognition Challenge*[8]). En esta competición los participantes tratan de reconocer los objetos que aparecen en la base de datos llamada *ImageNet*, compuesta por miles de categorías e imágenes. En el año 2012 una red neuronal convolucional, *AlexNet*[9], consiguió en la competición un error de 15,3 %, superando por un 10,8 % a sus competidores. Esto fue posible gracias al uso de *GPUs*, que permitieron entrenar un modelo tan costoso computacionalmente como lo es una red neuronal convolucional.

Esto abrió las puertas a que otros modelos convolucionales apareciesen en la competición, mejorando exponencialmente los resultados.

Pero lo realmente importante de esta competición fue la aparición de arquitecturas y modelos ya entrenados, que aprendieron a reconocer formas, patrones o colores en las imágenes, características que, sin importar el problema, son muy útiles a la hora de reconocer objetos en imágenes.

La liberación de estos modelos ya entrenados hace que sea menos costoso entrenar una red neuronal convolucional, surgiendo así el concepto de *Transfer Learning*. Por ejemplo, más adelante se usarán modelos de *Transfer Learning* entrenados con las imágenes de *ImageNet*.

Gracias a esto han aparecido multitud de aplicaciones para el reconocimiento de objetos y la clasificación de imágenes, como reconocimiento de matrículas, caras o cualquier otro objeto.

3.2. Reconocimiento de setas

En Internet se pueden encontrar múltiples APPs para el reconocimiento de setas. Algunas de ellas son:

Nombre	**Técnicas usadas**	**Requiere Internet**	**Tiempo de ejecución**	**Almacenamiento**	**Sistemas operativos**
Boletus	Claves dicotómicas o IA (No funciona)	Sí	Bajo	No	Android/iOS
Géneros de setas	Claves dicótomicas	No	Bajo	No	Android
Setas	Claves dicotómicas	No	Bajo	Sí	Android/iOS
Identificador de setas	IA con claves dicotómicas	Sí	Medio	No	Android/iOS

Figura 9: Comparativa de aplicaciones del mercado.

Estas son algunas de las aplicaciones que hay disponibles en el mercado para el reconocimiento de setas pero, a excepción de la última, ninguna de ellas utiliza técnicas de visión por computador para reconocer la especie a partir de una foto.

Además, la mayoría requieren de conexión a Internet o de almacenamiento de fotos en el dispositivo para hacer posible la identificación, algo que especialmente en los dispositivos móviles puede llegar a ser un problema dada la escasa capacidad de almacenamiento y el entorno en el que se encuentran las setas.

Por ello, las técnicas de *DeepLearning* son una buena opción, ya que no requieren de conexión a Internet, algo que es muy común en las salidas al campo, no requieren de almacenamiento de fotografías y clasifican la especie únicamente con la fotografía, sin necesidad de la experiencia humana. Por lo que una aplicación de este tipo resulta atractiva para aquellos usuarios inexpertos o principantes a los que les cuesta seguir una clave dicotómica.

4. Metodología

4.1. Obtención del dataset

Para poder comenzar el proyecto hay que encontrar un *dataset* (almacén de datos) que sirva para tal propósito. La elección de un *dataset* adecuado es muy importante, pues cuanto mayor y mejor sea, mejor será el clasificador. Además, a medida que aumenta el tamaño del *dataset* aumenta el tiempo de procesamiento, algo que hay que tener en cuenta a la hora de hacer la elección, teniendo presente los recursos de *hardware* disponibles.

El TFG va dirigido a clasificar distintos tipos de setas dada una imagen, que puede ser captada por el usuario mediante una aplicación móvil, por lo que se necesitará un *dataset* de imágenes de distintos tipos de setas. En Internet hay disponibles numerosos bancos de *datasets*, por ejemplo: *Kaggle*, *Google Dataset Search*, *UCI Machine Learning Repository* y muchos más. También se puede crear un *dataset* propio con imágenes de la Web, aunque teniendo muy en cuenta los derechos de autor.

Para el proyecto se usó un *dataset* obtenido del *FGVC5* (*Fifth Fine-Grained Visual Categorization*) del *CVPR 2018* (*Computer Vision and Pattern Recognition*[7]), una de las conferencias mundiales sobre visión por computador más importantes a nivel internacional. El *dataset* fue provisto por Svampe Atlas[5] y cuenta con casi 1.500 especies de setas diferentes con más de 85.000 fotos en total. Con este *dataset* se lanzó una competición de *Kaggle* a través de la cuál se accedió a los datos. En *Kaggle*, además de tener disponibles múltiples *datasets*, se puede encontrar código relacionado con la IA para poder usar de ejemplo y se organizan competiciones en las que el ganador puede llegar a llevarse cuantiosas sumas, dependiendo del patrocinador de la competición.

El *dataset* fue descargado del repositorio de *GitHub*, dónde se especifica que usa una licencia *MIT* (*Massachusetts Institute of Technology*)[6]. Además en cada imagen se muestran la licencia y el autor.

El *dataset* no cuenta sólo con las imágenes, sino que además incorpora algunos datos como las medidas de la imagen, la especie o la licencia. Todo esto viene dado en formato *COCO*, un formato para etiquetar imágenes. El formato viene en forma de *json*, que contiene los siguientes elementos:

- Anotaciones:

  Contiene el *id* de cada imagen y el *id* de la categoría a la que corresponde.

- Categorías:
  Contiene el nombre de la categoría y el nombre de la "super-categoría", correspondiente al género.
- Información:
  Contiene información sobre el *dataset*, como el año, versión, fecha de creación, autor, etc.
- Licencias:
  Contiene información sobre las licencias.
- Imágenes:
  Contiene información sobre la imagen, como puede ser el fichero, el tamaño, el tipo de licencia, el autor o poseedor de los derechos, etc.

Este *dataset* cuenta con cerca de 1.500 especies de setas y hongos, todas ellas fotografiadas en Dinamarca. Pero para el proyecto, sólo se tendrán en cuenta una muestra de las disponibles en la provincia de Teruel.

De cada especie se dispone de un número diferente de imágenes:

- *Agaricus arvensis*, Figura 10a: 228 imágenes
- *Agaricus xanthodermus*, Figura 10b: 137 imágenes
- *Amanita muscaria*, Figura 10c: 163 imágenes
- *Amanita phalloides*, Figura 10d: 38 imágenes
- *Boletus edulis*, Figura 10e: 81 imágenes
- *Boletus pinophilus*, Figura 10f: 15 imágenes
- *Calocybe gambosa*, Figura 10g: 122 imágenes
- *Lactarius chrysorrheus*, Figura 10h: 20 imágenes
- *Lactarius deliciosus*, Figura 10i: 61 imágenes
- *Marasmius oreades*, Figura 10j: 117 imágenes

(a) Agaricus Arvensis

(b) Agaricus Xanthodermus

(c) Amanita Muscaria

(d) Amanita Phalloides

(e) Boletus Edulis

(f) Boletus Pinophilus

(g) calocybe Gambosa

(h) Lactarius Chrysorrheus

(i) Lactarius Deliciosus

(j) Mariasmus Oreades

Como se ha dicho, el *dataset* a utilizar es reducido, contando con pocos ejemplos por cada clase, entendiendo por clase especies diferentes. Además es un *dataset* desbalanceado, teniendo más ejem-

plos de algunas clases que de otras, con una diferencia notable. Por ello el *dataset* no es adecuado para desarrollar un clasificador de gran precisión mediante el uso de redes neuronales simples, pero una vez más, servirá para el estudio.

4.2. Preprocesamiento de los datos

Una vez se tenga el *dataset* descargado hay que tratar los datos para empezar a trabajar con ellos.

Para empezar hay que conseguir cargar las imágenes con sus respectivas clases, pues como se ha dicho anteriormente el *dataset* viene en formato COCO. Así pues lo primero que se hará será seleccionar todas las imágenes disponibles de las especies arriba mencionadas, extraerlas a una carpeta y crear un *dataset* en el que aparezca la ruta de la imagen y su clase.

Una vez se tenga el *dataset* con las imágenes y sus etiquetas se tienen que hacer algunas modificaciones, pero primero se visualizarán las imágenes a ver si hay alguna que no se corresponda con su etiqueta para evitar la inconsistencia del *dataset*. Tratando las imágenes adecuadamente se puede mejorar la calidad del *dataset*. Por ejemplo la Figura 11a pertenece al *dataset* y es de baja calidad para la clasificación: En cambio tras un buen tratamiento de las imágenes se pueden conseguir

(a) Ejemplo de imagen del dataset.

(b) Ejemplo de imagen adecuada.

Figura 11: Comparación de imágenes.

extraer imágenes como la Figura 11b, que sí que son adecuadas para la clasificación de imágenes.

Una vez se haya "limpiado" el *dataset*, en caso de que fuese necesario, hay que observar el tamaño de las imágenes. Como se ha explicado anteriormente, las entradas de la red neuronal serán los píxeles de la imagen, por lo que todas las imágenes deben de tener el mismo número de píxeles tanto

por alto como por ancho. En el trabajo se van a reducir todas a $224px * 224px$, un tamaño aceptable que hace que el número de entradas no sea muy alto, 150.528 ($224px * 224px * 3canales(r, g, b)$).

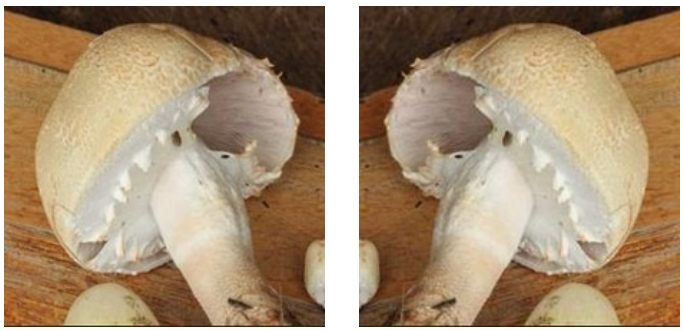
Para poder entrenar una red neuronal con éxito se necesitan muchos ejemplos, por lo que este *dataset* no es muy adecuado para ello. Sin embargo existen algunas formas de ampliar un *dataset* de imágenes. Estas técnicas se conocen como *Data Augmentation*.

Hay muchas formas diferentes de hacer *Data Dugmentation*, algunas de ellas son[4]:

- Inversión, consiste en invertir la imagen, ya sea en horizontal o en vertical. Se pueden aplicar ambas según el caso, aunque puede que sea contraproducente.
- Rotación, como su propio nombre indica, consiste en rotar la imagen hacia uno u otro lado. Se pueden obtener varias imágenes según el grado de rotación.
- Escalado, consiste en escalar la imagen aumentando o disminuyendo su tamaño.
- Recorte, recortar una parte de la imagen y, posteriormente, aumentar el recorte al tamaño original.

A pesar de que hay muchos tipos, no se pueden utilizar siempre todos, hay que saber cuáles de ellos son los más adecuados para según qué problema. Para el caso del proyecto, reconocimiento de setas, se puede usar la técninca de voltear las imágenes sobre su eje vertical, pero no se podrá hacer lo mismo sobre su eje horizontal, ya que no se va a dar el caso de que las setas crezcan con el tallo hacia arriba, y eso podría llevar a la red a ser entrenada incorrectamente.

Así pues en este TFG se usará inversión, invertiendo las imágenes sobre su eje vertical.



(a) Imagen original (b) Imagen invertida

Figura 12: Ejemplo de inversión

Una vez estén disponibles todas las imágenes a usar, incluidas las obtenidas con *Data Augmentation*, se debe preparar el *dataset*, dividiendo los ejemplos disponibles en imágenes de entrenamiento y en imágenes de validación. Para hacerlo de forma proporcionada se dividirá cada clase por separado y se juntarán posteriormente, ya que debido a la diferencia en el número de ejemplos de cada clase se podría dar la posibilidad, a la hora de dividir el *dataset*, que de alguna clase no hubiese ejemplos.

Dado el escaso número de ejemplos de algunas clases, se ha decidido eliminarlas del clasificador para no afectar al rendimiento. Las clases que se eliminarán son:

- Lactarius chrysorrheus
- Lactarius deliciosus
- Boletus pinophilus

(a) *Lactarius Chrysorrheus* (b) Lactarius Deliciosus (c) *Boletus pinophilus*

Figura 13: Ejemplos de clases eliminadas

Después de dividirlo, se quedan 1.810 ejemplos de entrenamiento y 322 de validación o testing. Para terminar, se clasificarán las *features* (valores de cada píxel) de las imágenes y su etiqueta, en diferentes archivos para poder tratarlos por separado posteriormente.

4.3. Creación de una red neuronal

Antes de desarrollar la red neuronal habrá que implementar algunas funciones a utilizar, como por ejemplo las funciones de activación o la función de pérdida. Como función de activación podemos usar las siguientes:

- Sigmoide ($sigm$):

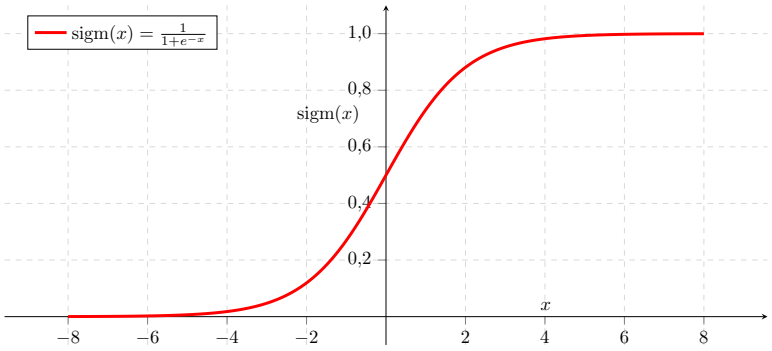


Figura 14: Función sigmoide

- Rectilineamente uniforme ($ReLu$):

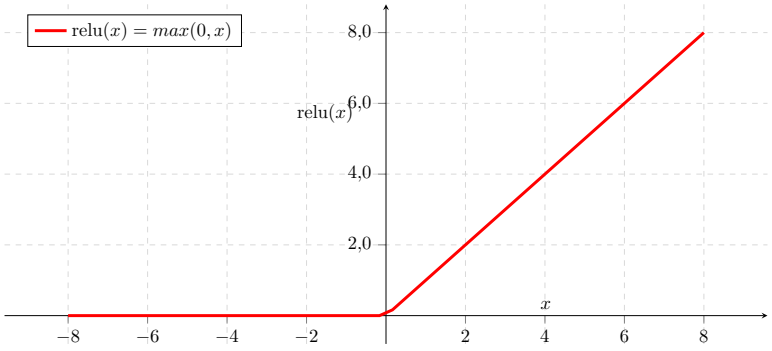


Figura 15: Función ReLu

- Tangente hiperbólica (tanh):

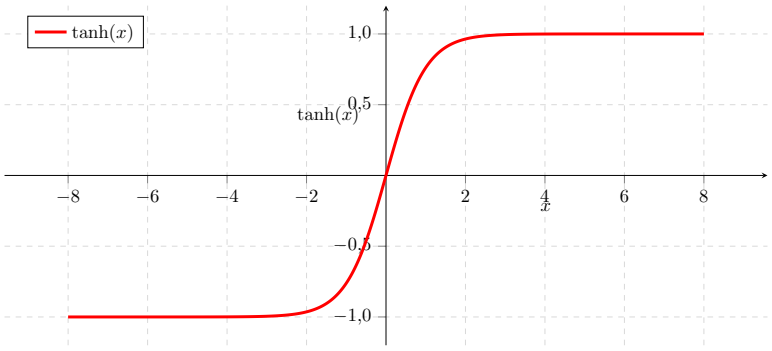


Figura 16: Función tanh

A pesar de que hay múltiples funciones de pérdida se implementarán dos:

- Error cuadrático medio (En inglés, Mean Square Error (MSE):

$$MSE(y1, y2) = \frac{1}{n}\sum_{t=1}^{n}(y1 - y2)_t^2 \tag{8}$$

- Cross-entropy:

$$H(y1, y2) = E_y 1[-\log y2] \tag{9}$$

Aunque lo mejor es realizar pruebas con varias funciones para ver el resultado, en general se utiliza la función *MSE* para problemas de regresión, mientras que *Cross-entropy* se suele utilizar para problemas de clasificación.

Una vez se hayan desarrollado las funciones, lo siguiente será la estructura de la red neuronal. Se puede entender la red como una lista de capas, donde cada capa es una lista de neuronas. Cada neurona se implementará como un diccionario, donde se guardarán los pesos, la bias, la función de activación de la neurona, la salida de la neurona y el error δ, calculado con el algoritmo *backpropagation*. Aunque podría buscarse alguna forma más eficiente de implementar la red, como una matriz de pesos. Esta metodología permite almacenar de forma sencilla valores que serán útiles *a posteriori*, como el error δ o z.

Los parámetros de entrada de la red serán los siguientes:

- Lista de número de neuronas por capa.
  El primer parámetro será una lista de enteros, en la que cada entero corresponde al número de neuronas que tendrá dicha capa. El primer entero deberá ser igual al número de entradas que tiene la red y el último al número de salidas.

- Funciones de activación.
  Por defecto se utilizará la función *sigmoide* para todas las neuronas, pero se pueden especificar otras de las vistas anteriormente. Si sólo se envía una de ellas, se utilizará para todas las capas. Si se quieren utilizar diferentes funciones de activación se puede pasar como parámetro una lista de funciones de activación, dónde cada función de activación corresponderá a la capa del mismo índice.

- Función de pérdida.
  También se podrá especificar la función de pérdida a optimizar. Por defecto se utilizará la función *mean square error*. Los valores posibles son: *mse* y *cross entropy*.

Con estos parámetros se creará el objeto de la red neuronal, que guardará la lista de capas y las funciones de activación elegidas. Las neuronas guardarán solamente el nombre de la función, mientras que la red guardará las funciones implementadas.

Para comprobar que se creaba la red correctamente se implementó un método de reporte, que recorre la red y muestra por pantalla las capas, neuronas y funciones de activación de cada capa.

El siguiente método a desarrollar es el *feedforward*. Este método es el que recorre la red neuronal realizando las sumas ponderadas y transmitiendo las salidas de las neuronas a la capa siguiente. Como parámetros necesita las entradas y, aunque se podría haber implementado que devolviese la salida de la red, se optó por que no devolviese nada y guardase en las neuronas los resultados de salida. Gracias a la solución de desarrollo elegida, implementar el método es muy sencillo, al igual que guardar los resultados. En el diccionario de la neurona se guardará el resultado (*output*) y z, que como se vio anteriormente viene dada por:

$$z^l = w^l a^{l-1} + b^l \tag{10}$$

Para comprobar el resultado, una vez más, se implementó un método de reporte más completo que, además de mostrar el número de neuronas y las funciones de activación, enseña los valores de los pesos, las bias y las salidas de las neuronas, para poder comprobar el resultado. Aparte se

implementó un método para obtener la salida de la red que devuelve las salidas de las neuronas de la última capa. Con el método *feedforward* y el método para obtener la salida, *get_output* ya se pueden realizar predicciones con la red, pues el proceso de predecir es realizar el *feedforward* y obtener la salida. También se realizó un método de reporte de la salida y de la última capa para visualizar más facilmente los resultados de la red.

Una vez desarrollado el método *feedforward*, el siguiente paso es implementar el algoritmo de *backpropagation*. Como se ha visto anteriormente para ello hay que calcular el error δ, empezando por la última capa y "volver hacia atrás". Para ello se recorrerán las capas de la red y se calculará el error siguiendo las fórmulas vistas anteriormente, una vez más, gracias a la implementación de la red neuronal a base de listas de diccionarios, será muy fácil guardar el error δ o utilizar el valor z de cada neurona, que ya se ha obtenido y guardado en el método *feedforward*.

Para terminar hay que implementar el algoritmo de optimización, que tratará de optimizar la función de coste variando el valor de los pesos y las *bias*. El algoritmo que se implementará será el descenso del gradiente estocástico, *SGD* por sus siglas en inglés, *Stocasthic Gradient Descent*.

Con ello ya se tendría la red neuronal implementada. Además de estos métodos también se implementaron algunos para guardar la red neuronal a un archivo, leer una red neuronal de un archivo o un método para predecir, que como se ha dicho anteriormente no es más que una llamada al método *feedforward* y obtener la salida con el método *get_output*.

Para probarla se testeará con dos *datasets* de la literatura para problemas de clasificación multiclase, el *Iris dataset* y el *MNIST*[1], ambos obtenidos a través de la librería *sklearn*, que se verá en detalle más adelante. Se usó una capa oculta de 100 neuronas, con la función *sigmoide* como función de activación y *Cross-Entropy* como función de coste o de pérdida. Tras 1000 iteraciones se obtuvo una precisión de 96,6 % en el *Iris dataset* y tras 100 iteraciones se obtuvo una precisión de 70,8 % en el *MNIST*[2].

Para el *dataset* micológico se utilizará una única capa oculta de 15 neuronas, con la función *sigmoide* como función de activación en todas las capas y *Mean Square Error* como función de pérdida. Tras 3 iteraciones se consigue un 13,7 % de precisión, muy bajo y debido, como se ve en la matriz de confusión, Figura 17, a que la red está prediciendo la misma clase para todos los ejemplos.

[1] El *Iris dataset* es un dataset que consta de datos tabulares extraídos de tres especies de *Iris*. El *MNIST* se compone de imágenes de números manuscritos.

[2] Ver Anexo.

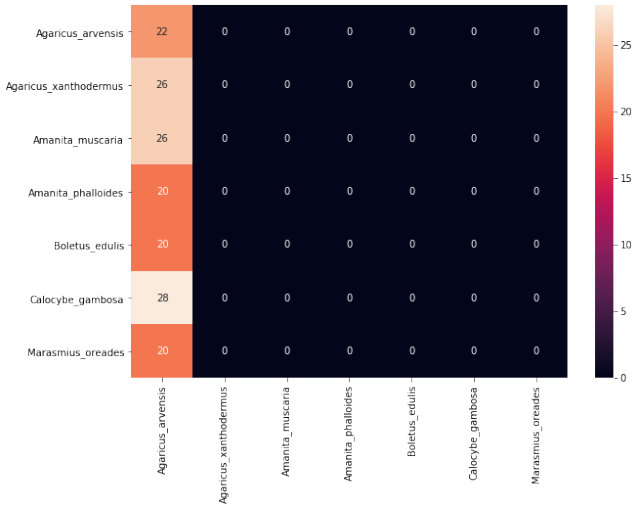


Figura 17: Matriz de confusión

4.4. Aprendizaje automático - Sklearn

La librería *scikit-learn*[10] es una de las librerías más completas y potentes disponibles y, por ende, una de la más usadas en el ámbito de la IA.

La librería, conocida generalmente como *sklearn*, ofrece un sinfín de utilidades para los científicos de datos, desde multitud de implementaciones de algoritmos hasta *datasets* o funcionalidades para el tratamiento de datos. Por ejemplo, se pueden encontrar *datasets* famosos como el de *MNIST* o el *iris dataset*, funcionalidades para dividir los datos en datos de entrenamiento y de test, algoritmos de *clustering*, de reducción de dimensiones, de *machine learning*, etc, y, por supuesto, ofrece una implementación de redes neuronales, bajo el nombre de *Multi-layer Perceptron*, esto es, perceptron multicapa, pues como se ha dicho en las primeras secciones una red neuronal densa es la unión de capas de neuronas o perceptrones.

Para hacer pruebas del funcionamiento de la librería se utilizarán algunas funciones antes de llevar a cabo una implementación concreta para el *dataset* de las setas. Para las pruebas se usarán las siguientes funcionalidades:

- *from sklearn import svm*: una implementación de las *SVM* (*Support Vector Machine*), aplicada al problema de clasificación, bajo el nombre de *SVC* (*Support Vector Classifier*).
- *from sklearn import metrics*: funcionalidad muy útil para conocer el rendimiento de los algoritmos. Ofrece matrices de confusión, reportes del rendimiento, etc.

- *from sklearn.neural_network import MLPClassifier*: implementación de una red neuronal de *sklearn* para clasificación, *MLPClassifier* (*Multi-layer Perceptron Classifier*).
- *from sklearn.model_selection import train_test_split*: funcionalidad para dividir un *dataset* en datos de entrenamiento y datos de validación o testing.
- *from sklearn.datasets import load_digits*: *dataset* conocido como *MNIST*. Conjunto de imágenes de números del 1 al 9.

Lo primero será cargar el *dataset* de *MNIST* con la función *load_digits*. Después se dividirá el *dataset*

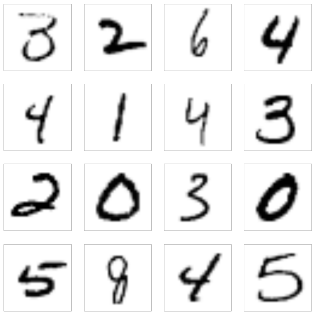


Figura 18: Ejemplo de *MNIST*

en datos de entrenamiento y de testing con la función *train_test_split*. El *dataset* de *MNIST* una vez dividido tiene la siguiente forma:

- X_train (1437, 64)
- y_train (1437,)
- X_test (360, 64)
- y_test (360,)

Esto significa que se tienen 1437 ejemplos de entrenamiento y 360 para test. Por último se entrenará un modelo de *SVC* y un modelo de *MLP*. Los parámetros a utilizar serán los siguientes:

- Parámetros del *SVC*:
  $\gamma = 0{,}001$
- Parámetros del *MLP*:
  Como algoritmo de optimización se utilizará *SGD*, $\alpha = 1e-5$, con una capa oculta de 15 neuronas.

Algoritmo	**Precisión en entrenamiento**	**Precisión en *testing***
SVC	99,79 %	98,88 %
MLPCLassifier	97,21 %	95,27 %

Tabla 4: Resultados *MNIST*

Los resultados obtenidos figuran en la tabla 4.

Como se puede observar, los resultados son algo mejores con un modelo *SVC*. Aunque ambos son algoritmos de aprendizaje supervisado, las *SVM* funcionan mejor en este caso debido a la cantidad de datos disponibles, pues las redes neuronales suelen obtener poca mejora si hay pocos datos disponibles, y a la arquitectura de la red, si se buscasen los parámetros adecuados se podría mejorar el rendimiento del *MLP*. A pesar de la escasa cantidad de datos los resultados no son para nada malos, y esto es debido a la simpleza del *dataset*.

Una vez se ha comprobado el funcionamiento de la librería con un *dataset* de la literatura, se va a probar la red neuronal con el *dataset* micológico visto anteriormente.

Se entrenará con las *features* de entrenamiento obtenidas anteriormente, junto a sus etiquetas. Después se realizarán las predicciones para los datos de validación y se obtendrá un reporte utilizando la librería de *sklearn*, *metrics* usada anteriormente. El clasificador será entrenado utilizando los parámetros vistos anteriormente. Los resultados obtenidos no son para nada aceptables, consiguiendo las siguientes puntuaciones:

Algoritmo	**Datos de entrenamiento**	**Datos de validación**
MLP	0.11	0.71

Tabla 5: Resultados de *MLP* de *Sklearn*

Como se puede observar los resultados son muy bajos, aunque era de esperar dada la calidad del *dataset* y la arquitectura de la red, similar a la creada sin librerías externas. El hecho de que la precisión en los datos de validación sea mayor se debe a la mayor cantidad de ejemplos de esa clase en el *set* de validación, en proporción con las demás clases.

Los resultados han sido obtenidos con la propia funcionalidad de la librería *sklearn*, que devuelve el *score* de cada clasificador. Para verlo en detalle se va a ver un reporte del *MLP* obtenido también

a través de la librería.

Como vemos el funcionamiento del algoritmo es de muy bajo rendimiento, como se puede observar

Especie	***precision***	***recall***	***f1-score***	***support***
Agaricus arvensis	0.0	0.0	0.0	22
Agaricus xanthodermus	0.0	0.0	0.0	26
Amanita muscaria	0.0	0.0	0.0	26
Amanita phalloides	0.0	0.0	0.0	20
Boletus edulis	0.0	0.0	0.0	20
Calocybe gambosa	0.17	1.0	0.29	28
Marasmius oreades	0.0	0.0	0.0	20
avg/total	0.03	0.17	0.05	162

Tabla 6: Reporte de resultados de *Sklearn*

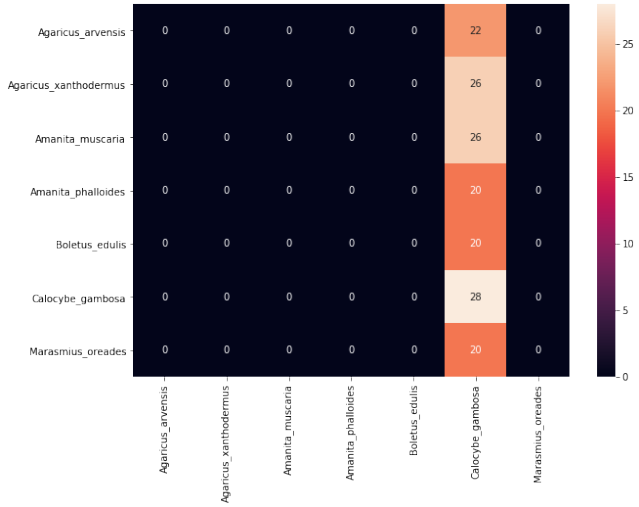


Figura 19: Matriz de confusión de *Sklearn*

en la matriz de confusión (Figura 19), siempre marca como resultado la misma clase, *Calocybe Gambosa*, por lo que la red no está sabiendo distinguir entre una y otra.

4.5. Deep Learning - Keras

Keras es una API de alto nivel de redes neuronales, escrita en Python y que es capaz de correr sobre TensorFlow, CNTK o Theano. Fue desarrollado con el enfoque de permitir una rápida experimentación. Entre sus principales características destacan:

- User-friendly: Keras es una API diseñada para humanos, no para máquinas, por lo que es fácil de utilizar, permitiendo desarrollar redes neuronales complejas en pocas líneas de código.
- Modularidad: Keras permite utilizar y combinar diferentes tipos de capas, optimizadores, funciones de coste, etc, de forma independiente de las demás.
- Fácil de extender: Añadir nuevos módulos es muy fácil, lo que permite utilizar Keras para realizar investigaciones avanzadas.

Por todo esto, Keras es una de las librerías más utilizadas en el ámbito del Deep Learning, pues permite realizar modelos complejos y personalizados en un corto periódo de tiempo, además de ser más sencillo de utilizar que otras librerías, como TensorFlow.

No obstante Keras es una API que trabaja sobre TensorFlow, CNTK o Theano, lo que hace que la ejecución sea más lenta que trabajar con estas librerías directamente, por lo que se puede perder algo de rendimiento. Por ello es recomendable usarla en fase de desarrollo o experimentación, o incluso en implementación si el rendimiento no es crítico, en cuyo caso es recomendable usar otra librería.

Entre las ventajas de Keras destaca que sea modular, lo que nos permite añadir y combinar capas, disponiendo de varios tipos de capas que permiten desarrollan redes complejas y de gran precisión. Por ejemplo se pueden utilizar capas convolucionales, capas de *Max Pooling*, etc. Estas capas son muy útiles de cara al trabajo con imágenes, pues permiten computar varios píxeles a la vez en lugar de cada píxel por separado. En la práctica cuando se trabaja con imágenes se usan las redes neuronales convolucionales (CNN por sus siglas en inglés), pero para poder hacer la comparación con las redes vistas anteriormente, que se componían unícamente de capas densas, se realizará un clasificador usando solamente este tipo de capas, lo que es comúnmente conocido como Perceptrón multi-capa. En las siguientes secciones, sin embargo, se utilizarán redes complejas buscando una alta precisión en el clasificador.

Al igual que en la sección anterior primero se realizará una prueba de la librería para ver cómo funciona. Al igual que *sklearn*, *Keras* también dispone de funciones de métrica, *datasets* o funciones útiles para el preproceso. Sin embargo, para comparar con los resultados de la sección anterior se utilizará el *dataset* de *MNIST* provisto por *sklearn*, y se dividirá en datos de entrenamiento y de validación con la función *train_test_split* de *sklearn*, con el mismo *random_state* para asegurarnos que son los mismos datos.

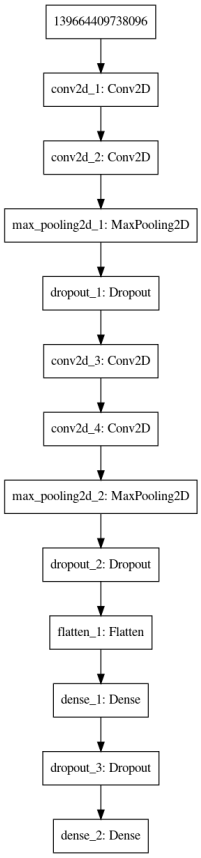


Figura 20: Arquitectura del modelo para MNIST

Para ver el potencial de la librería se cambiará la arquitectura de la red, tal y como se puede observar en la Figura 20. Ahora en la arquitectura hay capas convolucionales, de *max_pooling*, de *dropout* o de *flatten*, en total la red cuenta con 150.250 parámetros. Como optimizador se usa *RMSprop*, aunque podría haberse usado otro de los muchos disponibles, como *SGD* o *adam*. Como función de pérdida se usa *categorical cross entropy*. Tras 20 epochs se consigue un 99 % de precisión en los datos de validación, superando a los algoritmos vistos con *sklearn*. Una vez visto el funcionamiento de la librería implementaremos una red neuronal de una única capa oculta, con 15 neuronas, tal y como habíamos hecho anteriormente. Como optimizador se usará *SGD* y como función de pérdida se usará *categorical cross entropy*. Al igual que con el *MLP* implementado con *sklearn* el algoritmo no ha podido encontrar un patrón para diferenciar unas clases de otras, consiguiendo esta vez una *accuracy* de 0.16. Si vemos la matriz de confusión, Figura 21, vemos que ha predicho la misma clase para todos los ejemplos, al igual que en la sección anterior. La diferencia es que esta vez ha sido una clase diferente, en la que hay menos ejemplos y, por tanto, consigue menos *accuracy*, pero en el fondo ambos algoritmos son igual de ineficientes.

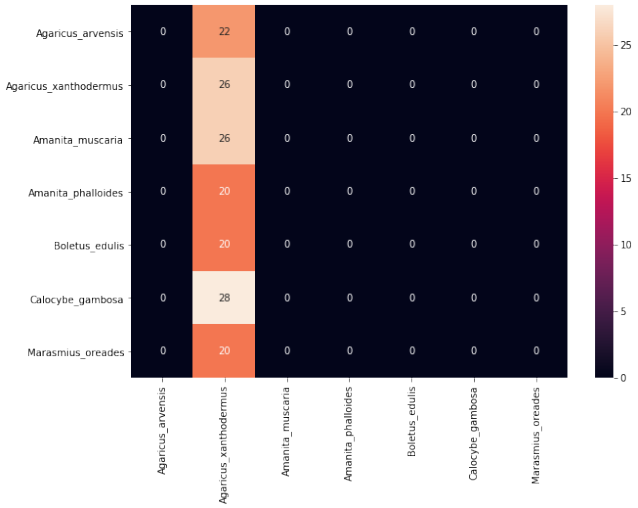


Figura 21: Matriz de confusión de *Keras*

4.6. Transfer Learning

El *Transfer Learning* o aprendizaje transferido, son técnicas de reutilización de arquitecturas complejas de redes neuronales ya pre-entrenadas. A partir de la competición de *ImageNet*, aparecieron nuevas redes neuronales, complejas, con gran cantidad de parámetros y entrenadas con *datasets* amplios. Estas redes neuronales "aprendieron" a reconocer formas, colores, entre otros, en las imágenes, características que es interesante reconocer y que no varían de un problema de clasificación de imágenes a otro, aunque siempre mejorarán si se entrena la red completa con los ejemplos de entrenamiento. Por ello se pueden reutilizar estas redes, modificando sólo las últimas capas de la red, las que actúan como clasificador, para crear uno propio con las clases necesarias.

De esta manera podemos optimizar el resultado sin necesidad de gran cantidad de recursos (como *hardware* o tiempo) que serían estrictamente necesarios para poder lograr el mismo resultado.

Hay muchos modelos pre-entrenados en la red, entre ellos se usarán los siguientes:

- *MobileNet V2*
- *ResNet 50*
- *Inception V3*
- *VGG 19*

Estos modelos están entrenados para las clases del *Imagenet*, por lo que se debe cambiar el clasificador (las últimas capas), para adaptarlo a las clases desadas. Para ello, a la hora de obtener el modelo

con *Keras*, se puede especificar si se quiere el modelo completo o sin el clasificador. De esta manera se obtienen las capas que se fijan en cosas comunes de las imágenes, como colores, contornos, formas, etc.

Como se ha eliminado el clasificador de estos modelos, se tiene que añadir uno. Se usará el mismo con todos modelos para poder compararlos, pero lo mejor sería buscar el óptimo para cada modelo, pues cada uno es diferente. El clasificador tendrá una capa densa de 256 neuronas, seguida de 3 capas densas de 100 neuronas, todas ellas con la función tangente hiperbólica como función de activación. Para terminar se usará una capa densa de 7 neuronas con la función *softmax* como activación, que será la encargada de clasificar en una de las 7 clases de setas. Como optimizador se usará *adam* y como función de pérdida *categorical crossentropy*.

Todos los modelos serán entrenados con el mismo *set* de entrenamiento y validación, y se entrenará durante 5 *epochs*. Se medirá la *accuracy*, el tiempo de entrenamiento y el peso del modelo en disco.

Aunque el tiempo de entrenamiento es considerablemente alto, los resultados son en algunos modelos más bajos de lo esperado, consiguiendo superar el 80 % de precisión en sólo 1 de los 4 modelos. No obstante si observamos la matriz de confusión de cualquiera de ellos, Figura 22, podemos observar que los modelos están aprendiendo patrones de los datos, por lo que se podrían mejorar los resultados cambiando la arquitectura de los modelos y buscando la mejor combinación de parámetros, aunque sin olvidar que la calidad del *dataset* es limitada.

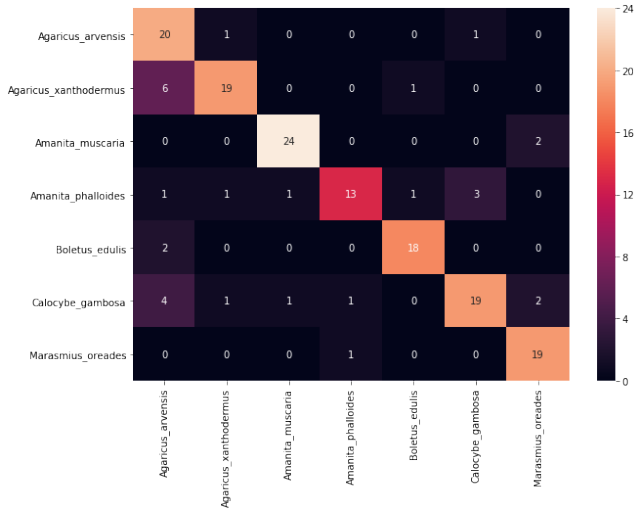


Figura 22: Matriz de confusión del modelo VGG19

5. Resultados

Como se ha visto, los modelos simples de redes neuronales densas no han servido para conseguir resolver el problema. Las 3 redes utilizadas han respondido de forma similar, clasificando todas las imágenes en una misma clase. Aunque los resultados en cuanto a calidad del algoritmo son similares, sí que se pueden comparar otros aspectos, como el tiempo de entrenamiento o el tamaño del algoritmo en disco. En la Figura 23 se observa una gráfica con los resultados. El eje X muestra el tiempo de

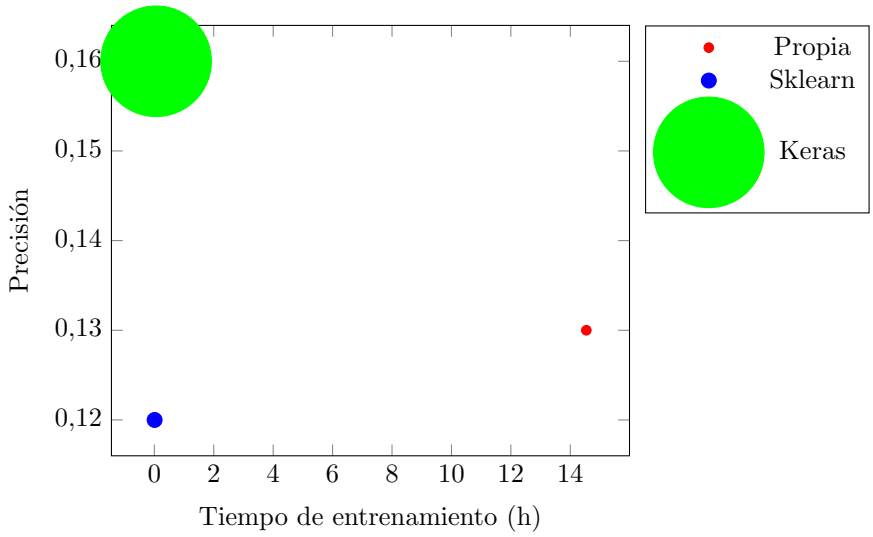


Figura 23: Comparación de redes neuronales densas

entrenamiento en horas. Como se puede observar el modelo desarrollado sin uso de librerías externas es claramente ineficiente, obteniendo un tiempo de 14 horas y 30 minutos. El modelo de la librería *Sklearn* ha sido el más rápido, seguido de cerca por el modelo de *Keras*.

El eje Y muestra la precisión de los modelos, que como se observa es muy similar en todos los casos, pues el rendimiento de los modelos es equivalente.

Por último, el tamaño de los círculos representa el tamaño de los modelos en disco. Se observa que el modelo desarrollado sin uso de librerías externas es el más ligero en tamaño. Esto se debe a que las opciones disponibles son mucho menores que en las otras librerías, ofreciendo menos funciones y parámetros, por lo que no tiene que almacenar tanta información. Se observa una clara diferencia en el modelo de *Keras*, que es claramente más pesado que el resto, en un factor de 10 respecto al modelo de *Sklearn*.

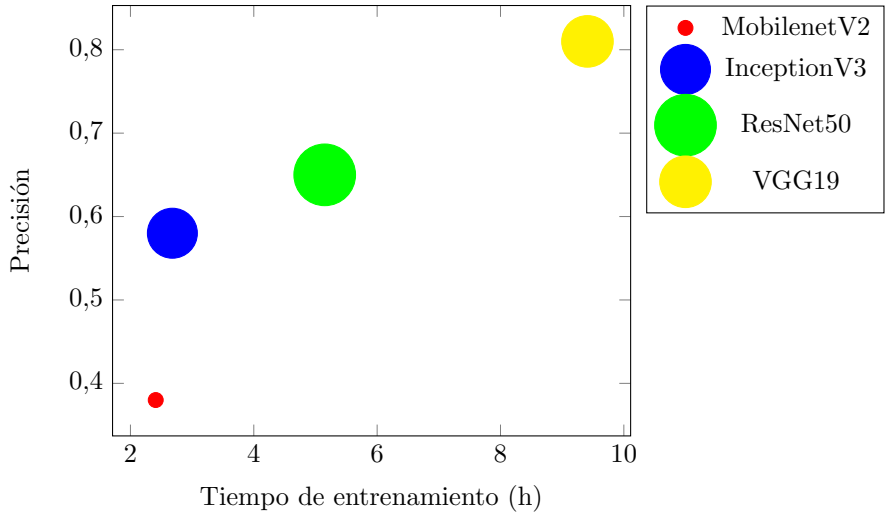


Figura 24: Comparación de modelos de *Transfer Learning*

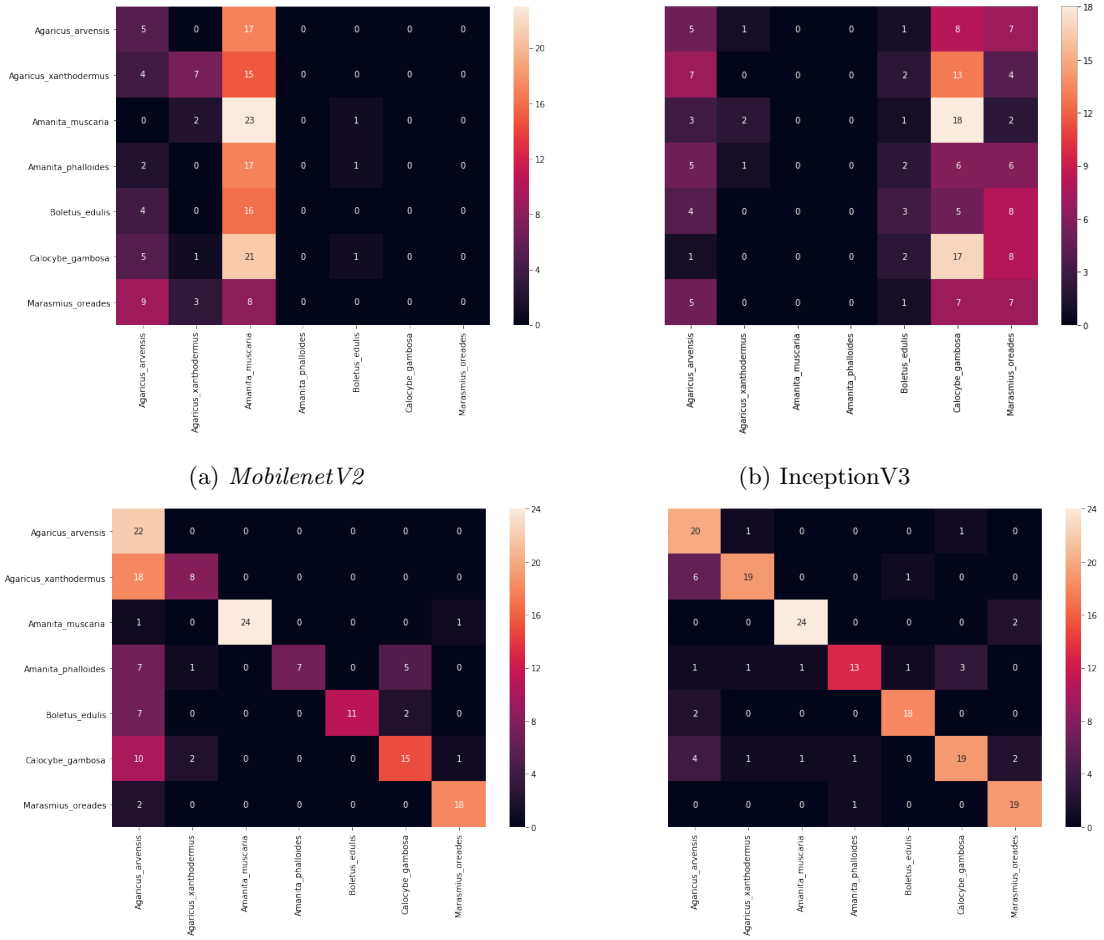
Los resultados de los modelos de *Transfer Learning* son más interesantes, pues aunque los resultados no sean del todo altos, sí que han sabido reconocer patrones en las imágenes y han llegado a detectar y clasificar algunas setas de forma correcta. Como se observa en la Figura 24, los tamaños, tiempos y resultados son muy variados, en parte por el objetivo de cada modelo. Por ejemplo, el modelo *MobilenetV2* está orientado a ser muy ligero para poder ser utilizado en dispositivos con poca capacidad de cómputo. Por ello, no es de extrañar que sea el modelo más pequeño en disco, que sea el más rápido en el entrenamiento o que, debido a tener menos parámetros, sea el menos preciso. Sin embargo, modelos como el *InceptionV3* o el *ResNet50* son opciones equilibradas, que proporcionan un rendimiento aceptable en relación al tiempo de entrenamiento y al tamaño. El problema a tratar es un problema complicado, por lo que la precisión no es alta, pero en problemas más comunes y/o simples son modelos que se comportan muy bien y que se deberían tener en cuenta.

Por último, el modelo *VGG19* es claramente el modelo con un mayor rendimiento, con más de un 81 % de precisión, pero extremadamente lento a la hora de entrenar, llegando a superar las 9 horas. El tamaño en disco es muy bueno en relación al resultado y el tiempo de entrenamiento, llegando a pesar menos que la *ResNet50*.

Pero estos resultados no implican que el *VGG19* sea mejor que los otros modelos. Cada uno ha sido entrenado de una forma diferente y puede dar mejor resultado que los otros según el problema, por lo que lo mejor es comparar todos los modelos, tratando de buscar una arquitectura en el clasificador, las últimas capas, para aumentar el rendimiento del modelo. Una vez se encuentre el modelo que mejor se ajuste al problema, habrá que buscar la mejor combinación de parámetros del clasificador

para maximizar los resultados.

Por otro lado, si se observan las matrices de confusión de los modelos, Figura 25, se puede ver que conforme aumenta la precisión deja de clasificar todos los ejemplos como la misma clase, lo que ocurría en las redes neuronales densas. Como se ve, los primeros modelos aún siguen clasificando la



(a) *MobilenetV2* (b) InceptionV3

(c) *ResNet50* (d) *VGG19*

Figura 25: Matrices de confusión

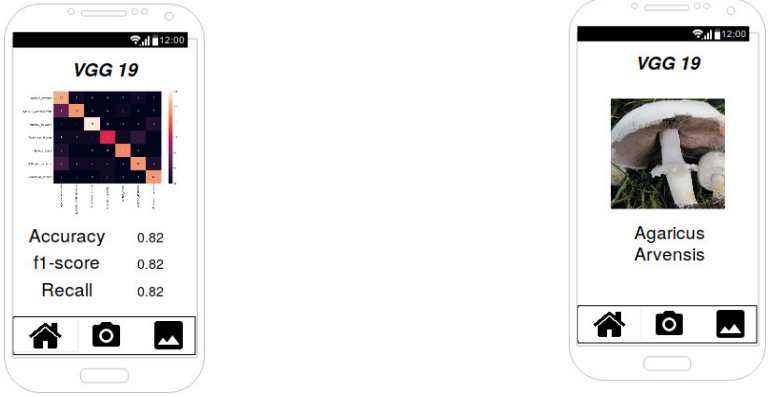
mayoría de las imágenes en la misma clase, mientras que los últimos, que tienen mayor precisión, aprenden a distinguir entre una y otra consiguiendo diferenciar la mayoría de los ejemplos.

6. APP Móvil

Entre todos los modelos entrenados habrá que elegir uno para utilizarlo en la aplicación móvil. Para la elección habrá que tener en cuenta los resultados obtenidos en el apartado anterior y los requisitos de la aplicación. En la APP a desarrollar se ha decidido que el espacio no es crítico, al igual que el tiempo, por lo que se ha podido elegir el modelo con mejor precisión, el *VGG 19*. Habrá que generar un grafo del modelo para poder utilizarlo con la *API* de *TensorFlow* para móviles. Esto puede hacerse fácilmente mediante la librería de *TensorFlow* de *Python*.

Una vez elegido el modelo y generado el grafo, habrá que diseñar la aplicación móvil. Aunque podría hacerse multiplataforma, se ha elegido desarrollar la aplicación únicamente para los sistemas *Android*.

La aplicación debería poder clasificar imágenes de setas, ya sean tomadas desde la cámara del *smartphone* o desde un archivo. Además debería poder mostrar información relativa al modelo, como la matriz de confusión. En la Figura 26a se puede ver el diseño de la primera pantalla de la aplicación. Es la que se ve al abrir la aplicación y en ella se puede ver la matriz de confusión del modelo, acompañado de 3 métricas (*Accuracy*, *f1-Score*, *Recall*). En la Figura 26b se puede ver el diseño de la pantalla resultante de clasificar una imagen, bien sea por medio de la cámara (botón central) o por elección de una imagen del dispositivo (botón derecho). Se puede volver a la pantalla incial o de información pretando en el botón de inicio (botón izquierdo).



(a) Pantalla de inicio/información.

(b) Pantalla de resultado de clasificación

Figura 26: Pantallas aplicación *Android*

Cuando se selecciona la imagen a clasificar, la aplicación móvil carga el modelo desarrollado previamente, ajusta el tamaño de la imagen y ejecuta el modelo para que, una vez se tenga el resultado, se muestre por pantalla con el resultado de la clasificación. En caso de tomar una fotografía, la imagen no se guarda en el dispositivo, evitando así almacenar imágenes para ahorrar memoria.

Como se ve en la Figura 27, la aplicación consta de un *Activity* principal, donde se irán mostrando las imágenes o los datos. La *Activity* será la que reciba las órdenes del usuario mediante las pulsaciones en la pantalla y, cuando el usuario haya seleccionado o tomado una imagen, enviará la imagen a *ImageClassifier*, una clase encargada de cambiar el tamaño de la imagen, cargar el modelo y enviar la imagen al modelo. El modelo, que es el grafo que se ha desarrollado anteriormente, realizará el proceso de clasificación de la imagen y devolverá los resultados, que la clase *ImageClassifier* interpreterá y devolverá al *MainActivity*, para que sean devueltos al usuario mostrándolos por pantalla.

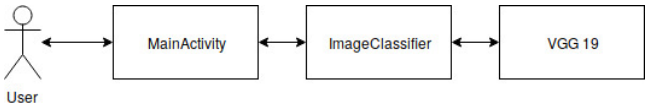


Figura 27: Componentes flujo clasificación

En la Figura 28 se pueden ver las opciones de las que dispone el usuario en la aplicación. Puede ir a la pantalla de inicio, donde se muestra la información relativa al modelo, o por otro lado puede realizar el proceso de clasificación, bien sea con una imagen tomada desde la cámara, bien sea con una imagen importada desde un archivo.

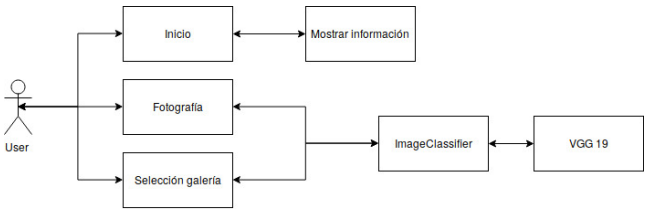


Figura 28: Opciones del usuario

Una vez desarrollada la aplicación *Android* con una interfaz de usuario sencilla, se procederá a mejorar la interfaz gráfica para hacerla más usable de cara al usuario, siguiendo las pautas indicadas en la familia de normas ISO 25010:

- Capacidad para reconocer su adecuación.

- Capacidad de aprendizaje.
- Capacidad para ser usado.
- Protección contra errores de usuario.
- Estética de la interfaz de usuario.
- Accesibilidad.

Finalmente, habrá que probar su funcionamiento.

6.1. Descarga

Para poder descargar y ejecutar la aplicación se necesitará un dispositivo Android 5.0 o superior.

La aplicación se encuentra alojada en *Github*, un repositorio online de código. Puede descargarse desde: `https://github.com/marmg/DeepMyco/raw/master/deepmyco.apk`

7. Licencias

7.1. Licencia Software

Todo el código fuente desarrollado en este proyecto cuenta con una licencia Apache:

7.2. Licencia documental

El presente documento, así como todo el trabajo de estudio realizado, se encuentra al amparo de una licencia Creative Commons.

8. Conclusiones y trabajo futuro

Las redes neuronales son una herramienta perfecta para el reconocimiento de patrones. Si se encuentra la combinación ideal de parámetros, pueden llegar a resolverse problemas realmente complejos. Esto, junto al *Transfer Learning*, hacen que sean una de las opciones más valoradas en el ámbito del aprendizaje automático, llegando a conseguir muy buenos resultados. Uno de sus principales problemas es la gran capacidad de cómputo necesaria para poder entrenar estos algoritmos, que según la arquitectura implementada pueden llegar a ser realmente costosos.

Pero no siempre consiguen resolver el problema propuesto. Como se ha visto, una red neuronal con una capa densa no puede diferenciar entre diferentes tipos de setas. Si se añaden más capas, incrementando el coste de entrenamiento, podría mejorar el rendimiento, al igual que usando diferentes tipos de capas, como las capas convolucionales. No obstante a veces esto no es suficiente, y es posible que el problema no pueda ser resuelto.

El uso de *Transfer Learning* reduce considerablemente el coste computacional y proporciona muy buenos resultados, aunque al final, si se poseen los medios, siempre se conseguirá mejor resultado entrenando la red desde el principio. Esto, además, ha supuesto un gran avance en el campo y en el uso de redes neuronales, aumentando en gran medida el uso de estos algoritmos.

La librería *Keras* acelera la creación de modelos que pueden llegar a ser realmente potentes, permitiéndonos combinar multitud de modelos y trabajar directamente sobre *TensorFlow* de forma muy fácil. Esto sumado a la gran integración con *TensorFlow* hace que sea muy sencillo exportar los modelos para trabajar sobre otras *APIs* de *TensorFlow* o para trabajar directamente con el modelo de *TensorFlow* mejorando así el rendimiento de ejecución y entrenamiento.

8.1. Trabajo futuro

Una vez identificado el modelo de *Transfer Learning* a utilizar habría que realizar una búsqueda de los mejores parámetros para resolver el problema y reentrenar la red desde el principio para intentar aumentar el rendimiento, pero esto es un trabajo que requiere de una gran capacidad de cómputo y de tiempo, por lo que no siempre es posible.

Además, podrían añadirse nuevas clases al modelo, para poder clasificar entre más especies de setas.

Para enriquecer la aplicación móvil se podrían añadir nuevas opciones, como la posibilidad de ver la lista de clases disponibles, o información sobre cada una de las especies: si es comestible, su taxonomía, su localización, sus características, sus claves dicotómicas, etc.

También podría añadirse un historial de las clasificaciones, información de dónde fueron tomadas las fotografías e incluso añadir un apartado de comunidad dónde el usuario pueda tener amigos para ver sus clasificaciones.

Para poder llegar a más usuarios, se debería llevar la aplicación a otras plataformas y sistemas operativos, como iOS, y hacerla accesible para poder ser usada por usuarios con alguna discapacidad.

9. Anexo

9.1. Medidas de rendimiento

Para medir el rendimiento de un algoritmo de *machine learning* se pueden usar varias métricas. Todas ellas basadas en la matriz de confusión, que relaciona las predicciones con los valores reales. Como se observa en la Tabla 7, si la predicción es correcta se considera Verdadero Positivo (en caso de que la predicción sea positiva) o Verdadero Negativo (en caso de que sea negativa). Si por el contrario la predicción es errónea se considera o bien Falso Positivo o bien Falso Negativo. A partir

	Clase predicha		
Clase real		*Positiva*	*Negativa*
	Positiva	Verdadero Positivo	Falso Negativo
	Negativa	Falso Positivo	Verdadero Negativo

Tabla 7: Matriz de confusión

de ella se pueden extraer las métricas de rendimiento. Las más famosas son las siguientes:

Accuracy Es la relación entre las predicciones correctas y el total. Viene dada por:

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN} \tag{11}$$

Precision Relaciona únicamente las predicciones positivas. Viene dada por:

$$Precision = \frac{VP}{VP + FP} \tag{12}$$

Recall Relaciona los Verdaderos Positivos con los que realmente son positivos. Viene dada por:

$$Recall = \frac{VP}{VP + FN} \tag{13}$$

F1-score No tan intuitiva como las anteriores pero a menudo más útil. Es una media ponderada de la *precision* y del *recall*. Viene dada por:

$$F1 - score = \frac{2 * Recall * Precision}{Recall + Precision} \tag{14}$$

9.2. Optimizadores

Para poder actualizar los pesos de las capas se pueden utilizar diferentes tipos de algoritmos de optimización, conocidos por optimizadores o *solvers*.

Aunque hay muchos algoritmos de optimización disponibles, todos parten de una base común, el *Descenso del Gradiente*, que modifican para mejorarlo.

El *Descenso del gradiente* calcula las derivadas de todas las muestras del conjunto, lo que le lleva a encontrar el mínimo, pero de una forma muy costosa. De él nacen modificaciones como el *SGD*, algoritmo que se ha visto anteriormente que, en esencia, elige muestras aleatorias en lugar de calcular las derivadas de todas las muestras, de manera que el coste es menor y se comporta mejor con un mínimo local. Otra modificación es el *Descenso de Gradiente por lotes*, *Mini-batch Gradient Descent*, que en lugar de calcular las derivadas de todo el conjunto o de forma aleatoria, lo hace de un pequeño subconjunto. Esto hace que sea más eficiente que el *Gradient Descent* o que el *SGD* en cuestiones temporales, pero es más probable que encuentre un mínimo local.

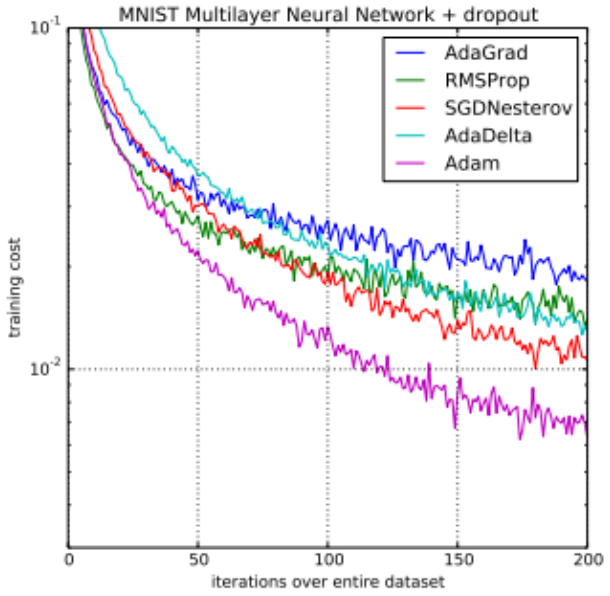


Figura 29: Comparación optimizadores

En estos algoritmos el *learning rate*, parámetro de modificación y actualización de los pesos, se mantiene estable. En versiones posteriores se modificaron para añadir el *Momentum*, una manera de actualizar el *learning rate* realizando una combinación lineal entre el gradiente y el incremento aplicado a las variables en la iteración anterior.

A partir de estos optimizadores evolucionaron otros que suelen proporcionar mejores resultados, como el *AdaGrad*[13], una modificación del *SGD* que utiliza *learning rates* diferentes en lugar de

uno único, que además actualiza teniendo en cuenta el gradiente acumulado. Una variación de este algoritmo es el *RMSProp*[14], que también utiliza *learning rates* específicos, pero con la diferencia de que solo considera los gradientes más recientes y no el gradiente acumulado.

Una combinación de *Momentum*, *AdaGrad* y de *RMSProp* es *Adam*[15]. Para mantener los *learning rates* de cada variable, *Adam* calcula una combinación lineal entre el gradiente y el incremento de la iteración anterior, unida a los gradientes más recientes.

9.3. Tipos de capas

Una capa es un conjunto de neuronas que interactúan con las neuronas de la capa anterior y de la capa posterior. Hay muchos tipos de capas, cada uno de ellos con una funcionalidad específica. Por ejemplo:

- Capas Densas:
  Las capas densas son capas en las que todas las neuronas reciben como entrada todas las neuronas de la capa anterior y, a su vez, se relacionan con todas las neuronas de la capa posterior.

- Capas convolucionales:
  Estas capas trabajan con neuronas dispuestas en 3 dimensiones: anchura, altura y profundidad. Cuando reciben la entrada la recorren tomando pequeñas ventanas de las entradas, aplicando una serie de filtros que se pueden configurar. Las neuronas propagan el resultado al resto de neuronas del espacio, consiguiendo obtener información de un conjunto de entradas y no de ellas por separado. Esto es muy útil en tratamiento de imágenes, donde un píxel no proporciona ninguna información mientras que un conjunto de píxeles sí.

- Capas de Pooling:
  Las capas convolucionales suelen ir acompañadas de capas de *Max Pooling*, que reducen las dimensiones tomando una muestra de los datos, los mayores valores de cada subconjunto.

Bibliografía

[1] Inteligencia Artificial: Un enfoque Moderno - Peter Norvig, Stuart Russell

[2] Redes Neuronales Artificiales - Antonio J. Serrano, Emilio Soria, José D. Martín.

[3] Redes Neuronales: Conceptos básicos y aplicaciones - Damián Jorge Matich

[4] *Data Augmentation* `https://medium.com/nanonets/how-to-use-deep-learning-when-\you-have-limited-data-part-2-data-augmentation-c26971dc8ced` 10/04/2019

[5] *Dataset* `https://svampe.databasen.org/` 03/02/2019

[6] *MIT* `http://www.mit.edu/` 05/05/2019

[7] *Computer Vision competition* `http://cvpr2019.thecvf.com/` 25/01/2019

[8] *ImageNet* `http://image-net.org/` 20/04/2019

[9] *AlexNet* `https://papers.nips.cc/paper/4824-imagenet-classification-with\-deep-convolutional-neural-networks.pdf` 22/04/2019

[10] *Sklearn* `https://scikit-learn.org/stable/documentation.html` 18/02/2019

[11] *Keras* `https://keras.io/` 24/03/2019

[12] FPGA-based Accelerators of Deep Learning Networks for Learning and Classification: A Review - Scientific Figure on ResearchGate. Available from: `https://www.researchgate.net/figure/ImageNet-Competition-Results-50_fig1_329975404`

[13] Adaptive Subgradient Methods for Online Learning and Stochastic Optimization: John Duchi, Elad Hazan, Yoram Singer

[14] Improving the Rprop Learning Algorithm: Christian Igel, Michael Hüsken

[15] Adam: A Method for Stochastic Optimization: Diederik P. Kingma, Jimmy Ba

[16] *Convolutional neural networks* `http://cs231n.github.io/convolutional-networks/` 22/05/2019