Trabajo de Fin de Grado

Grado en Ingeniería Informática

Redes neuronales para clasificar imágenes de sedimentos arqueológicos

Neural networks to classify archaeological sediments images

Rafael González de Chaves González

La Laguna, 3 de septiembre de 2018

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Ayudante doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Carolina Mallol Duque**, con N.I.F. 03.880.747-A profesor Titular de Universidad adscrito al Departamento de Geografía e Historia de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Redes neuronales para clasificar imágenes de sedimentos arqueológicos"

ha sido realizada bajo su dirección por D. **Rafael González de Chaves González**, con N.I.F. 79.098.361-G.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 3 de septiembre de 2018

Agradecimientos

Agradecer a mi tutor Rafael Arnay del Arco y a mi cotutora Carolina Mallol Duque por el conocimiento y el apoyo ofrecido para completar este TFG.

Y sobre todo, agradecer a mis compañeros y familia que me han ayudado a seguir adelante durante la carrera de Ingeniería Informática.

Licencia

Resumen

El objetivo principal de este trabajo ha sido crear un programa que clasifique varios tipos de sedimentos arqueológicos en una imagen.

Para ello he creado una red neuronal convolucional capaz de resolver este objetivo, la cual, tras la reiteración de pruebas he personalizado para que sea lo más precisa y rápida posible. Para complementar a la red he desarrollado varios programas con interfaz gráfica en Python, con el fin de hacer más intuitivo y fácil el uso de la red para el usuario.

Tras la realización de los programas complementarios y la personalización de la red, el usuario puede introducir al programa las imágenes que quiera clasificar y éste lo hará automáticamente, todo esto con una precisión del 89% y de forma rápida.

Todo esto ha sido desarrollado usando diferentes lenguajes, tecnologías y frameworks. Estos son Python, TensorFlow, Keras, NumPy, Pillow y Tkinter entre otros.

Abstract

The main objective of this project is to create an application to classify different features in microphotographs of archaeological sediment.

I created a convolutional neural network capable of tackling this goal. Then, after running a test I customized it to be as accurate and fast as possible. To complement the neural network, I developed several graphical interface applications using Python to make the neural network more intuitive and user friendly.

After the completion of the supplementary applications and the customization of the neural network, the user can enter the desired images and these will be automatically classified with an 89% accuracy.

All of this has been developed using different programing languages, technologies and frameworks. This are Python, TensorFlow, Keras, NumPy, Pillow and Tkinter.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1
Introducción

1.1 Introducción y problemática

En este proyecto se pretende entrenar una red neuronal para identificar distintos tipos de estructuras conocidas como 'poros', presentes en imágenes de láminas delgadas de sedimentos arqueológicos. Hay cinco tipos: Canales, Vesículas, Fisuras, Cavidades irregulares y Cámaras. Los poros destacan en este tipo de imágenes al ser siempre blancos. La idea es que una vez se detecta la presencia de un poro, éste sea clasificado dentro de una de las cinco categorías expuestas y cuantificadas. Las imágenes con las que se va a trabajar serán cedidas por la Dra. Carolina Mallol, investigadora Ramón y Cajal adscrita al Departamento de Geografía e Historia y al Instituto de Bioorgánica Antonio González, donde es directora del Laboratorio de Micromorfología y Biomarcadores Arqueológicos.

Figura 1.1-1 Canales

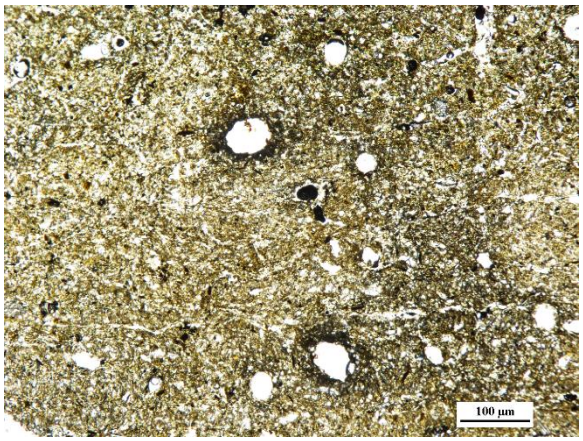

Figura 1.1-2 Vesículas

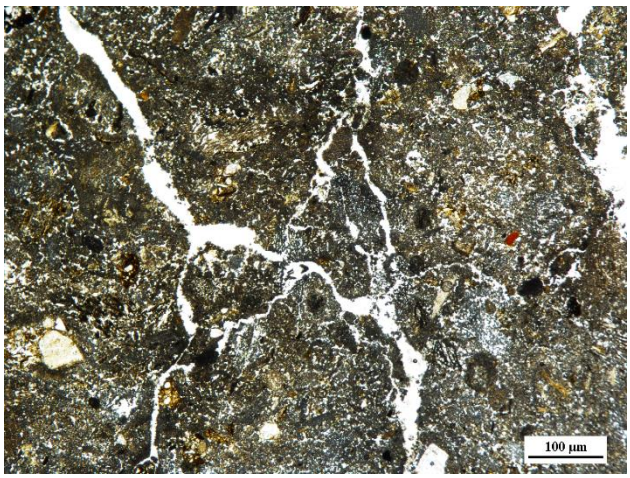

Figura 1.1-5 Fisuras o planos

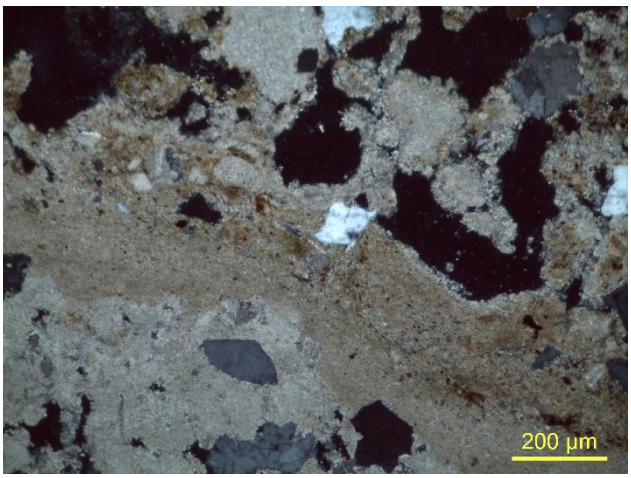

Figura 1.1-4 Cavidades irregulares

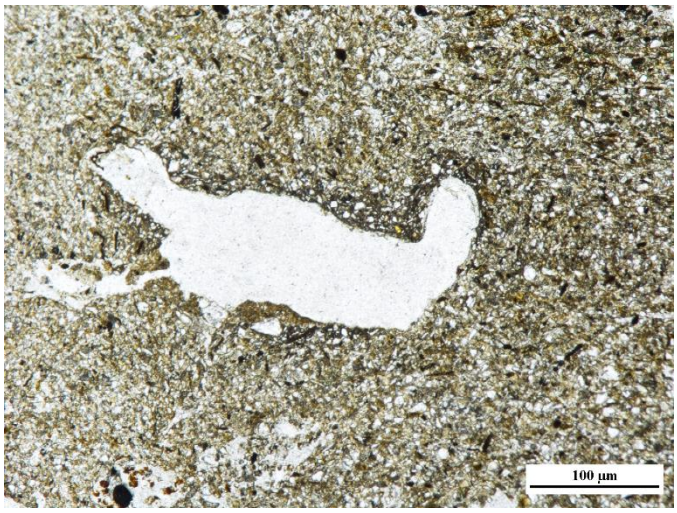

Figura 1.1-3 Cámaras

1.2 Antecedentes

Actualmente se han usado múltiples veces las redes neuronales para la clasificación de imágenes en diversos campos, y también en el campo de la arqueología pero para detección de restos arqueológicos con imágenes aéreas [(1)], sin embargo, nunca se había usado en la clasificación de imágenes arqueológicas de extractos del suelo.

(1) M.C. Cantero, R.M. Pérez, J. Plaza, E. Cerrillo y A. Bejarano, TÉCNICAS NEURONALES PARA DETECCIÓN REMOTA DE RESTOS ARQUEOLÓGICOS, X Congreso de Teledetección Cáceres España, 2003, 443-446

1.3 Objetivos y Fases

- Recolección y clasificación de imágenes arqueológicas según el tipo de poro existente.
- Creación de una interfaz gráfica para la fácil clasificación de imágenes arqueológicas.
- Diseño y entrenamiento de una red neuronal para el reconocimiento de la presencia o no de poros y el tipo de los mismos.
- Realización de pruebas con diferentes estructuras de redes neuronales.
- Búsqueda de la estructura con menor índice de error con este tipo de imágenes.
- Creación de una interfaz gráfica para el uso de la red y la visualización de los resultados del reconocimiento.

Capítulo 2
Desarrollo del proyecto

2.1 Explicación de la tecnología

En mi trabajo de fin de grado he desarrollado un programa que es capaz de detectar los diferentes tipos de poros en las láminas de sedimentos arqueológicos presentes en una imagen, para ello he usado redes neuronales convolucionales.

Las redes neuronales son un conjunto de neuronas artificiales, organizadas en capas que intenta imitar el funcionamiento de las neuronas del cerebro, con el fin de aprender como hace un humano.

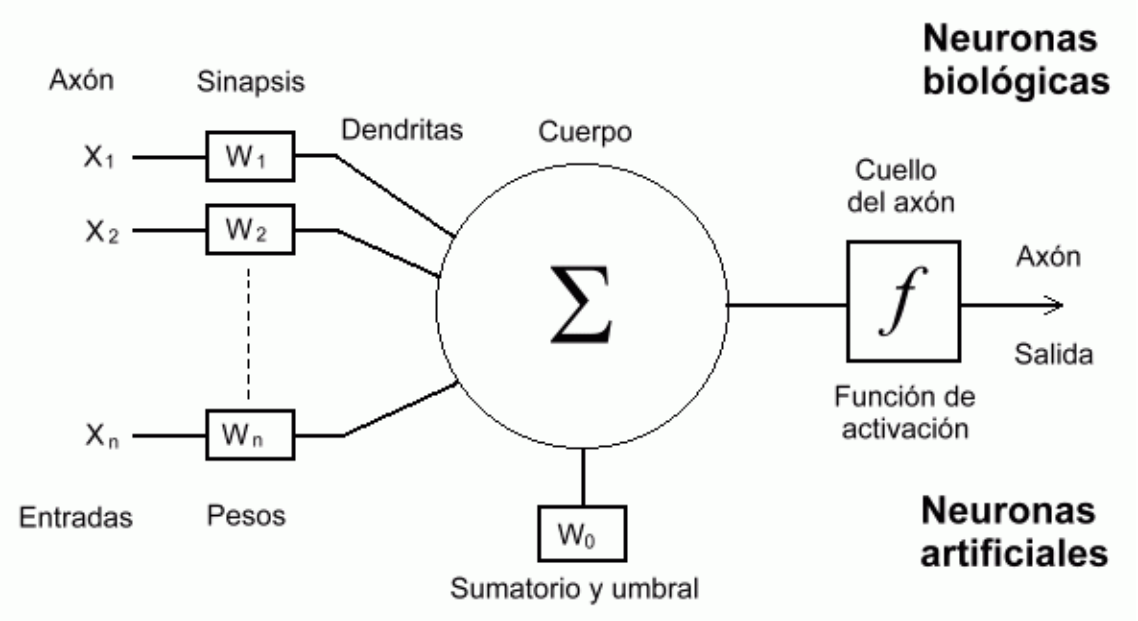

Figura 2.1-1 Neurona Artificial

Dentro de una red neuronal, cada neurona es un pequeño programa que recibe un vector de entradas y que tiene que emitir una única salida, para ello la neurona necesita que las entradas tengan asignadas unos pesos según la importancia del valor, éste se usará para combinar todas las entradas en un único valor combinado, esto se puede conseguir usando muchas operaciones diferentes pero una de las más comunes es la suma ponderada, en esta operación el valor combinado se obtiene a partir del sumatorio de todas las entradas

multiplicadas por sus respectivos pesos. Una vez obtenido el valor combinado se ha de emitir una salida, esta salida dependerá de la función de activación [1] a la que se someterá al valor combinado.

Hace años la función más comúnmente usada era la función escalón, en esta función si el valor combinado igualaba o sobrepasaba el umbral establecido previamente, la salida de la neurona era 1, y si no lo alcanzaba la salida de la neurona era 0, esto hacía que pequeños cambios en las entradas de la red neuronal pudiesen provocar grandes cambios en los resultados finales de la misma, por ello actualmente se usan otros tipos de funciones, que permiten un entrenamiento más preciso gracias a que pueden devolver infinitos valores entre 0 y 1, así los pequeños cambios en las entradas afectan poco a las salidas. Las funciones de activación usadas actualmente [2] son identidad, sigmoidea, tangente, gaussiana, unidad lineal rectificada (ReLU) y softmax, siendo esta última como la ReLU pero suavizada.

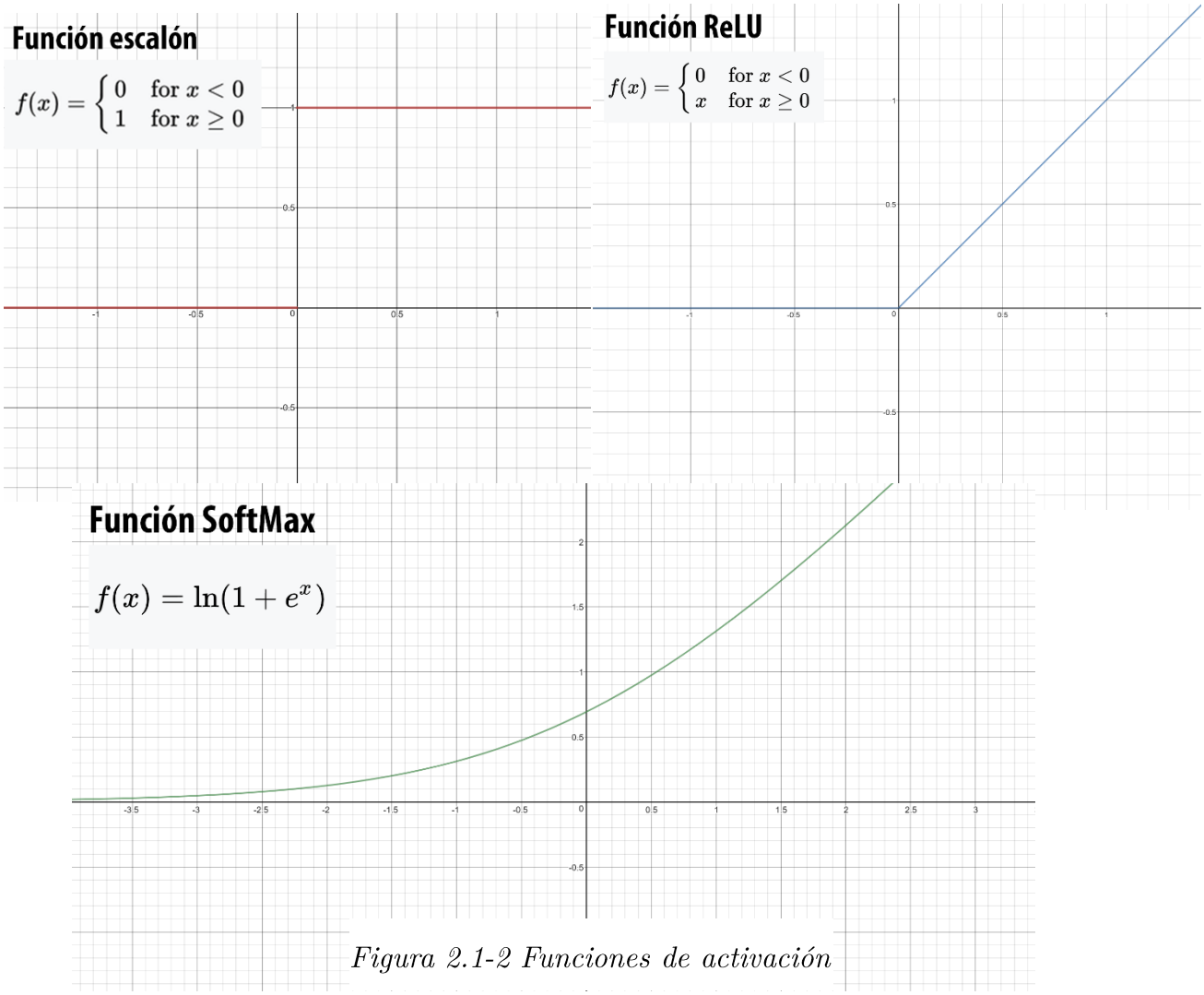

Figura 2.1-2 Funciones de activación

Combinando estas neuronas entre ellas y conectándolas se consiguen crear capas de neuronas, y conectado estas capas entre ellas se consigue crear una red neuronal.

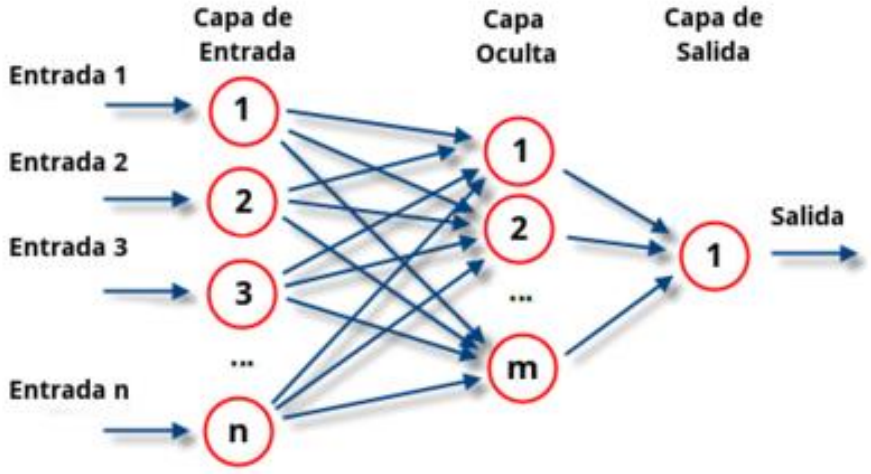

Figura 2.1-3 Capas de una red

2.2 Tipos de capas de la red

Las redes neuronales pueden estar formadas por cientos de capas conectadas entre sí, sin embargo hay muchos tipos de capas dependiendo del tipo y las conexiones de las neuronas de su interior, ya que no todas las capas cumplen la misma función dentro de una red neuronal. Los tipos de capas más comunes en una red neuronal convolucional como la que yo hice son la capa de entrada, la capa densa, la capa de desecho, la capa de aplanamiento, la capa de reestructuración, la capa convolucional, la capa de agrupamiento y la capa de salida.

La capa de entrada [3] es una capa que se encarga de introducir los datos de entrada en la red como puede ser unas imágenes, esta capa solo puede ser usada como primera capa de la red, los datos introducidos por esta capa se manipulan en la red en forma de tensor, la unidad de datos en las redes neuronales basadas en la API TensorFlow [16] (La API más usada en el mundo de las redes neuronales). Esta unidad guarda las entradas a la red como vectores multidimensionales de datos, por ejemplo en mi proyecto las entradas de la red son un conjunto de imágenes RGB, éstas son guardadas como un vector de 4 dimensiones, ya que una imagen es representada como una matriz de valores, es decir un vector de 2 dimensiones, si le añadimos color necesitaremos 3 canales

(Red, Green, Blue), es decir, 3 vectores de 2 dimensiones, lo que nos daría un vector de 3 dimensiones que representa una imagen RGB, y como queremos entrenar la red con decenas de miles de imágenes para que sea precisa, necesitamos un vector de imágenes RGB, por lo que es necesario un vector de 4 dimensiones.

Un tensor también almacena la forma que serán agrupadas las entradas en conjuntos de entrenamiento, la forma que poseen esas entradas, que en el caso del ejemplo anterior la forma sería 4 dimensiones, la primera del tamaño del número de imágenes, la segunda del tamaño del alto de las imágenes, la tercera del tamaño del ancho de las imágenes, y la última del tamaño del número de canales de la imagen, esto se suele representar con esta notación (x,y,z,t), lo que para 50 imágenes RGB de tamaño 32 píxeles por 64 píxeles sería (50, 32, 64, 3). Y por último un tensor también almacena el tipo de los datos en los que está la información, por ejemplo números enteros o números flotantes, y dentro de ellos la precisión con la que están almacenados, 16 bits, 32 bits y 64 bits son las precisiones más comunes.

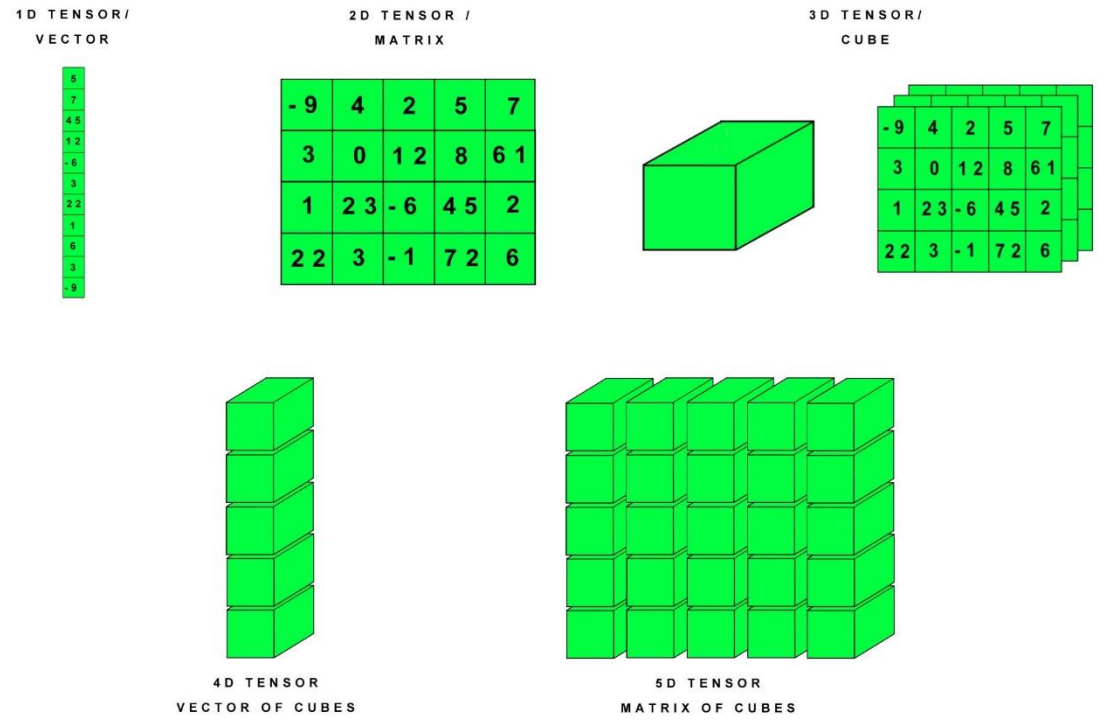

Figura 2.2-1 Dimensiones de un vector

La capa densa [4] es una capa formada por un conjunto de neuronas completamente conectadas entre ellas, es decir, cada una de las neuronas de esta capa recibe todas las entradas de la capa, es decir, todas las salidas generadas por las neuronas de la anterior capa. Esta capa se usa para aplicar una operación lineal a la salida de la anterior capa o para pasar los resultados de la anterior capa por una función de activación diferente.

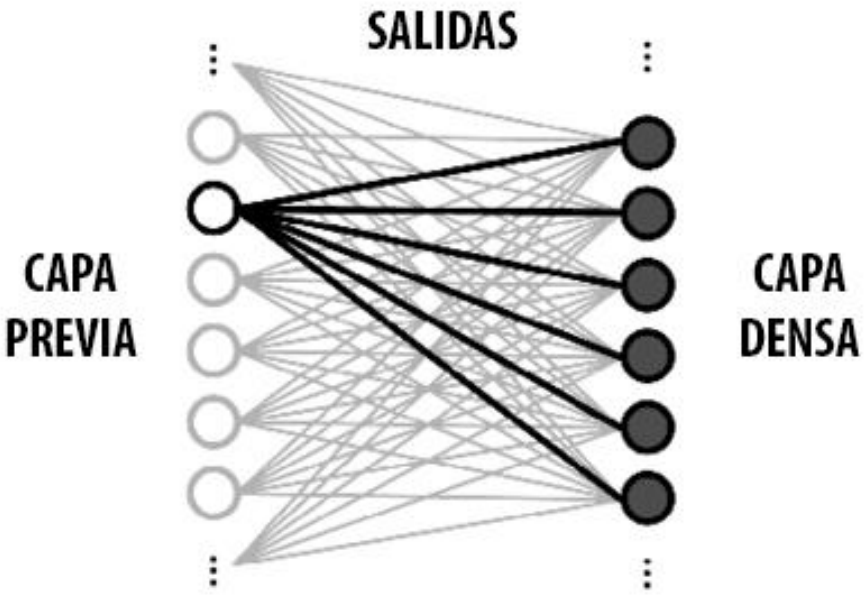

Figura 2.2-2 Capa densa

La capa de desecho [5] es una capa que se encarga de desactivar en el entrenamiento un porcentaje de las neuronas de forma aleatoria.

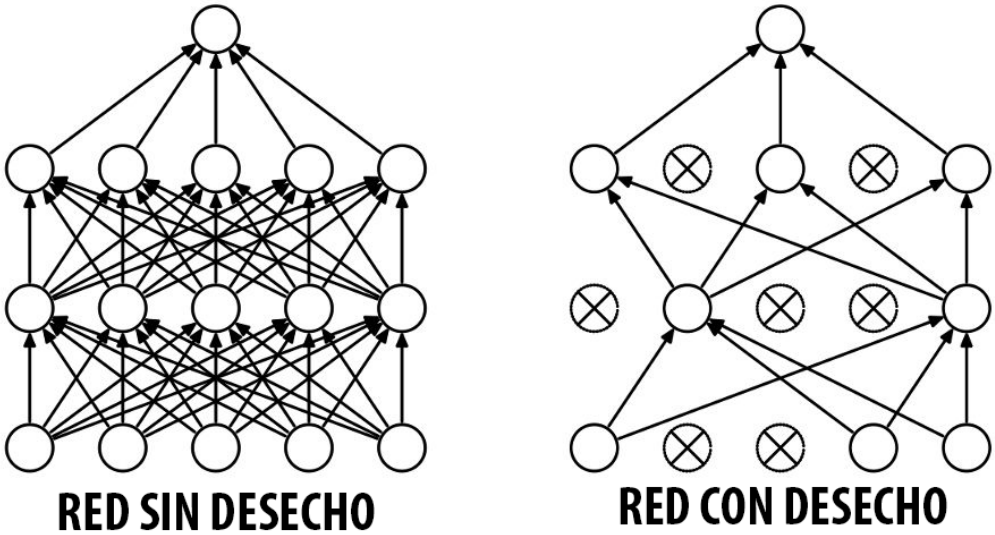

Figura 2.2-3 Capa de desecho

Esto se hace para evitar un fenómeno llamado sobreentrenamiento. Este fenómeno se da cuando se entrena mucho una red con los mismos datos o con un número reducido de datos, la red aprende a predecir la salida según unos criterios que no son los deseados, o simplemente aprendería cuales son los resultados deseados para cada una de las entradas por reiteración, y con nuevos datos fallaría en la predicción. Esta capa le resta un poco de precisión a la red pero le añade generalización porque fuerza a varias neuronas de la misma a diferenciar las mismas características por lo que red funcionará un poco peor con los datos de entrenamiento pero mantendrá ese funcionamiento con todo los posibles datos de entrada. En una red sin esta capa la precisión de la red con datos diferentes a los de entrenamiento disminuye.

La capa de aplanamiento [6] es una capa que se encarga de transformar la forma de los datos de entrada a un vector de n datos de una sola dimensión.

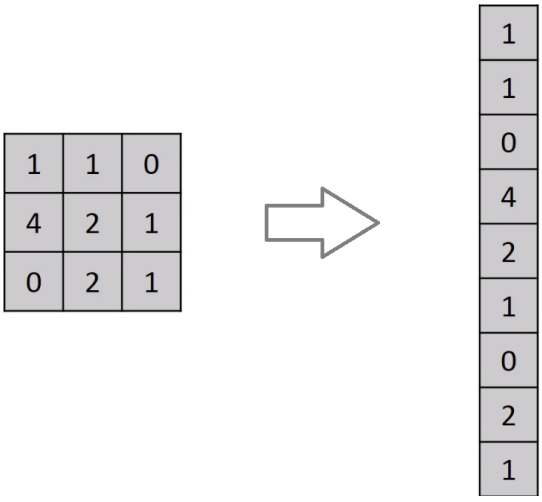

Figura 2.2-4 Capa de aplanamiento

Basándonos en el mismo ejemplo explicado en la capa de entrada, si esta capa recibe un tensor de forma (3, 64, 64, 3), 3 imágenes RGB de 64 píxeles por 64 píxeles, lo aplanaría a un vector de n datos de una sola dimensión dando como resultado un tensor de forma (3, 12288) que es equivalente (3, 64*64*3), esta capa solo modifica la forma de representar y manipular los datos pero nunca los datos en sí mismos.

La capa de reestructuración [7] es una capa que se encarga de transformar la forma de los datos de entrada a la forma deseada en la salida. Funciona como la capa de aplanamiento pero esta capa te permite cambiar la forma a la deseada, siempre y cuando la forma del tensor de entrada sea equivalente a la forma del tensor de salida, es decir, tienen que contener los mismos datos y la misma cantidad de datos.

Figura 2.2-5 Capa de reestructuración

Por ejemplo si esta capa recibe un tensor de forma (3, 64, 64, 3), 3 imágenes RGB de 64 píxeles por 64 píxeles, esta capa podría modificar su forma a (3, 3, 4096) que contiene la misma información pero en la salida el ancho y alto de las imágenes es representado en una sola dimensión, es lo mismo 3 imágenes RGB de 64 píxeles por 64 píxeles que 3 imágenes RGB de 4096 píxeles, o también se podría modificar su forma a (9, 4096) que contiene la misma información pero cada canal es una imagen diferente, es lo mismo 3 imágenes RGB de 4096 píxeles que 9 imágenes monocromo de 4096 píxeles.

La capa convolucional [8] es una capa que es capaz de aplicar al tensor de entrada filtros que se aprenden durante el entrenamiento, este tipo de capa es básico para el uso de imágenes en redes neuronales y el aprendizaje de los patrones y detalles en las mismas, esta capa genera salidas mucho más grandes que las entradas, por eso estas capas suelen ir acompañadas por capas de agrupamiento que reducen el tamaño de las salidas.

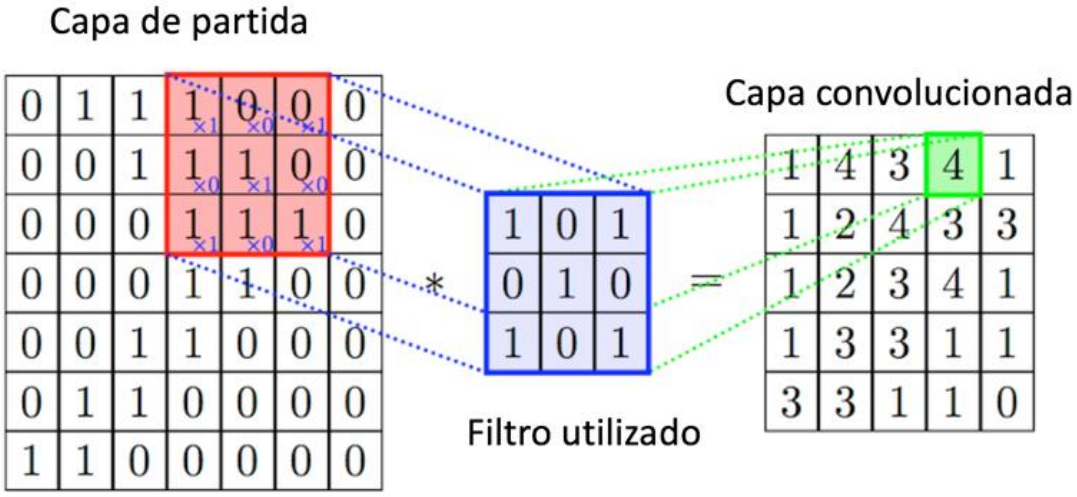

Figura 2.2-6 Capa de convolución

La capa de agrupamiento [9] es una capa que reduce el tamaño de las entradas que recibe, dependiendo de cómo esté configurada la capa, esto lo consigue gracias a que se calcula un dato de la salida usando el máximo, mínimo o media de X datos de la entrada, esto hace que la salida sea X veces más pequeña.

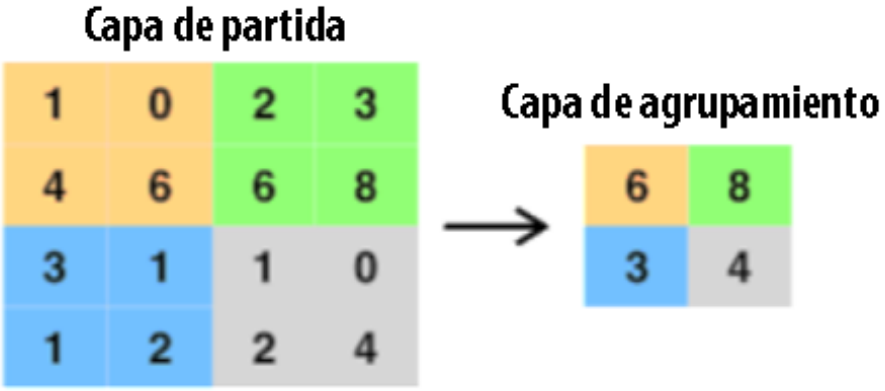

Figura 2.2-7 Capa de agrupamiento

La capa de salida [10] es una capa densa con tantas neuronas como posibles respuestas pueda dar la red, las posibles respuestas pueden ir desde un sí o no hasta miles de categorías diferentes, cada salida de esta capa significará el peso con el que la red piensa que ese es el resultado, la red neuronal no tiene blancos o negros sino una escala muy precisa de grises. Si diseñas una red que identifique si hay gatos en una imagen, ella te responderá un valor entre 0 y 1 que significa el valor de confianza de que haya.

2.3 Elección de la red

Gracias a la gran cantidad de características diferentes y personalizables que posee una red neuronal hay muchos tipos de redes, pero en la actualidad para la clasificación de imágenes se usan redes neuronales convolucionales, estas redes son en la actualidad las mejores para esta tarea, porque funcionan de forma parecida a las neuronas de la corteza visual del cerebro humano, sin embargo su representación se hace con matrices bidimensionales de varios canales, como podría ser una imagen RGB.

Estas redes están formadas por cientos de capas con miles de neuronas cada una, normalmente estas redes usan una capa de entrada, varias capas densas, varias capas convolucionales, varias capas de agrupamiento, una capa de desecho, una capa de aplanamiento y una capa de salida, aunque esta estructura básica se puede modificar con tipos de capas diferentes o algunos que se están desarrollando a día de hoy, como por ejemplo las capas convolucionales separables.

Para el desarrollo de este trabajo era necesario el uso de una red neuronal convolucional como la descrita, pero tenía dos opciones para elegir cómo hacerla, podía hacer mi propia red y personalizarla por completo yo mismo para resolver el problema de diferenciar varios tipos de poros en láminas de sedimentos arqueológicos, o podía reutilizar redes creadas por otras personas, ganadoras de concursos importantes [11], y entrenadas en naves de servidores, a las cuales yo podía reentrenar parte de ellas para que la red no solo sea precisa para el propósito que fueron creadas, y también sean capaces de resolver el problema de diferenciar varios tipos de poros.

Entre estos posibles caminos decidí probarlos todos y quedarme con el que mejores resultados diese tras las pruebas.

2.4 Elección de las herramientas

Pero antes de empezar tuve que elegir la tecnología que usaría para hacer el desarrollo de la red neuronal, había diferentes posibilidades como Torch [12], Caffe [13], Theano [14], CNTK [15] o TensorFlow [16]. Estos son los frameworks de redes neuronales más usados, y además todos estos permiten el uso de GPUs para acelerar su entrenamiento y funcionamiento, aunque finalmente me quedé con TensorFlow por ser el framework más usado con diferencia.

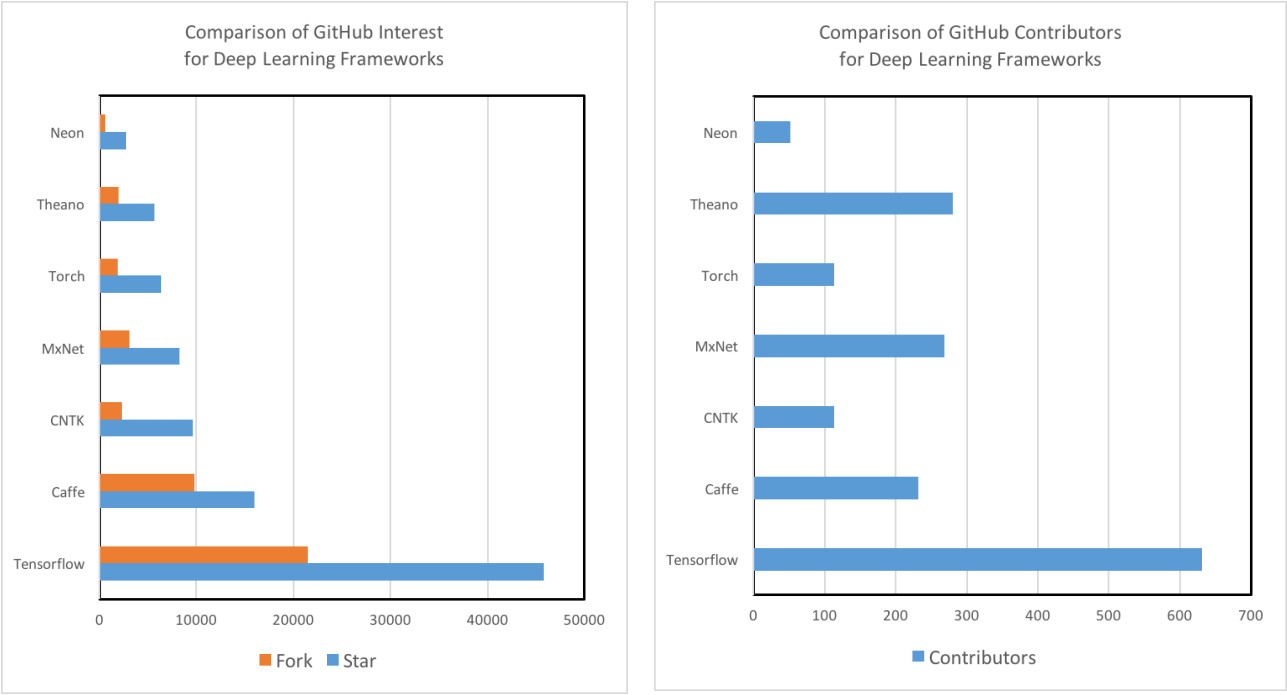

Figura 2.4-1 Uso de TensorFlow

También es uno de los mejores en rendimiento, además tiene flexibilidad en el lenguaje de programación a usar, y está desarrollado por Google, por lo que tiene mucho soporte y actualizaciones, y como última característica, muy útil para personas que están aprendiendo como lo estaba yo, TensorFlow posee un panel llamado TensorBoard [17], que te permite ver los resultados y los cambios que se producen en la red y en las funciones de la red durante su entrenamiento, y además ver cómo está estructurada internamente la red y la conexión entre las capas, todo esto de manera gráfica, lo que lo hace muy sencillo.

Una vez elegí el framework a usar, tuve que elegir el lenguaje de programación en el que desarrollaría la red neuronal. Mi primera intención y gracias a la flexibilidad que me daba TensorFlow era usar C++, ya que era el lenguaje que más manejaba, pero tras investigar me di cuenta de que no estaba muy bien soportado aun oficialmente y que no había mucha documentación por Internet, así que investigué cual era el lenguaje que usaba la mayoría de la gente, y descubrí que para la inteligencia artificial el lenguaje más usado es Python [18], y que a su vez el lenguaje con más soporte en TensorFlow era Python, así que después de tomar todas estas decisiones tuve que aprender Python y TensorFlow.

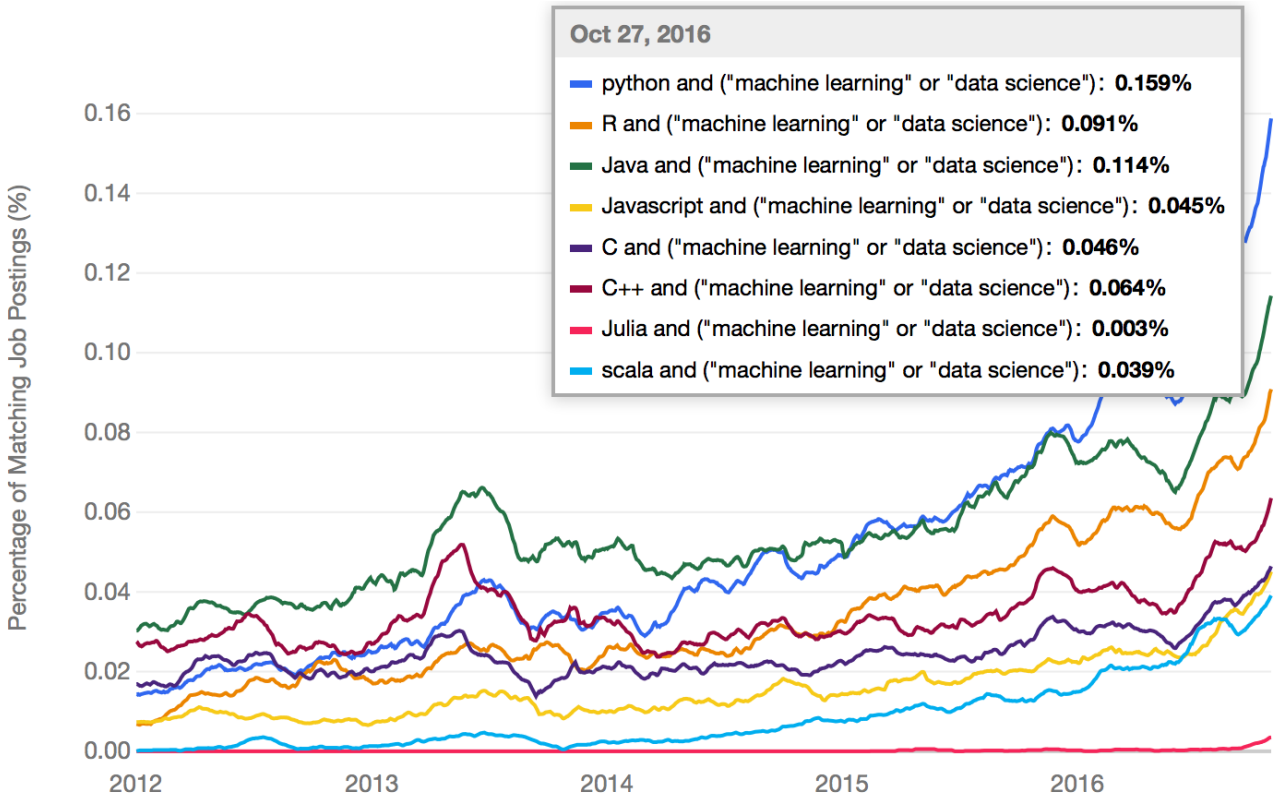

Figura 2.4-2 Uso de Python

Para aprender ambas cosas hice muchos ejemplos de redes, para otros propósitos como la identificación de dígitos escritos a mano por una persona y para propósitos similares al mío, y aprendiendo e investigando por Internet descubrí un framework llamado Keras [19], éste usa TensorFlow de fondo y obtiene el mismo rendimiento que TensorFlow, pero se pueden desarrollar redes neuronales de forma más fácil y rápida, y también se programa en Python e incluye soporte para el uso de GPUs, y mantiene el soporte para TensorBoard. Por lo que decidí continuar mi aprendizaje sobre este framework, ya que con el conocimiento que ya poseía de Python y TensorFlow el cambio sería transparente y no me supondría mucho esfuerzo.

2.5 Adquisición de las imágenes

Pero antes de empezar a crear redes neuronales y entrenarlas tenía que adquirir y clasificar las imágenes de los poros, la base de este trabajo. Mi cotutora Carolina Mallol Duque me proporcionó una biblioteca de aproximadamente 200 imágenes, divididas en 6 tipos de poros diferentes, estos son: Cámaras, Canales, Vacíos empaquetados, Planos, Vesículas y Cavidades.

Pero no podía usar estas imágenes, ya que la red no admite entradas de imágenes muy grandes, pero las imágenes que poseía tenían tamaños entre 800x800 píxeles hasta algunas de 2500x2500 píxeles, por ello debía dividir las imágenes en trozos más pequeños, lo suficientemente pequeños para que la red los aceptara, y lo suficientemente grandes para que la mayor parte de los objetos importantes de las imágenes cupiesen dentro. Al principio no sabía cuál era el tamaño ideal así que decidí utilizar 4 tamaños diferentes (100x100, 150x150, 200x200 y 250x250) y probar redes con los 4 para descubrir cual me proporcionaba mejores resultados.

Pero al solucionar este problema me surgía otro problema, al dividir las imágenes en trozos más pequeños no todos los trozos contendrían algún tipo de poro, algunos solo contendrían información no relevante para lo que se estudiando así que habría que clasificarlo como Vacío, por lo que hay que coger cada una de las 200 imágenes, dividirlas en trozos más pequeños, y clasificar cada uno de los trozos según el poro que posean, el problema de esto es que se generarían una cantidad enorme de trozos, por ejemplo para 44 imágenes que contienen Planos una vez divididas en trozos de 100x100 píxeles y clasificadas se generaron 11025 trozos de imágenes que contenían Planos y 38694 trozos que contienen Vacío, ya que se manejan cantidades de decenas de miles de trozos imágenes es inviable hacer la división y clasificación de las mismas manualmente, por ello desarrollé un programa para hacer más fácil esta tarea, y aprovechando que tenía que aprender Python lo desarrolle en este lenguaje.

Este programa está creado para poder crear y clasificar los trozos de las imágenes de manera más cómoda y en menos tiempo, no solo para que yo pueda usarlo para entrenar la red, sino para que también mi cotutora Dra. Carolina Mallol pueda en un futuro crear más imágenes de entrenamiento, y con ellas entrenar más la red con nuevas entradas para conseguir que sea más precisa.

Este programa está creado con la API de interfaces gráficas nativa de Python, Tkinter [20], es una API bastante básica pero soportada por todos los sistemas y desde versiones anteriores a Python 3, y además tiene la ventaja que cuando se instala Python esta API ya está incluida en el paquete por lo que no es necesario instalar más dependencias. Sin embargo al ser tan básica tiene algunos problemas, por ello la redimensión de las imágenes y de los controles de personalización está calculada de manera manual, para adaptarse al tamaño de la ventana, esta calcula en un método propio que se llama cada vez que la ventana cambia de tamaño. Además la API Tkinter está hecha para funcionar de forma monohilo, y eso hacía que cuando se ejecutaba una tarea pesada, como dividir la imagen, el programa congelase la interfaz para poder dividir la imagen en segundo plano y parecía que el programa se había dejado de funcionar, por ello hice la interfaz multihilo para que no se bloqueara, y también para que el usuario sepa que hay una tarea haciéndose en segundo plano añadí una barra de progreso infinita en un hilo independiente, que avisa al usuario que se está trabajando en segundo plano y bloquea la interacción con la interfaz para que ésta no se sature, y desaparece sola cuando la tarea termina.

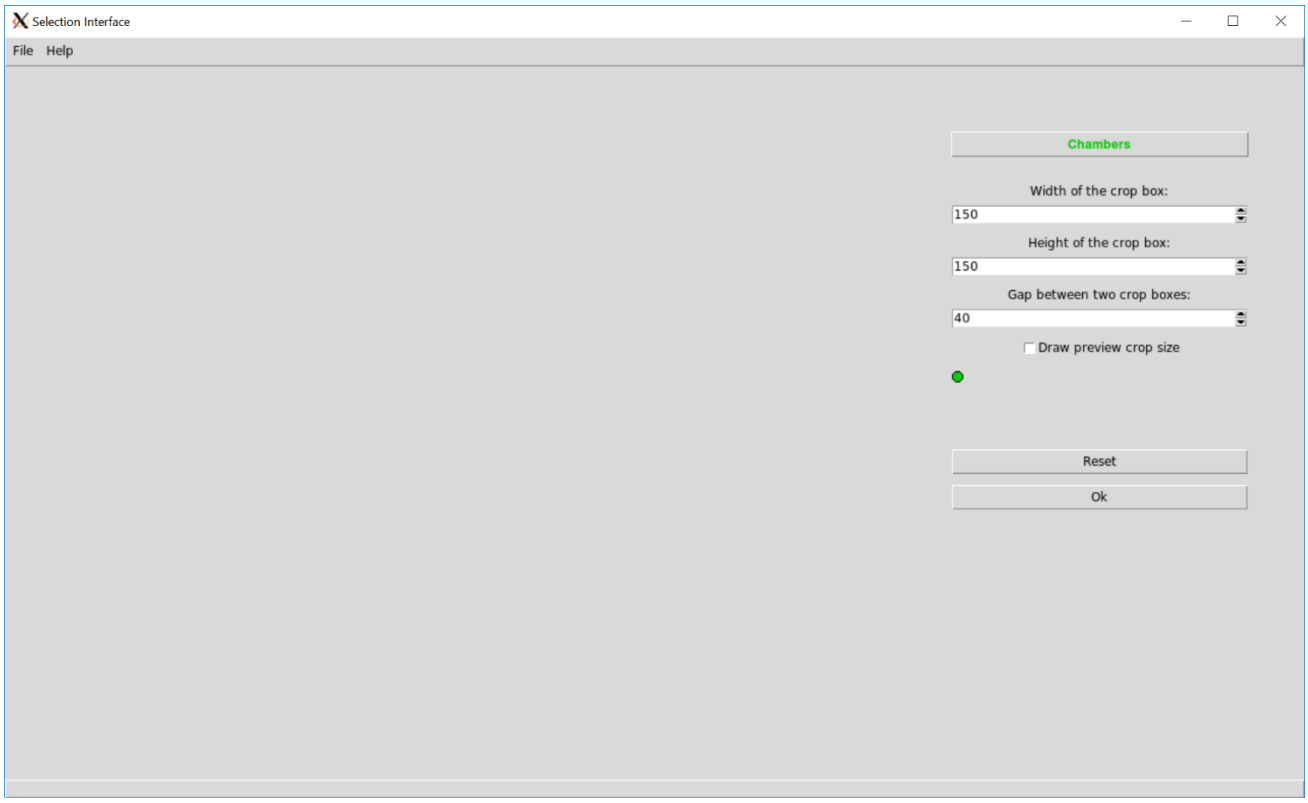

Figura 2.5-1 Interfaz de clasificación

Cuando se ejecuta este programa se abre una interfaz gráfica, en ella se puede observar la mayor parte de la interfaz vacía por ahora, también unos menús en la parte superior izquierda de la ventana y unas opciones de personalización en la parte derecha de la ventana.

Para empezar, analicemos el contenido de los menús de la parte superior izquierda de la ventana. Dentro del primer menú Archivo (File) hay dos opciones, la primera opción es para abrir una imagen (Open file), se puede hacer click en ella o acceder a través del atajo de teclado Alt+O, al acceder a esta opción se abre una nueva ventana que nos permite navegar por el árbol de directorios para buscar la imagen que se desea abrir, están limitados los archivos que se pueden abrir a solo 3 extensiones, PNG, JPG y TIF ya que son las extensiones más usadas, aunque en este caso solo usé imágenes en formato TIF ya que es un formato sin perdidas que permite mejor calidad en las imágenes.

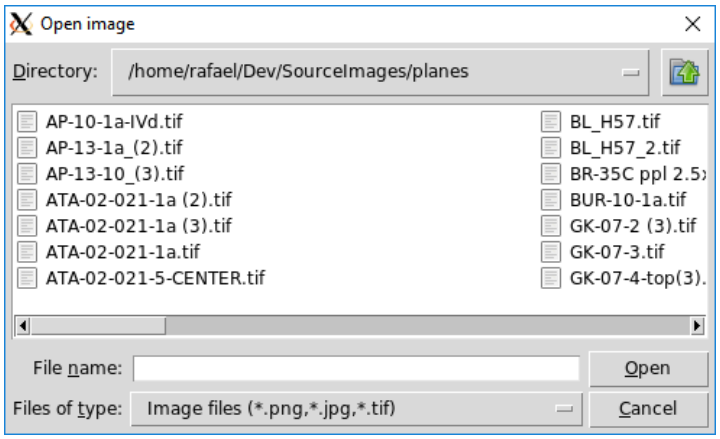

Figura 2.5-2 Ventana de selección del fichero

La segunda opción dentro del menú Archivo (File) es para cerrar el programa (Quit), se puede hacer click en ella o acceder a través del atajo de teclado Alt+Q. Y por último dentro del segundo menú Ayuda (Help) solo hay una opción para conocer más sobre el programa (About), cuando se hace click en ella se abre una nueva ventana con información sobre el programa.

Para empezar a operar con el programa hay que abrir una imagen como se describió anteriormente, una vez se abra la imagen, en la barra inferior de la ventana se mostrará la ruta absoluta de la misma.

Figura 2.5-3 Ruta del fichero

En la zona anteriormente vacía está dibujada la imagen, ocupando todo el espacio disponible pero siempre manteniendo la relación de aspecto original, además se redimensiona con la ventana del programa para adaptarse a todos los tamaños de pantalla. Ahora se puede hacer click en zonas de la imagen que se marcarán como el tipo de poro que esté seleccionado, no solo se marcará el punto sino un círculo de puntos alrededor de éste, y el programa dibujará un indicador en forma círculo del color del tipo de poro seleccionado, para que el usuario pueda saber dónde ha hecho click y los puntos que se han seleccionado, además en segundo plano el programa guardará esos puntos para más adelante usarlos en la división de la imagen.

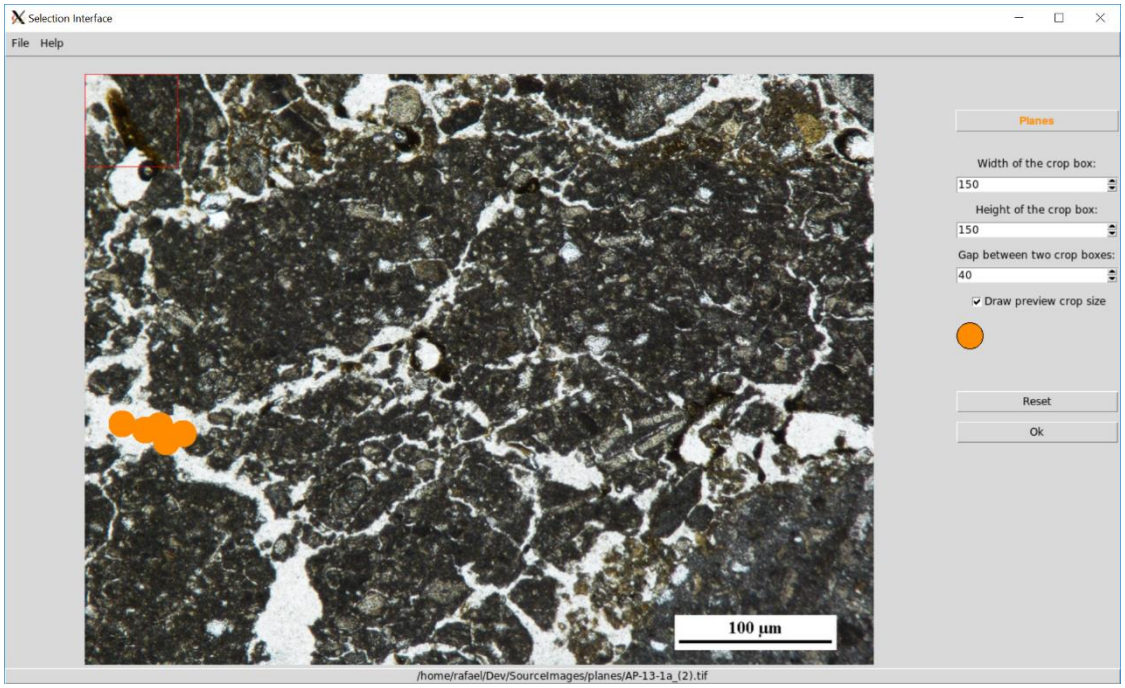

Figura 2.5-4 Uso de la interfaz de clasificación

Ahora analicemos las opciones de la derecha de la ventana, en primer lugar hay un selector, si se hace click en él se puede cambiar el tipo de poro que se va a marcar en la imagen. Una imagen puede contener los 6 tipos de poros. Cada una de las opciones del selector está coloreado con el mismo color con el que se marcará en la imagen con el indicador, al seleccionar una opción se cambiará el color del indicador y el tipo con el que guarda internamente el programa las coordenadas del punto.

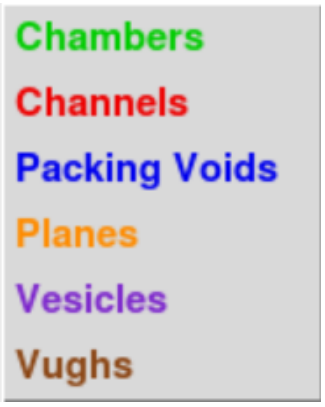

Figura 2.5-5 Selector de poro

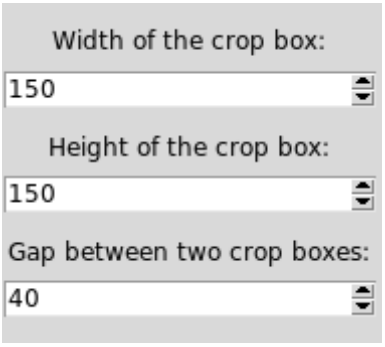

Figura 2.5-6 Dimensiones del corte

Debajo de ese selector hay 3 selectores numéricos, el primero para seleccionar el ancho de los trozos de imagen una vez dividida, el segundo para seleccionar el alto de los trozos de imagen una vez dividida, por defecto ambos están asignados a 150 pero se pueden modificar hasta un mínimo de 50 y un máximo de 250, el último de los 3 es para seleccionar es desplazamiento de los trozos con respecto al anterior, por defecto este campo está asignado a 40 pero se puede modificar hasta un mínimo de 10 y un máximo de 150. Cuando se divide la imagen se pasa un cuadro deslizante por ella del tamaño seleccionado por el usuario en los dos selectores anteriores, cuando esa ventana deslizante genera un nuevo trozo de imagen, se desplaza la cantidad seleccionada por el usuario en el último selector y genera un nuevo trozo de imagen, y así hasta que recorre toda la imagen original.

Debajo de estos 3 selectores numéricos hay dibujada una pista del indicador que se pintará en la imagen, esto es útil para el usuario para ver en un simple vistazo el tipo de poro seleccionado por el color del indicador, y también le permite al usuario saber el tamaño del indicador, ya que como se explicó antes el usuario cuando selecciona un punto en realidad marca un circulo de puntos alrededor del seleccionado. El diámetro del círculo de puntos que se seleccionan alrededor del seleccionado y el diámetro del indicador que se dibuja en la imagen, se puede aumentar o disminuir dentro de unos límites con la rueda del ratón, girando la rueda del ratón hacia delante se puede aumentar y girándola hacia detrás se puede disminuir, y gracias a esta pista el usuario siempre puede saber el tamaño real que tendrá la selección que quiere hacer en la imagen antes de hacerla.

Al final de las opciones de la derecha de la ventana hay dos botones, el primero es el de Reset y sirve para borrar visualmente e internamente todos los puntos seleccionados en la imagen, y empezar a seleccionar poros de nuevo desde cero en la imagen, el segundo botón es el de Ok y sirve para aceptar la selección hecha sobre la imagen, una vez se hace click sobre el programa pregunta el directorio donde se desea guardar los trozos de imagen generados.

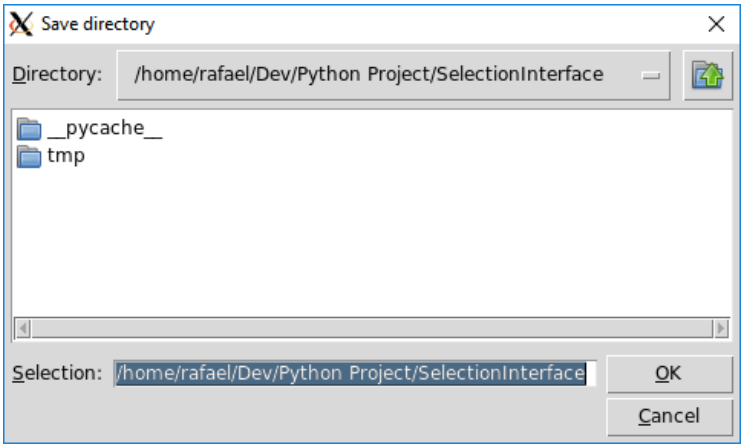

Figura 2.5-7 Ventana de selección de salida

Una vez seleccionado el directorio destino el programa empieza a dividir la imagen en trozos, y a guardarlos dentro de directorios con el nombre del tipo de poro dentro del directorio destino. Para dividir la imagen llama a un método con la imagen original, las opciones de ancho y alto de la ventana deslizante, la opción de desplazamiento de la ventana, el vector de puntos seleccionados y el directorio destino. Este método recorre la imagen original con una ventana deslizante de las características que recibió, y en cada deslizamiento comprueba si la ventana deslizante en la posición actual contiene algún punto seleccionado por el usuario, si es así guarda ese trozo de imagen en un directorio con el nombre del poro seleccionado, si no es así guarda ese trozo en un directorio con el nombre de Vacío (Empty).

Este proceso hay que hacerlo con cada una de las 200 imágenes, algo que conlleva mucho tiempo, pero mucho menos que hacerlo manualmente.

2.6 Desarrollo de las redes

Una vez obtenidas todos los 600.000 trozos de imágenes de los 6 tipos tenía que desarrollar la red neuronal que clasificase las imágenes, pero antes de empezar a desarrollarla tuve que decidir entre 3 posibles caminos. El primer camino posible era el de desarrollar y personalizar una red neuronal para que recibiendo una foto me dijera que tipo de poros contenía. El segundo camino posible era el de desarrollar y personalizar un red neuronal por cada tipo de poro, y que cada una de ella solo me supiera decir si contenía ese tipo o no. El último camino posible era el de utilizar unas redes neuronales ya desarrolladas por profesionales y ya entrenadas en naves de servidores, y reentrenar parte de ellas para que fueran capaces de diferenciar los tipos de poros, como no me decidí por cuál era mejor opción decidí probar las 3 opciones y basarme en los resultados para saber con cuál me quedaría.

Al principio desarrollé una red neuronal que fuese capaz de decirme si un trozo de imagen contenía Planos o no, este es el modelo de red neuronal más básico que podía crear al principio, y lo usé para aprender cómo crear todo tipo de redes neuronales y las cosas básicas que deben de contener todos los tipos de redes que iba a usar.

La primera cosa básica que tuve que aprender fue como pasarle toda la información de los trozos de imágenes ya clasificados a la red neuronal, ya que como dije anteriormente una red solo es capaz de tratar la información contenida en los tensores, así que tenía que ser capaz de formatear la información de forma que la red la entendiese y la introdujese en un tensor gracias a la capa de entrada. Descubrí que la forma idónea era leer cada imagen y la carpeta en la que estaba contenida para saber cómo había sido clasificada, y crear dos vectores, uno que contuviese cada una de las imágenes como un vector tridimensional, y otro que contuviese para cada una de las imágenes del otro vector el tipo de poro que contiene la imagen. El problema es que este vector no podía contener sino números así que hice un transformación de tipo a número, esta transformación era automática e instantánea gracias a un vector que contenía los posibles tipos de poros (["Planes", "Chambers", "Channels", "Packing Voids", "Vesicles", "Vughs"]), el número asignado en el lugar del tipo solo era el índice que ocupaba el tipo en este vector. Un detalle importante era

que el vector de imágenes y el vector de tipos de poros tenían que mantener el mismo orden, ya que la única asociación entre imagen y su tipo era el índice de su posición en los vectores.

Una vez calculados estos dos vectores los convertí en vectores del paquete de Python NumPy [21], un requisito que me pedía Keras para poder leer sin problemas los vectores. NumPy es un paquete que añade una biblioteca de funciones para manejar vectores y matrices en Python a más alto nivel, y se ha tomado como un estándar para pasar información en vectores entre diferentes paquetes o librerías de Python.

Esta tarea es bastante pesada y hacerla cada vez que necesitara entrenar la red no es una opción viable por tiempos. Por ejemplo en los primeros casos que solo entrenaba con 40000 trozos de imágenes y el entrenamiento completo tardaba alrededor de 15 minutos, de los cuales 3 o 4 minutos eran dedicados solo a calcular la entrada. Parece poco tiempo pero si es necesario entrenar la red cientos de veces para personalizarla y más adelante entrenarla con 4 o 5 veces más la cantidad de trozos de imágenes eso supondría mucho tiempo perdido, así que lo que hice fue calcular los vectores de imágenes y de tipos de poros para cada tamaño de trozos una sola vez, y guardar los valores en un archivo. Todo esto gracias a usar el paquete NumPy que me permite guardar y leer vectores en archivos, el hacerlo reduce la tarea de entrada de datos a leer un vector en un fichero, lo que me redujo el tiempo dedicado a la entrada de datos a un minuto y medio, algo menos de la mitad que anteriormente.

Cabe destacar que la mejora de tiempos gracias a este método depende mucho de la velocidad de lectura del disco duro en el cual estén almacenados los archivos con los vectores. Con los vectores estaban almacenados en mi disco duro HDD, con velocidad de lectura de 150 MB/s, el tiempo se reducía de 3 o 4 minutos a 2 minutos y medio aproximadamente. Pero con los vectores almacenados en mi disco duro SDD, con velocidad de lectura de 550 MB/s, el tiempo se reducía de 3 o 4 minutos a un minuto o un minuto y medio aproximadamente.

Una vez resuelto el problema de obtener la información de la forma necesaria para que la capa de entrada pueda leerla, solo me faltaban dos cosas para que la capa de entrada interprete la información, normalizar la información e indicarle la forma que tendrán los datos de entrada. Para normalizar la

información solo es necesario transformar el vector a un tipo de dato flotante con la precisión deseada, 16, 32 o 64 bits y dividir todos los valores del vector entre 255 ya que estamos tratando imágenes RBG. Esto es porque cada píxel solo puede tener un valor entero entre 0 y 255 por cada canal, porque cada píxel es representado por 8 bits por canal, después de hacer la división todos los valores serán números flotantes entre 0 y 1, por lo que los valores quedarían normalizados. Para indicarle a la capa la forma que tendrán los datos de entrada hay que pasarle un conjunto de datos, hay dos posibles conjuntos para imágenes RGB, (3, Ancho píxeles, Alto píxeles) o (Ancho píxeles, Alto píxeles, 3), algunos datos están almacenados con los datos de los canales al principio y otros con los canales al final, y es necesario especificárselo a la capa de entrada.

Ya con ambos vectores preparados para ser introducidos en la red tuve que dividirlos, porque la red no solo hay que entrenarla sino también comprobar su precisión, para ello es normal dedicar un 20% del total de la información y entrenar con el otro 80% de la información. También para que la red no siempre entrene y compruebe con los mismos datos hice un método que reordenaba ambos vectores aleatoriamente cada vez, pero ambos vectores de la misma forma para no perder la relación de la información en ambos vectores, con esto cada vez que cogiera el 20% del vector serían imágenes diferentes. Una vez hecho todos estos pasos ya la red neuronal es capaz de interpretar la información y manipularla, pero aun la red no hace nada, así que ahora tuve que hacer que la red predijera imágenes.

Al principio hice la estructura básica que comparten la mayoría de redes neuronales convolucionales, para a la hora de personalizar la red poder partir de la parte invariable de la misma, por ello lo primero que hice fue añadir la capa de entrada, la capa de aplanamiento, la capa densa, la capa de desecho y la última capa densa.

En Keras el proceso de creación de la red empieza creando un modelo [22], en este caso de tipo secuencial (Sequential), y después añadiendo al modelo con el método add en orden cada una de las capas que contendrá la red. Los tipos de capas ya existen predefinidos en Keras, solo es necesario importarlos, crear el tipo de capa con los parámetros deseados y añadirlas al modelo.

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape, padding="same"))
model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', padding="same"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=3))
```

Figura 2.6-1 Creación del modelo

Por ejemplo la primera capa que añadí al modelo fue la capa de entrada, y lo hice llamando al constructor Input() con el parámetro forma (shape) con el valor (100, 100, 3), y así la red ya lee la información y la introduce en un tensor.

```
model.add(Input(shape=(100,100,3)))
```

Figura 2.6-2 Creación de capa de entrada

Esta capa no suele especificarse en Keras porque muchas de las capas pueden suplantar a esta añadiéndola internamente, se habilita pasándole el parámetro forma de entrada (input_shape) a la primera capa del modelo.

```
model.add(Conv2D(64, kernel_size=(4, 4), activation='relu', input_shape=input_shape, padding="same"))
```

Figura 2.6-3 Integración de la capa de entrada

La siguiente capa que añadí fue la capa de aplanamiento llamando al constructor Flattern() sin parámetros, lo que consigue es que la forma de la entrada de esta capa fuera (n, 100, 100, 3) y la forma de salida después de aplanarla fuera (n, 30000).

```
model.add(Flatten())
```

Figura 2.6-4 Creación de capa de aplanamiento

La siguiente capa que añadí fue una capa densa llamando al constructor Dense() con los parámetros número de neuronas (units) con valor 1024, y con el parámetro función de activación (activation) con valor ReLU. Esta capa se suele añadir en las redes convolucionales como último recurso para aprender algunas características no lineales que no ha aprendido la red en las capas previas de convolución.

```
model.add(Dense(1024, activation='relu'))
```

Figura 2.6-5 Creación de capa densa

La siguiente capa que añadí fue la capa desecho llamando al constructo Dropout() con el parámetro ratio o porcentaje de desecho (rate) con el valor 0.4, es decir, va a desactivar el 40% de las neuronas. `model.add(Dropout(0.4))`

```
model.add(Dense(num_classes, activation='softmax'))
```

Figura 2.6-6 Creación de capa densa final

La última capa que añadí fue una capa densa llamando al constructor como antes pero con el parámetro número de neuronas (units) con valor 1, en esta última capa equivale al número de posibles resultados, y con el parámetro función de activación (activation) con valor Softmax. Esta última capa es la que nos da los resultados de la predicción, en el caso de red que estaba diseñando si la neurona devuelve un valor pequeño cercano a 0 no contiene Planos y si devuelve un valor grande cercano a 1 contiene Planos.

Con la parte invariable de la red ya hecha solo necesitaba añadir la parte variable, y al principio para tener una red sencilla solo añadí a la red dos capas convolucionales y una capa de agrupamiento después de la capa de entrada. Las dos capas convolucionales las cree de forma idéntica, llamando a su constructor Conv2D() con los parámetros número de filtros (filters) con valor 32, el parámetro tamaño de filtro (kernel_size) con el valor (3, 3), es decir 3 de ancho por 3 de alto, el parámetro función de activación (activation) con el valor ReLU, y por último el parámetro relleno (padding) con el valor igual (same).

```
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', padding="same"))
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', padding="same"))
```

Figura 2.6-7 Creación de capas convolucionales

Este último parámetro sirve para mantener o no el tamaño de la imagen, porque cuando se aplica un filtro a una imagen si el filtro no es de tamaño 1 de alto y ancho no es posible calcular el filtro de los píxeles de los bordes de la imagen. Esto se debe a que si el filtro es una matriz de tamaño mayor que 1x1 al posicionar esta matriz sobre la imagen para calcular la imagen filtrada el filtro no puede sobresalir de la imagen, como se muestra en la imagen.

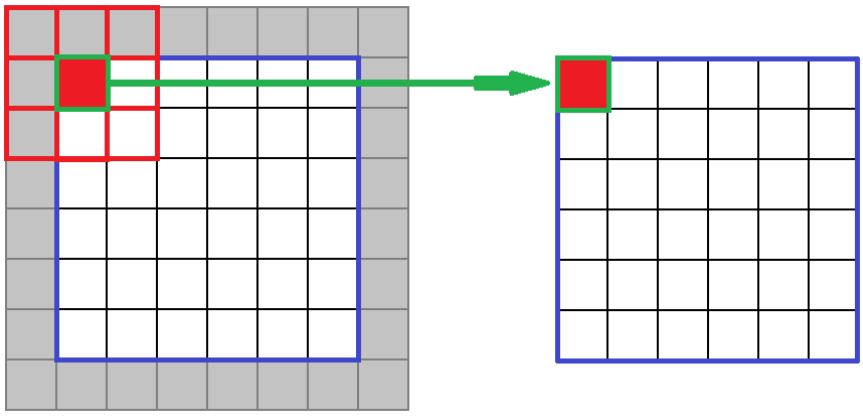

Figura 2.6-8 Filtro sobre una imagen

Por ejemplo en una imagen de 8x8 píxeles con un filtro de 3x3 que empieza en la esquina superior izquierda de la imagen, como en la figura superior, el filtro solo se puede mover 3 píxeles hacia la derecha desde la columna inicial y 3 píxeles hacia debajo desde la fila inicial. Solo da como resultado 6 cálculos en cada sentido, o una imagen filtrada de 6x6 píxeles. Entonces en con el valor igual para este parámetro, el programa incorpora en los bordes de la imagen las filas y columnas de ceros necesarias para que la imagen filtrada mantenga la misma dimensión que la imagen original.

Y para completar la red tras la última capa convolucional añadí la capa de agrupamiento llamando al constructor MaxPooling2D() con los parámetros tamaño de filtro de agrupamiento (pool_size) con el valor (2, 2), es decir 2 de ancho y 2 de alto, y el parámetro separación (stride) con el valor 2. Con estos parámetros la capa de agrupamiento recorrerá el resultado de la capa anterior con un filtro de tamaño 2 de ancho y alto. De esos 4 valores obtendrá un nuevo valor que será el máximo de ellos, y como la separación es 2 moverá el filtro en saltos de 2 píxeles en la horizontal y vertical, esta capa reducirá el tamaño de la salida 4 veces.

```
model.add(MaxPooling2D(pool_size=(2, 2), strides=2))
```

Figura 2.6-9 Creación de capa de agrupamiento

Con todas estas capas añadidas al modelo la red más básica ya estaba creada, después de esto tenía que compilar la red y empezar a entrenarla. Compilar la red es necesario para hacer el modelo definitivo, para especificar la función de pérdida o de costo, el optimizador a utilizar y las métricas a obtener del entrenamiento de la red. Para compilar la red de ejemplo llamé al método

compile() del modelo pasándole como parámetros la función de costo [23] entropía binaria cruzada (binary_crossentropy), utilicé el optimizador [24] Adamax y obtuve la métrica de precisión [25] (accuracy).

```
model.compile(loss=keras.losses.binary_crossentropy,
              optimizer=keras.optimizers.Adamax(),
              metrics=['accuracy'])
```

Figura 2.6-10 Compilación del modelo

Una vez compilada la red solo hacía falta elegir el valor de tres parámetros necesarios para entrenar la red, el primero es el tamaño del grupo de entrenamiento, que es el conjunto de imágenes que están a la vez en la memoria dinámica de la GPU y que son introducidas juntas a la red. Este parámetro no modifica los resultados de precisión de la red pero si la velocidad de entrenamiento de la misma por la velocidad de los accesos a las diferentes memorias. El segundo es el número de iteraciones del entrenamiento con el mismo grupo de imágenes, y por último tuve que elegir si la red reordena los vectores o no. Después de esto ya pude empezar a entrenarla pasándole los dos vectores de entrenamiento, los dos vectores de comprobación, el tamaño del grupo de entrenamiento (batch_size) con el valor 100, el número de

```
model.fit(x_train, y_train,
          batch_size=100,
          epochs=20,
          verbose=2,
          shuffle=True,
          validation_data=(x_test, y_test)
          )
```

Figura 2.6-11 Entrenamiento del modelo

entrenamientos de la red (epochs) con el valor 20 y el parámetro de reordenación (shuffle) siempre con el valor si o verdadero.

Tras entrenar la red neuronal, para obtener los valores de precisión obtenidos por esta red la evalué con los vectores de comprobación con el método evaluate(), el cual devuelve un valor de precisión entre 0 y 1 referente al porcentaje de acierto en la evaluación. Los valores deseados es que acierte más de un 50% de los casos, lo que implica que da resultados aleatorios, aunque cuanto más cerca del 100% mejor.

```
score = model.evaluate(x_test, y_test, verbose=0)
```

Figura 2.6-12 Evaluación del entrenamiento

Esta red al principio tenía una precisión de casi 60%, pero se la puede mejorar, pero fue necesario personalizar todas las variables de ella para conseguir el mayor rendimiento posible para esta tarea. Para obtener los valores que mejoraran la precisión en esta tarea hice un script que entrenara la red continuamente con diferentes valores, y que escribiera en un archivo los resultados de cada uno de los entrenamientos y comprobaciones, y después comprobé estos valores para hallar la combinación de valores más precisa. Además este script guarda el progreso de la ejecución automáticamente tras cada entrenamiento ya que todos los entrenamientos conllevan mucho tiempo seguido y puede suceder algún error o algún problema que apague el ordenador.

Pero antes de empezar a personalizar las redes decidí terminar de crear los tres tipos de redes, así que a partir de esta red binaria empecé a hacer una red neuronal que ella sola fuera capaz de decirme cualquiera de los tipos de poros que aparecen en la imagen. Desarrollar esta red a partir del modelo anterior es bastante fácil ya que solo hace falta cambiar el número de neuronas que contiene la última capa densa al número de nuevos tipos que tiene que diferenciar la red, en este caso 7, y además en el momento de compilarla en el parámetro de la función de costo hay que usar la entropía categórica cruzada (categorical_crossentropy). Con esta red también realicé un script que entrenara la red continuamente con diferentes valores y que escribiera en un archivo los resultados de cada uno de los entrenamientos y comprobaciones, y debes comprobé estos valores para hallar la combinación de valores más precisa.

Aunque aún me quedaba un tipo de red por desarrollar, o en este caso, por aprender a entrenar, me faltaba hacer un script para entrenar redes desarrolladas para concursos por profesionales. Este tipo de redes es diferente porque no tuve que hacerlas yo mismo, tuve que obtener sus modelos y los pesos que obtuvieron después de entrenarla, pero tenía que modificarlas un poco para mi objetivo.

Dentro de Keras están incluidas 10 de las redes con mejores resultados en el mundo de las redes neuronales, aunque para el caso de ejemplo solo usare la red Xception, actualmente la más precisa con respecto al peso, tamaño y velocidad, y de las más novedosas porque usa capas convolucionales separables.

Para obtener el modelo de esta red tuve que llamar a su constructor Xception() con el parámetro pesos (weights) con el valor imagenet, esto es para

especificarle que utilice los pesos que obtuve cuando la entrenaron con la base de datos de imágenes ImageNet. También le pasé el parámetro incluir encima (include_top) con el valor falso para especificarle que esta red será usada como la base, y no será usada como continuación de otra red previa, este parámetro solo cambia el que incluya o no la capa de entrada. Le pasé el parámetro forma de la entrada (input_shape) con la forma de entrada de datos que tendría, este parámetro solo es necesario si el parámetro incluir encima es falso. Y por último le pasé el parámetro agrupamiento (pooling) con el valor max para que las capas de agrupamiento calculen el píxel agrupado con el máximo de los anteriores.

```
base_model = Xception(weights="imagenet", include_top=False, input_shape=(img_rows, img_cols, 3), pooling="max")
```

Figura 2.6-13 Obtención de redes pre-entrenadas

Con el modelo de la red almacenado le añadí una capa densa llamando al constructor Dense() con los parámetros número de neuronas (units) con valor 1024 y con el parámetro función de activación (activation) con valor ReLU. Y le añadí una segunda y última capa densa llamando al constructor como antes pero con el parámetro número de neuronas (units) con valor 7, el número de tipos de poros posibles, y con el parámetro función de activación (activation) con valor Softmax. Esta última capa es la que nos da los resultados de la predicción, en este caso es necesaria para personalizar la red a nuestro objetivo de predicción.

```
x = base_model.output
x = Dense(1024, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)
```

Figura 2.6-14 Modificación de redes pre-entrenadas

Con esto ya tenía la red personalizada y lista para compilar y entrenar, sin embargo, después de personalizarla tenía una red con más de 100 capas de las cuales solo 2 no habían sido entrenadas, y las otras habían sido entrenadas para otro objetivo, así que tenía que reentrenar la red pero aprovechando el entrenamiento previo con ImageNet que yo no podía replicar. Para ello tenía que entrenar algunas capas y otras no, pero no es un número exacto así que hice un script que entrenara continuamente todas las redes probara entrenando más o menos capas en cada iteración, y así pude obtener que cual era la opción más precisa.

```
for layer in model_pre.layers[:50]:
   layer.trainable = False
for layer in model_pre.layers[50:]:
   layer.trainable = True


model_pre.compile(optimizer=optimizers.Adamax(),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

model_pre.fit(x_train, y_train,
              batch_size=100,
              epochs=25,
              verbose=2,
              shuffle=True,
              validation_data=(x_test, y_test)
              )

score = model_pre.evaluate(x_test, y_test, verbose=0)
```

Figura 2.6-15 Entrenamiento parcial de redes pre-entrenadas

2.7 Entrenamiento de la red

Para entrenar estas redes y que aprendan cómo funcionar se les proporcionas muchos ejemplos de entrada y sus salidas correctas, las redes comienzan con unos pesos por valor aleatorios y con ellos intentan predecir la salida. Según van iterando con equivocaciones y aciertos van modificando estos pesos ligeramente para intentar predecir con mayor precisión y más frecuentemente el resultado correcto. Para ellas saber si los resultados que están prediciendo son correctos, y para saber cómo esta mejora su precisión utilizan una función de costo que intentan minimizar, cuando ésta es muy pequeña entonces la red está acertando y su precisión está siendo alta. Tras muchos ejemplos de práctica la red es capaz de predecir los resultados de forma rápida y precisa, sin embargo, ésta no es una tarea fácil ya que hay redes capaces de clasificar imágenes en más de 18000 categorías con un porcentaje de acierto de alrededor del 96%. Para conseguir estos resultados es necesario entrenarlas con alrededor de 350 millones de imágenes diferentes, una tarea muy pesada que actualmente se lleva a cabo en naves de servidores con potencias de cómputo inmensas e imposible de conseguir excepto para empresas muy grandes.

Así que ya terminado el proceso de desarrollo de las redes empecé a entrenar las redes con los script, para hallar que tipo de red y con qué parámetros conseguía los mejores resultados para mi problema, pero en los script de los diferentes tipos de redes se personalizaban diferentes variables.

En el script para las redes de profesionales solo se modificaban las variables de tamaño de los trozos de imágenes de entrada que podía ser 100x100, 150x150 o 200x200. La variable de la precisión de los datos a usar que podía ser flotantes de 16, 32 o 64 bits. La variable del número de capas totales a entrenar que variaba entre 0 y el número total de capas en la red. La variable del número de entrenamientos de la red que podía ser 10, 15, 20 o 25. Y la variable del optimizador que podía ser SGD, Adadelta, Adagrad, Adam, Adamax o Nadam.

En los script para las otros dos tipos de redes se modificaban muchas más variables, y en ambos script las mismas variables y con los mismos posibles valores para estas. Las variables que se modifican son el tamaño de los trozos de imágenes de entrada que podía ser 100x100, 150x150 o 200x200. La variable de la precisión de los datos a usar que podía ser flotantes de 16, 32 o 64 bits. La variable del número de filtros que hay en las capas convolucionales que podía

ser 32 o 64. La variable de los optimizadores que podía ser SGD, Adadelta, Adagrad, Adam, Adamax o Nadam. La variable del número de entrenamientos de la red que podía ser 10, 15, 20 o 25. La variable del porcentaje de desecho en la capa de desecho que podía ser 0%, 40%, 60% y 80%. La variable de la separación entre los filtros de la capa de agrupamiento que podía ser 2, 3 o 4. La variable del tamaño de los filtros en la capa de agrupamiento que podían ser de 2x2 o 3x3. La variable del tamaño de los filtros en la capa convolucional que podía ser desde 3x3 hasta 10x10, siempre teniendo el mismo ancho que alto. Y la variable del número de capas convolucionales que podía ser desde 1 hasta 12.

Además de todas las posibles combinaciones, para cada posibilidad se entrenó y comprobó la red 3 veces, para evitar que la aleatoriedad y posibles fallos en el entrenamiento influyeran en los resultados y así eliminar errores en las conclusiones obtenidas.

Al principio me pareció buena idea no cerrarme puertas y abarcar todas las posibilidades pero como se ha podido ver en lo hasta ahora son muchas, cuando acabé de hacer el script me di cuenta y me puse a hacer cálculos y con la cantidad de posibilidades que había puesto yo. Calculé que cada una de las redes excepto las de los profesionales tenía alrededor 2.500.000 de posibilidades diferentes, por lo que optimicé lo más posible la red para que el entrenamiento fuera lo más rápido posible, y evalué cuanto tardaba en entrenar esta tras los cambios.

Pero el entrenamiento es un proceso bastante costoso en tiempo de cómputo, por ejemplo yo la entrené con una GTX 1070, con este modelo de GPU cada entrenamiento tardaba alrededor de 20 minutos. Aunque varía según la precisión en la que estuvieran los datos por las velocidades de transferencia de las memorias y por la potencia de cómputo de la GPU con cada precisión. En el modelo con los datos en la precisión flotante de 16 bits el entrenamiento tarda 15 minutos con la potencia de cómputo de la GPU de 107 GFLOPS. En el modelo con los datos en la precisión flotante de 32 bits el entrenamiento tarda 20 minutos con la potencia de cómputo de la GPU de 6,8 TFLOPS. En el modelo con los datos en la precisión flotante de 64 bits el entrenamiento tarda 25 minutos con la potencia de cómputo de la GPU de 214 GFLOPS. Por lo que de media entrenar una red tarda 20 minutos aproximadamente.

Con estas mediciones calculé que era imposible calcular todas las posibilidades, para estos tiempos el calcular todas la posibilidades para solo un tipo de red me conllevaría alrededor de 95 años, y calcular todas las posibilidades para todos los tipos me conllevaría algo más de 200 años, Ya que solo disponía de unos meses esto era inviable, esta era una tarea para superordenadores, por ejemplo para observar la falta de potencia de cómputo que tenía, en el superordenador IBM Summit [26] inaugurado en 2018 tiene una potencia de cómputo de 122,3 PFLOPS o 122.300 TFLPOS para estar en el mismo orden de magnitud de mi GPU. Asumiendo que el entrenamiento de todas las combinaciones se hiciera con la potencia de mi GPU para precisión de 32 bits que es la más potente. El superordenador es 18.000 veces más rápido, lo que significa que un entrenamiento de la red no tardaría 20 minutos sino 66 milisegundos, lo que extrapolado al entrenamiento de todas las posibilidades, lo que a mí me conllevaba 95 años a este superordenador le conllevaría algo menos de 2 días.

Tras ver estos números decidí no probar todas las posibilidades sino ir poco a poco descartando posibilidades según los resultados, es decir, primero pobre la red con las combinaciones solo de los posibles optimizadores, posibles separaciones entre los filtros de la capa de agrupamiento, posibles tamaños de los filtros de la capa de agrupamiento, posibles porcentajes de la capa de desecho y posibles número de capas. Una vez probadas estas 1728 posibilidades que me conllevaron de 24 días de cómputo reduje los posibles valores de las variables probadas, para la siguiente prueba de los 6 posibles optimizadores me quede solo con 3, de los 2 posibles tamaños de filtros me que con 1 solo, de las 3 posibles separaciones entre los filtro me quede con 1 sola, de los 4 posibles porcentajes de desecho me quede con 2 solo y por último de los 12 posibles números de capas me quede con 6 solo. Esto reduce las 1728 posibilidades a 36 para el próximo paso. Y así paso a paso hasta comprobar todas las variables.

2.8 Resultados de la red

Continuando con esta estrategia fue posible comprobar con las diferentes posibilidades los resultados de la red, y llegue a la conclusión de que los mejores resultados de precisión me los ha dado la red neuronal creada por profesionales VGG16, esta ha sido capaz de predecir el resultado correcto con un precisión de casi el 90%.

Este resultado está muy cerca del 100%, lo que sería una predicción exacta, pero poniéndolo en contexto es una precisión bastante más alta de lo que parece. La peor precisión posible es un 14,3%, que significa aleatoriedad en la predicción, esto es así porque el objetivo es predecir 7 resultados posibles, la presencia o no de un poro y el tipo del mismo. Observando esto la precisión es muy superior a la aleatoriedad. Además la red no revisa una imagen individual, sino que divide la imagen en trozos y analiza un cada uno de ellos, por lo tanto ese 11% de error se disipa, porque un mismo poro aparece en docenas de trozos diferentes y la probabilidad de que no lo detecte al menos en la mayoría de ellos es casi 0%.

Los resultados que obtuve de entrenar y personalizar las redes creadas por profesionales fueron los mejores, aunque la que mejor resultados obtuviese fue la red VGG16 muchas otras dieron buenos resultados también.

Red	**Tamaño imagen**	**Capas entrenadas**	**Numero entrenamientos**	**Precisión**
Xception	100x100	72 últimas	25	70%
NASNetLarge	100x100	642 últimas	25	57%
NASNetMobile	100x100	412 últimas	25	53%
InceptionV3	150x150	144 últimas	30	58%
InceptionV2	150x150	433 últimas	30	47%
VGG16	150x150	11 últimas	25	89%
VGG19	100x100	19 últimas	30	43%

Tabla 2.8-1 Resultados redes entrenadas previamente

Como se puede observar por los resultados expuestos se obtienen mejores precisiones con tamaños de imagen pequeños, las pruebas con imágenes de tamaño 200x200 y 250x250 han dado peores precisiones y peores tiempos de entrenamiento. Además se puede observar que cuanto mayor es el número de entrenamientos mejor es la precisión, desde 5 entrenamientos en adelante crece la precisión, pero deja de crecer cuando se acerca a 25 o 30 entrenamientos.

Todas las redes fueron entrenadas con diferentes optimizadores pero el que mejores resultados ofreció fue el optimizador Adamax en la mayoría de ellas, excepto en la red VGG16, en la cual el optimizador SGD fue un poco superior en la precisión final a pesar de ser más inestable durante el entrenamiento.

En las siguientes gráficas se puede observar una comparación entre las precisiones y pérdidas de la red VGG16 cambiando el optimizador, en color azul la red entrenada con el optimizador SGD y en color rojo la red entrenada con el optimizador Adamax.

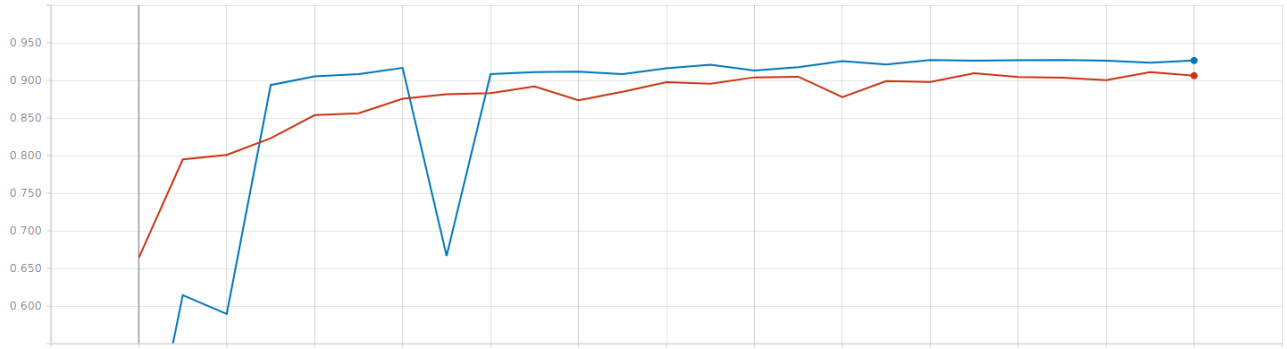

Figura 2.8-2 Comparación de precisión entre optimizadores en VGG16

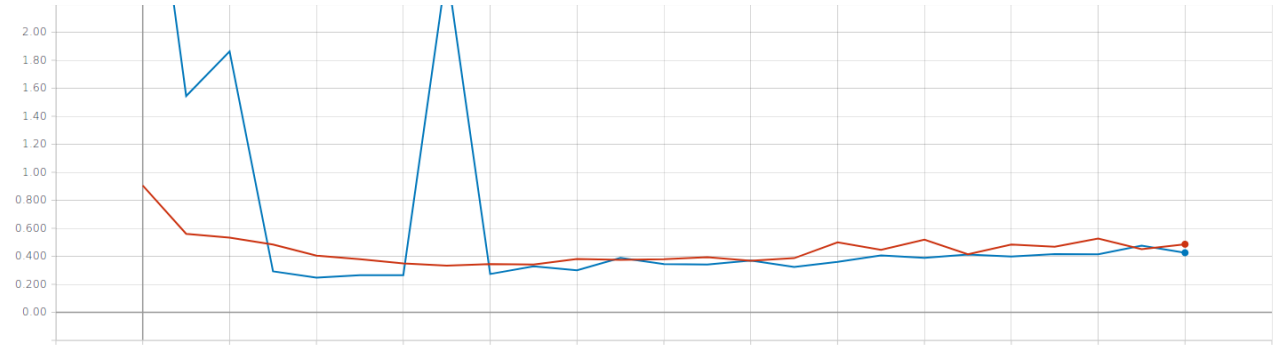

Figura 2.8-1 Comparación de pérdidas entre optimizadores en VGG16

Aunque la que red que mejores precisiones ha dado fue VGG16, la red no binaria creada por mí también obtuvo resultados cerca del 85% precisión, esto fue posible gracias a personalizarla con las pruebas. Con ello descubrí que los mejores resultados los obtenía usando imágenes de tamaño 150x150 con el tipo de dato float32. Utilizando 10 capas convolucionales con 64 filtros de tamaño 4x4, y 4 capas de agrupamiento con filtros de tamaño 2x2 y separación 3. Usando también una capa de desecho del 40% y entrenando la red con el optimizador Adamax durante 25 entrenamientos.

En las siguientes gráficas se puede observar la progresión estable de la precisión y la pérdida durante los entrenamientos.

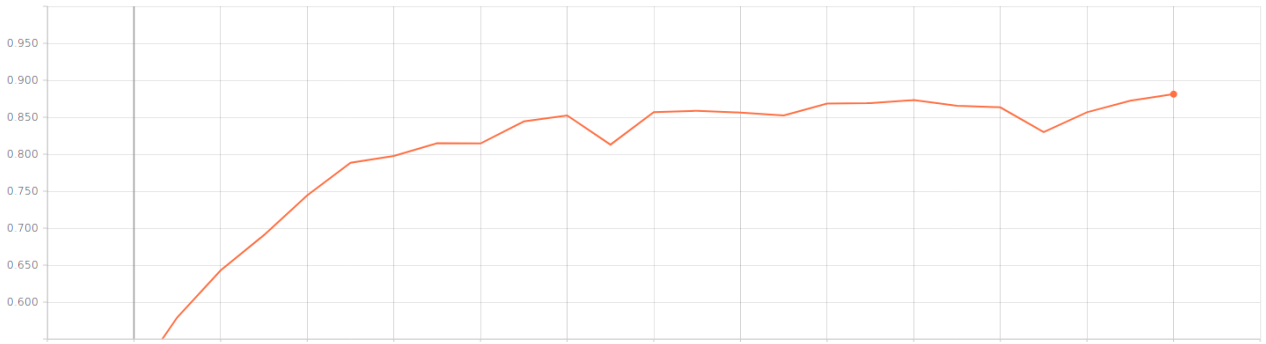

Figura 2.8-4 Precisión de la red no binaria

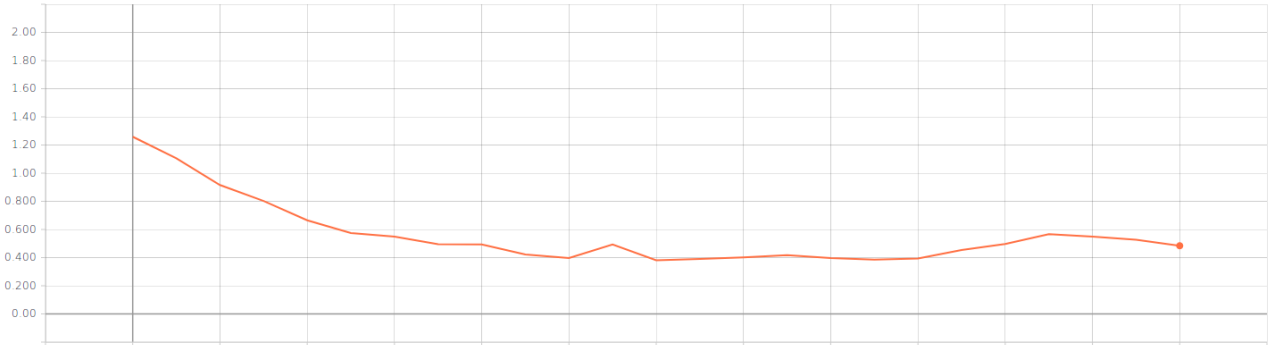

Figura 2.8-3 Perdidas de la red no binaria

Sin embargo la red binaria creada por mí no obtuvo tan buenos resultados, sus mejores resultados en precisión se quedaron alrededor del 60%. En esta red no afectó mucho la personalización ya que los resultados variaban muy ligeramente con los cambios, pero la mejor configuración fue muy similar a la de la red no binaria, solo que usando 7 capas convolucionales con 32 filtros y durante 20 entrenamientos.

Los motivos que han limitado este resultado final a una precisión del 89% son principalmente 2. El primero de los motivos es la falta de conocimientos en arqueología, durante este proyecto he tenido que dividir y clasificar imágenes de poros arqueológicos, esta tarea es muy importante ya que es la fuente de datos de la red, y uno de los factores que más puede afectar a los resultados de la red, por lo que es muy importante hacer una buena clasificación para obtener buenas predicciones, yo creo que he hecho una muy buena clasificación y en parte por ello los buenos resultados obtenidos, pero es innegable que no tengo conocimientos de arqueología por lo que si la clasificación de las imágenes la hiciera una persona experta en el ámbito el resultado de precisión aumentaría.

El segundo de los motivos es la falta de medios y herramientas, éste proyecto ha sido desarrollado en un ordenador doméstico con una potencia de cómputo muy pequeña, por lo que ha sido imposible comprobar todas las opciones de personalización para adaptarlas lo mejor posible al cometido de esta red, por lo tanto si el entrenamiento y personalización hubiera sido hecho en ordenadores más potentes el resultado de precisión aumentaría.

2.9 Visualización de los resultados

Una vez ya tenía la red personalizada y entrenada tuve que crear un programa en el cual se pudiera cargar una imagen y que diera como resultado los tipos de poros presentes en la imagen. Para esto partí del programa que había creado para generar las imágenes de entrenamiento de la red y lo modifiqué para que cumpliera la nueva función.

Primero le quité toda la funcionalidad de marcar en la imagen y de generar los trozos de la imagen, y en la interfaz quité las opciones de la parte derecha de la ventana, y por último limpié el programa de código residual.

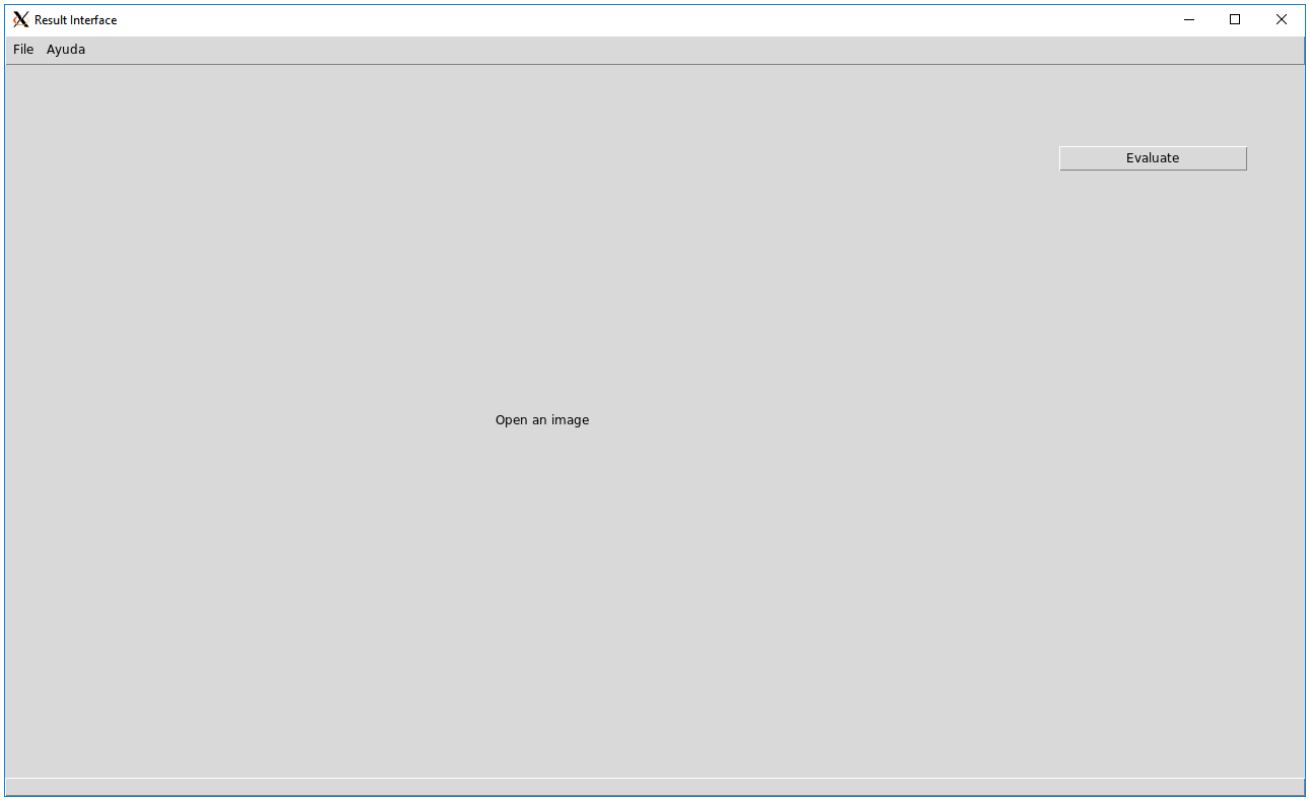

Figura 2.9-1 Interfaz de resultados

Después de la limpieza añadí a la derecha de la ventana un botón de calcular y unas etiquetas para mostrar los resultados de la predicción, y añadí la nueva funcionalidad. Para predecir los resultados tuve que reproducir parte de los pasos usados para entrenar y probar la red, es decir, una vez el usuario le daba a calcular el programa tenía que dividir la imagen en trozos, crear los vectores para los trozos, tratar los vectores para introducírselos a la red, crear el modelo de la red y compilar la red. Pero ahora no quería entrenar la red sino

usar la red ya entrenada, para eso tuve que entrenar la red más precisa de nuevo y después de entrenarla tuve que guardar los pesos finales en un fichero en formato h5 para que el programa pueda cargar los pesos y comprobar el resultado que da la red para cada uno de los trozos.

Tras adquirir los resultados para cada trozo tuve que tratarlos para que fueran porcentajes de 0 a 100% y excluyera los resultados de Vacíos ya que no se quiere saber cuento es el porcentaje aparición de cada tipo de poro en la imagen sino el porcentaje de aparición de cada tipo con respecto a los otros.

```
for x in range(0, len(x_test)):
    test = x_test[0].reshape(1, img_rows, img_cols, 3)
    score += model.predict(test)[0]

score /= len(x_test)
diff = 1 - score[0]
score /= diff
score *= 100
```

Una vez calculados todos estos valores hice que se mostraran en la interfaz, debajo del botón calcular, indicando al usuario el resultado.

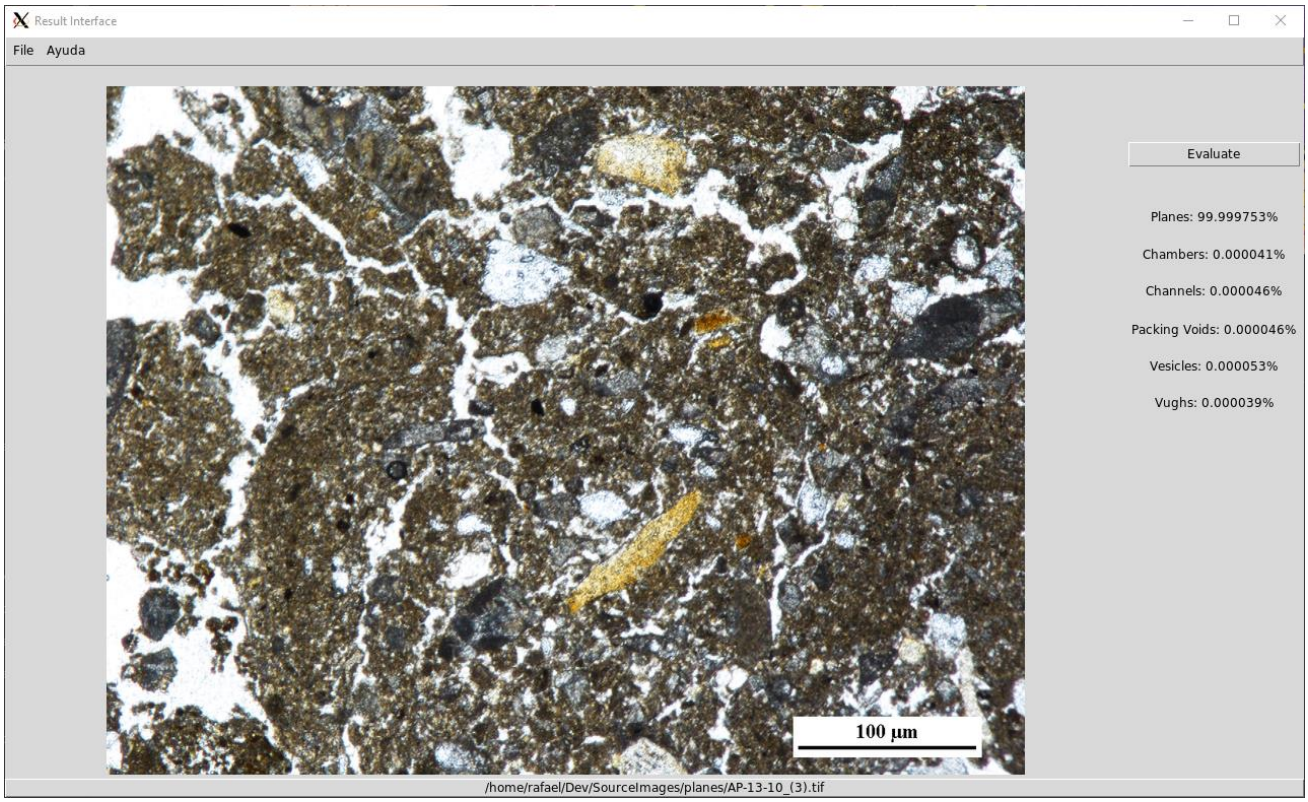

Figura 2.9-2 Uso de la interfaz de resultados

Capítulo 3
Conclusiones y líneas futuras

En conclusión me ha gustado el trabajo que he realizado porque me ha introducido en el mundo de la inteligencia artificial, y me ha permitido aprender mucho sobre este campo en crecimiento.

Con este trabajo me he dado cuenta de las dificultades que tiene el uso de inteligencia artificial en proyectos, sobre todo para la clasificación de imágenes. También he descubierto los avances que se han hecho en la clasificación de imágenes, sobre todo estos últimos 15 años en los que ha avanzado mucho.

Además he comprobado por que no todo el mundo puede usar inteligencia artificial, la magnitud de potencia de cómputo necesaria para crear redes con algo de complejidad es enorme.

Los posibles avances que se le pueden hacer al proyecto para mejorar su funcionamiento son clasificar más imágenes para entrenar la red con más datos y mejorar la precisión, otra mejora podría ser integrar la red neuronal en servidores como los que ofrece Amazon para poder tener más potencia de cómputo.

Los posibles avances para mejorar su uso por el usuario son mejorar las interfaces gráficas, para hacer su uso más fácil para personas que se inicien en la inteligencia artificial, y también agregar un mapa de resultados que coloree la imagen según la predicción de la red para que el usuario pueda observar qué predice la red en que zonas de la imagen.

Capítulo 4
Summary and Conclusions

In conclusion, I liked this work because it introduced me to the world of artificial intelligence and allowed me to learn a lot about this growing field.

I have realized the difficulties in the use of artificial intelligence has in projects, especially for the classification of images. I have also discovered the progress that has been made in the classification of images, especially in the last 15 years when it has made great progress.

I also found out why not everyone can use artificial intelligence, the magnitude of computing power needed to create neural networks with some complexity is enormous.

The possible steps the project can take to improve its performance are to classify more images to train the network with more data and improve accuracy, another improvement could be to integrate the neural network in cloud servers such as those offered by Amazon to have more computing power.

Possible steps to improve its use by the user are to improve the graphical interfaces, to make it easier to use for people who are new to artificial intelligence, and also to add a results map that colors the image according to the network prediction so that the user can see what the network predicts in which areas of the image.

Capítulo 5
Presupuesto

5.1 Presupuesto personal

Descripción	Cantidad	Precio
Horas de trabajo	500	3500€

Tabla 5.1-1 Presupuesto

Capítulo 6
Apéndice

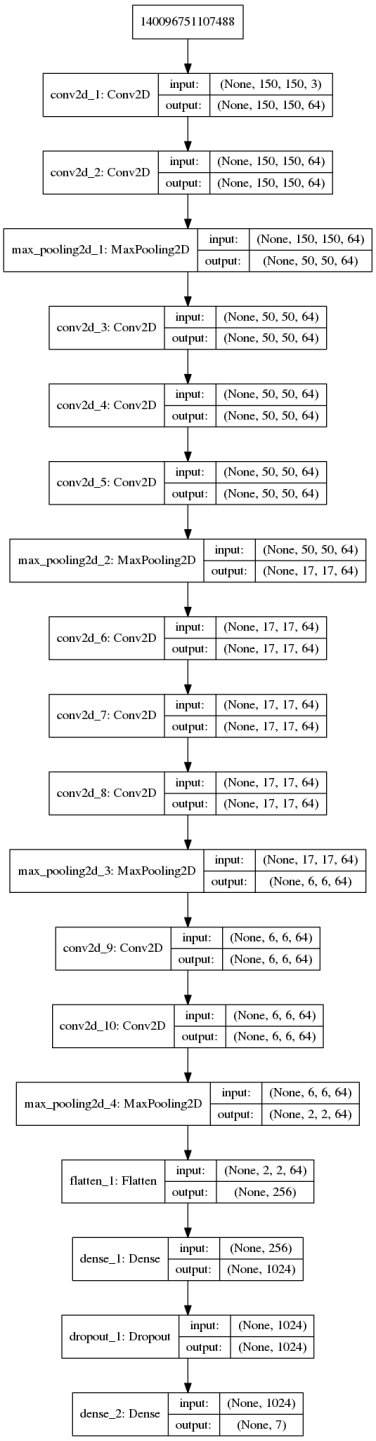

Figura 5.1-2 Estructura red no binaria

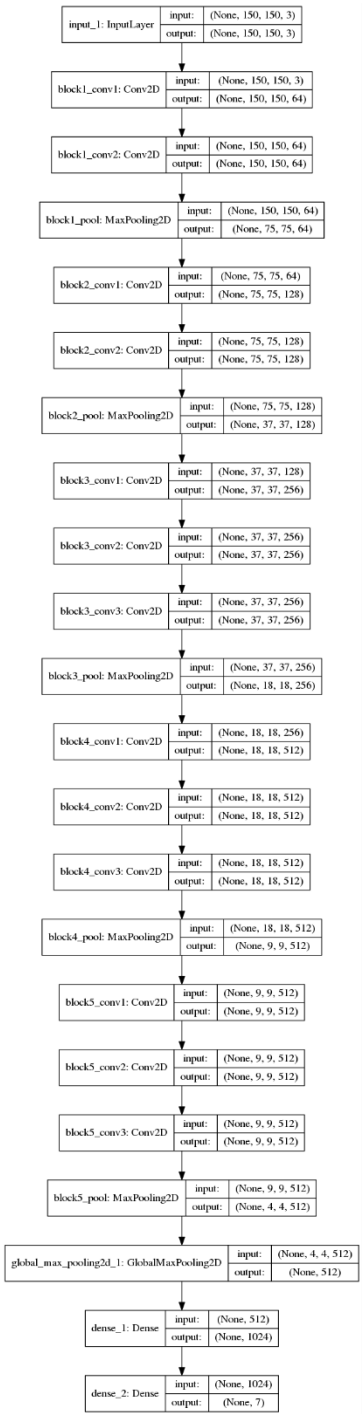

Figura 5.1-1 Estructura red VGG16

Capítulo 7
Bibliografía

[1] https://en.wikipedia.org/wiki/Activation_function

[2] https://analyticsindiamag.com/most-common-activation-functions

[3] https://keras.io/layers/core/#input

[4] https://keras.io/layers/core/#dense

[5] https://keras.io/layers/core/#dropout

[6] https://keras.io/layers/core/#flatten

[7] https://keras.io/layers/core/#reshape

[8] https://keras.io/layers/convolutional/

[9] https://keras.io/layers/pooling/

[10] https://keras.io/layers/core/#dense

[11] https://keras.io/applications/

[12] https://pytorch.org/

[13] http://caffe.berkeleyvision.org/

[14] http://deeplearning.net/software/theano/

[15] https://www.microsoft.com/en-us/cognitive-toolkit/

[16] https://www.tensorflow.org/

[17] https://www.tensorflow.org/guide/summaries_and_tensorboard

[18] https://www.python.org/

[19] https://keras.io/

[20] https://wiki.python.org/moin/TkInter

[21] http://www.numpy.org/

[22] https://keras.io/getting-started/sequential-model-guide/

[23] https://keras.io/losses/

[24] https://keras.io/optimizers/

[25] https://keras.io/metrics/

[26] https://en.wikipedia.org/wiki/Summit_(supercomputer)