Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Optimización de modelos de redes neuronales convolucionales para diagnóstico de glaucoma

Optimization of convolutional neural network models for glaucoma diagnosis

Aleiandro García Perdomo

La Laguna, 26 de mayo de 2023

D. **José Francisco Sigut Saavedra**, con N.I.F. 43.786.043-T profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Francisco José Fumero Batista**, con N.I.F. 45.731.321-F doctorando adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor externo

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

Que la presente memoria titulada:

"Optimización de modelos de redes neuronales convolucionales para diagnóstico de glaucoma"

ha sido realizada bajo su dirección por D. **Alejandro García Perdomo**,

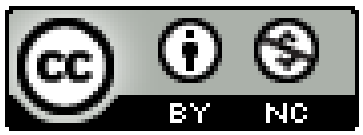
con N.I.F. 42.268.320-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 26 de mayo de 2023.

Agradecimientos

Quiero agradecer a la universidad por lo aprendido en este grado, a José y a Fran por la ayuda inmensa en este trabajo resolviendo mis dudas y dándome consejos, a mis compañeros de grado y a mi familia y amigos por apoyarme en todo momento

Licencia



Resumen

El objetivo de este trabajo ha sido buscar técnicas de optimización de redes neuronales convolucionales (CNN) y aplicarlas para la puesta en producción de modelos con menor tamaño. Al tener un menor tamaño tendrán un mejor rendimiento en dispositivos menos potentes, como puede ser un teléfono móvil, o desde el lado del cliente si hablamos de entornos web. En nuestro caso, los modelos a optimizar se utilizan para el diagnóstico del glaucoma mediante una retinografía.

La creación de modelos de redes neuronales más pequeños no solo hace que el gasto sea menor y que a la hora de la inferencia sea más rápido, también es más sencillo entrenar pequeñas redes que hagan distintas tareas que entrenar una red neuronal compleja que haga todas las pequeñas tareas al mismo tiempo.

Para ello se han utilizado dos técnicas que son la cuantificación (Quantization) y la destilación del conocimiento (Knowledge Distillation). Para aplicar estas técnicas se ha utilizado la librería Keras de TensorFlow en Python, además de otras herramientas como "tensorflowjs_converter", TensorFlowJS y TensorFlowLite para la ejecución en JavaScript. Principalmente, se han creado modelos con un tamaño menor utilizando Knowledge Distillation, llegando a reducir casi 100 veces el tamaño del modelo original.

Los modelos generados con estas técnicas de optimización son mejores tanto en velocidad en dispositivos menos potentes como en consumo en general, y además tienen la posibilidad de ser integrados en más entornos, como web, móvil y en dispositivos sin GPU. Por lo tanto, utilizar estos modelos para diagnóstico de glaucoma, u otro tipo de diagnóstico o tarea, nos permite crear herramientas más veloces que resuelven el problema, algo importante en temas médicos.

Palabras clave: Optimización, Convolucional, TensorFlow, Keras, Python, Cuantificación, Red neuronal, Knowledge Distillation, Glaucoma, Diagnóstico, Clasificación

Abstract

The goal of this work was to search for and integrate techniques on convolutional neural networks (CNN) optimization to produce smaller models. These smaller models are expected to be more efficient and faster and have better performance in devices such as mobile phones, devices without a GPU. In this case, the optimized models will be convolutional neural networks trained for Glaucoma diagnosis. It is not only for better overall performance, but also because it is faster to train smaller, less complex, models than a big one that does everything that is needed.

TensorFlow in Python and Keras, and other tools like TensorFlowLite, TensorFlowJS for JavaScript, were used in the Quantization technique and the Knowledge Distillation technique. Using these techniques, a 4MB model was obtained from a nearly 500 MB model using Knowledge Distillation.

As we build faster, smaller models, they can be integrated in smaller, less powerful devices. In a medical environment, speed is a key factor, so maybe for glaucoma diagnosis this kind of model is good for a web application, but in other cases this can be a helpful solution to speed up some big models.

Keywords: Optimization, Medical, Glaucoma, Diagnosis, TensorFlow, Keras, Quantization, Knowledge Distillation, Neural Network, Convolutional, Classification

Índice general

Capítulo 1 Introducción **1**

Capítulo 2 Estado del arte **3**

2.1 Weight Pruning 3

2.2 Cuantificación 3

2.3 Knowledge Distillation 4

Capítulo 3 Desarrollo **5**

3.1 Entornos y herramientas 5

3.2 Cuantificación 5

3.2.1 Cuantificación durante el entrenamiento 5

3.2.2 Cuantificación post-entrenamiento 5

3.2.3 JavaScript 7

3.2.4 Resultados y comparaciones 8

3.3 Knowledge Distillation 9

3.3.1 Original con Keras 10

3.3.2 Prueba Oracle Knowledge Distillation 11

3.3.3 Ejemplo de uso. Modelo de clase única. 12

3.3.4 Generación de arquitecturas de redes neuronales convolucionales 14

Capítulo 4 Conclusiones y líneas futuras **17**

Capítulo 5 Summary and Conclusions **19**

Capítulo 6 Presupuesto **22**

6.1 Tareas 22

6.2 Medios materiales 22

Capítulo 7 Código relacionado **24**

7.1 Sistema de generación de arquitecturas de redes neuronales convolucionales (Python) 24

Índice de figuras

Figura 1: Imagen de una retinografía 1
Figura 2: Gráfica de relación de acierto y alfa 11
Figura 3: Esquema de OneClass (imagen original de One-Class Convolutional Network [23]) 12
Figura 4:Matriz de confusión del modelo estudiante en la destilación de One Class 14
Figura 5: Esquema visual del sistema de generación de redes convolucionales usando Knowledge Distillation 16

Índice de tablas

Tabla 1: Resultados de cuantificación con TFLite en Python 7
Tabla 2: Resultados de tiempo de ejecución en dispositivo móvil (ms) 9
Tabla 3: Resultados de tiempo de ejecución en PC (ms) 9
Tabla 4: Tareas y sus costes 22
Tabla 5: Costes de materiales 23

Capítulo 1 Introducción

El glaucoma es una enfermedad crónica del ojo que produce ceguera parcial permanente. Se origina por un deterioro del nervio óptico y la retina, debido al aumento de la presión en el interior del ojo. Normalmente, esta presión es generada cuando se acumula el humor acuoso, el líquido que está en el interior del globo ocular, y no se drena lo suficiente a través de la apertura existente entre el iris y la córnea.

Al ser una enfermedad que se va desarrollando lentamente, la mayoría de personas no lo nota hasta que es grave. Por ello, utilizar herramientas como Deep Learning puede resultar beneficioso para el diagnóstico del glaucoma antes de que el paciente tenga daños irreparables, realizando estas pruebas en chequeos médicos, con una respuesta rápida y segura.

El equipo formado por el Grupo de Investigación en Imágenes Médicas (MIAG) de la ULL y el Servicio de Oftalmología del HUC ya tiene modelos de Deep Learning, concretamente redes neuronales convolucionales, encargados del diagnóstico del glaucoma, utilizando imágenes como la figura 1.1 para el entrenamiento de estos [1]. Estos modelos son complejos y pueden llegar a ser amplios en términos de espacio, por lo que no podrían llegar a un resultado de forma rápida en entornos web de forma directa (desde el dispositivo que se encuentra en la web) o no sería posible integrarlos en dispositivos que no tengan una GPU.

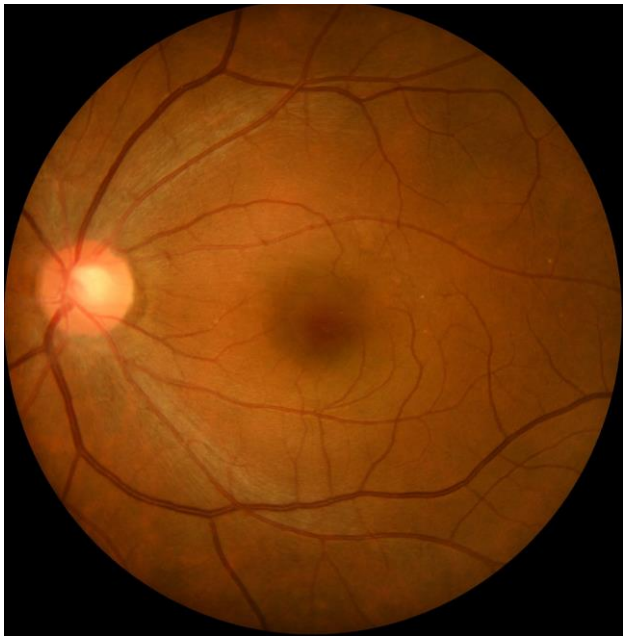


Figura 1: Imagen de una retinografía

Durante los últimos años, y gracias al auge del mundo de la inteligencia artificial y el aprendizaje profundo, empresas han ido creando entornos, bibliotecas y herramientas capaces de optimizar modelos como los mencionados. En Google se desarrolló TensorFlow [2] que permite crear modelos de Deep learning, y más tarde TensorFlow.js [3] y TensorFlow Lite [4], que permiten el uso de modelos en JavaScript y dispositivos móviles o empotrados respectivamente.

Pero incluso con estas herramientas, y tras búsqueda de estudios sobre el tema [5], donde se utilizan varias técnicas de Deep Learning, no se ha encontrado una aplicación de estas para la optimización de modelos utilizados para la detección del glaucoma.

El objetivo de este TFG es la búsqueda de técnicas de reducción de tamaño y optimización de redes neuronales convolucionales. También, utilizar estas técnicas para tener las herramientas y que se pueda poner en producción modelos de redes neuronales convolucionales más pequeños y con buenos resultados. Esta puesta en producción generará modelos que serán mucho más eficientes, tanto en consumo energético como en espacio, y realizando la inferencia de los resultados de forma más veloz.

Concretamente, estas técnicas se van a desarrollar en modelos ya creados, que se encargan del diagnóstico de glaucoma, principalmente. Estos modelos, que han sido creados por los equipos ya mencionados, se basan en arquitecturas conocidas para la clasificación de imágenes como VGG19, Xception, Inceptionv3 o resNet50. También los conjuntos de datos que se utilizan para entrenamiento y pruebas fueron proporcionados por los equipos y se utilizaron algunas públicas [1][6][7][8][9].

Por otro lado, se quiere realizar una integración de estos modelos en un entorno web para evitar la necesidad de enviar datos a un servidor para su procesamiento. Este entorno web realizará la inferencia del modelo en el propio navegador del dispositivo. La idea principal es que estos modelos que se utilicen en el entorno web no sean de diagnóstico, sino de tareas de preprocesamiento, como puede ser la comprobación de si una imagen es una retinografía o no, o la localización del nervio óptico. Precisamente, a esta tarea de determinar si se trata de una imagen de fondo de ojo o cualquier otra cosa, se ha dedicado, también, una parte del trabajo realizado que se expone en los siguientes capítulos.

Capítulo 2 Estado del arte

Actualmente existen distintas técnicas para la compresión y la optimización de modelos de redes neuronales convolucionales [10]. Algunas de las más usadas son las siguientes.

2.1 Weight Pruning

Una técnica conocida desde los años 90 es el Weight Pruning [11] o poda de pesos, que consiste en la eliminación de los pesos que menos impacto tienen en el resultado, sin tener que entrenar de nuevo el modelo, técnica que ha ido evolucionando con el tiempo [12][13].

La idea original que se menciona en [11], parte del aumento de la complejidad de los modelos utilizados para problemas reales, así como el tamaño, y era necesario reducir el tamaño de estas, sin perder resultados.

Lo primero que es necesario es un modelo entrenado, tras entrenar el modelo, se calcula el valor de los pesos o parámetros para saber cuáles se pueden eliminar y cuáles deben permanecer. Para ello existen distintas técnicas como la segunda derivada, o matriz hessiana, la "sensibilidad" de los mismos o simplemente la magnitud del peso.

Seguidamente se ordena por el valor que se ha calculado, eliminando, o manteniéndolos a 0, los pesos que no sobrepasen un valor acordado.

Seguidamente, se vuelve a entrenar el modelo, para realizar de nuevo el proceso del cálculo y eliminación, realizándose de forma cíclica hasta que se llegue al equilibrio de un buen resultado con un tamaño más pequeño.

2.2 Cuantificación

La cuantificación trata de reducir la precisión que se utiliza en los parámetros del modelo. Es decir, que, en vez de utilizar 32 bits para el almacenamiento de un único parámetro, se utilicen 16 bits o incluso 8. Esta técnica puede reducir el espacio que ocupa hasta un 75% y acelera la inferencia cuando se utiliza con enteros (necesario en algunos casos donde dispositivos menos potentes utilicen CPU).

Actualmente se utiliza tanto durante el entrenamiento como posteriormente.

En la cuantificación post-entrenamiento, simplemente se modifica el valor de los tensores del modelo con una precisión menor. El cálculo que se realiza es limitar el rango de los valores que se encuentran en el modelo entre el mínimo y el máximo valor de la parte a cuantificar.

A la hora de cuantificar un valor, se hace una transformación tal que $x \epsilon [A, B]$ pasa a un valor de n-bits donde $x_q \epsilon [A_q, B_q]$ siendo A_q y B_q el valor mayor y el valor mínimo representados con n-bits respectivamente, donde $x = A => x_q = A_q$ y $x = B => x_q = B_q$

Y la función para la cuantificación es la siguiente:

Cuantificación: $x_q = round(\frac{1}{s}x + z)$

Donde $s = \frac{B - A}{B_q - A_q}$ y "z" es el valor cero, es decir, el valor que hace que el 0 siga siendo 0 en el proceso de cuantificación, siendo $z = \frac{A\,B_q - B\,A_q}{B - A}$

Por ejemplo, teniendo el valor 0.42, en un rango de valores [-1,1], y se quiere pasar a entero de 8 bits, $x_q \epsilon [-128, 127]$. el valor cuantificado será.

$$x_q = round(\frac{0.42}{2\,/\,255} + 0.5\,) = round(54.26) = 54$$

Los valores que se salgan del rango inicial serán convertidos al valor que tengan más cercano, es decir, si superan el valor máximo, será el valor máximo y será igual con los valores menores que el mínimo.

Para realizar la cuantificación, durante el entrenamiento se emula el modelo como si estuviese cuantificado, pero en realidad es únicamente a la hora de la propagación hacia delante. La propagación hacia atrás se hace con la precisión completa de los pesos y otros atributos. Esto genera una reducción de la pérdida de acierto a la hora de cuantificar el modelo final [14].

2.3 Knowledge Distillation

La técnica que más se ha utilizado en este trabajo ha sido la técnica de destilación de conocimiento o Knowledge Distillation, que, aunque sus orígenes se remontan a 2006 [15], actualmente se está profundizando en esta técnica con distintas variaciones [16].

La idea es hacer que un modelo más pequeño, que llamaremos estudiante, se entrene teniendo en cuenta los resultados de un modelo más grande ya entrenado, que llamaremos profesor, y que tenga buenos resultados. A la hora de entrenar al estudiante para la tarea, en nuestro caso de clasificación, se utilizarán dos funciones de entropía, para saber la diferencia con el profesor, y también de la etiqueta de clasificación real.

Para la clasificación real se utilizará la función de entropía cruzada, típicamente utilizada para el entrenamiento de redes neuronales, y para la comparación con el profesor se utiliza una función de entropía relativa, o Divergencia de Kullback-Leibler, para conocer la diferencia entre la distribución de los resultados de ambos.

Cuando se tiene el resultado de ambas, se combinan en un único valor, dándole un peso a cada uno, y ese resultado es el valor que se utiliza en la propagación hacia atrás.

Esa es la idea básica, pero ha ido evolucionando y actualmente existen distintas técnicas como utilizar varios profesores, y utilizar la media de sus resultados como comparación [17], o utilizar únicamente los resultados del profesor cuando acierta [18] y utilizar los valores originales cuando no, o entrenar conjuntamente a ambos, utilizando al profesor como guía [19] mientras se sigue mejorando, reduciendo el número de épocas necesarias para ambos.

Capítulo 3 Desarrollo

En este capítulo se explicará las decisiones que se tomaron para el desarrollo.

3.1 Entornos y herramientas

Para tener un entorno online en el que poder hacer pruebas y utilizar las técnicas, se decidió utilizar Google Colab, y aunque tenga limitaciones diarias con el uso de la GPU, sigue siendo un buen entorno para ejecutar código de Python.

Se propuso utilizar una máquina externa que utilizase Jupyterhub basado en contenedores de Kubernetes, para no tener las limitaciones diarias con el uso de la GPU. Pero tras varias pruebas en local hubo problemas con el despliegue y se decidió abandonar la idea.

Para el entorno web que se utiliza en las pruebas, se despliega un servidor mediante la librería de JavaScript "express" [25], ya que se quiere una web sencilla para el testeo de los modelos.

3.2 Cuantificación

Para la cuantificación se siguió el tutorial existente en la página de TensorFlow [20] que habla de la cuantificación, tanto durante el entrenamiento como post-entrenamiento.

3.2.1 Cuantificación durante el entrenamiento

A la hora de utilizar esta herramienta, se encontraron varios problemas.

La primera opción fue directamente seguir los pasos para realizar la cuantificación según la herramienta de TensorFlow. Pero al hacer la copia del modelo para hacer la cuantificación, hubo un error porque no se puede utilizar esta herramienta con modelos de Keras que tengan submodelos dentro, como era nuestro caso. Al crear modelos utilizando Transfer Learning, los modelos que se generaban eran modelos que tenían dentro las estructuras de VGG19, por ejemplo.

La posible solución que se encontró era tener el modelo sin esa encapsulación. Para utilizar estructuras como VGG19 sin tener submodelos en Keras, hay que crearlo como la clase "Functional" [21] ya que es más flexible y genera los modelos como un grafo, permitiendo a la herramienta emular cada capa del modelo, a diferencia de la forma anterior, que no permite emular el submodelo.

A la hora de la ejecución, se generó un error que no se pudo solucionar relacionado con el tamaño del input de una de las capas. A la hora de la emulación, genera un input entre capas de tamaño 1x6 cuando debería ser de tamaño 1x4.

3.2.2 Cuantificación post-entrenamiento

La cuantificación post-entrenamiento se puede hacer solamente con los pesos del modelo, pero para que sea realmente efectivo y tenga una mejora en velocidad y consumo, también hay que cuantificar las funciones de activación, como las funciones softmax y sigmoid. Este

cambio se hace para que los cálculos se realicen todos en la misma precisión, normalmente enteros de 8 bits, para que sea más rápido, y es necesario hacerlo para dispositivos que trabajen en 8 bits o en aceleradores como Coral Edge TPU.

Para realizar esta conversión se utilizó el conversor de TensorFlow Lite (TFLite). TFLite es un conjunto de herramientas que ayuda a implementar modelos de redes neuronales en dispositivos móviles o integrados. Tiene compatibilidad con distintos lenguajes como Python, Java, C++, Swift; pero lo transformamos para utilizarlo en JavaScript con el navegador.

Aunque tenga compatibilidad con JavaScript, todavía está en una versión Alpha (se utilizó la versión 0.0.1-alpha.9), por lo que no tiene todas las funciones implementadas y puede tener algunos errores.

Al utilizar esta librería en Python se debe crear una función de datos representativos (entre 100 y 500 ejemplos), para poder fijar el rango de la entrada del modelo, ya que no se puede realizar esta calibración sin inferencia en tensores que puede ser variables como las funciones de activación, al contrario de los pesos del modelo que son fijos y sí se pueden cuantificar sin ninguna inferencia. En nuestro caso se utilizan 100 imágenes para esta calibración de los tensores que son variables.

Existen 3 conversiones:

- Conversión a Float 16. La conversión a Float 16 cambia la precisión de los pesos. No tiene ninguna característica especial, aparte de que tiene la posibilidad de ser más rápido en GPU que el original.

- Conversión a Int 8 (Entero 8 bits). La conversión a Int 8, realiza la misma conversión que la anterior, pero con una precisión de 8 bit. Para que sea una conversión completa, es necesario cambiar el tipo del tensor de entrada y el de salida a entero. Se necesita hacer la función de datos representativos previamente mencionada para su transformación completa. Además, los datos que se le pasen también deberán ser enteros.
  Se puede utilizar con CPU, Edge TPU y en microcontroladores que trabajen únicamente con enteros, en lenguajes como C++.

- Conversión en rango dinámico. Esta conversión realiza la cuantificación en los pesos del modelo como enteros. Si no se especifica, las funciones de activación se quedan como punto flotante.

En nuestro caso se ha utilizado la función de datos representativos en todas las conversiones, por lo que la cuantificación se hará también en los tensores variables, como ya se ha explicado anteriormente.

En la tabla (4.2.2) podemos ver resultados de la conversión a TFLite en Python, relacionando el peso total en Megabytes y el acierto para cada conversión, con distintos modelos.

Como se puede observar, los modelos cuantificados, teniendo la mitad de peso o incluso un 75% menos del peso original, tienen porcentajes de acierto parecidos o iguales. Además,

estos modelos se pueden utilizar en otros lenguajes como C++ o Java, utilizando librerías de TFLite ya creadas en ese lenguaje.

En nuestro caso, queremos utilizar estos modelos para realizar pruebas en JavaScript, ya que originalmente el grupo MIAG de la ULL y el Servicio de Oftalmología del HUC ya disponen de una aplicación web en la que se podrían añadir pequeños modelos con tareas de preprocesamiento.

Modelo	Original		TFLite No Cuantificado		TFLite (Float 16)		TFLite (Int 8)		TFLite (Rango Dinámico)	
	Peso (MB)	Acc	Peso (MB)	Acc	Peso (MB)	Acc	Peso (MB)	Acc	Peso (MB)	Acc
VGG19	88.65	0.959	88,65	0.959	44.34	0.959	22.32	0.952	22.32	0.952
ResNet50	138.6	0.952	138.6	0.952	69.36	0.952	35.38	0.945	35.38	0.945
Inception v3	147.1	0.952	147.1	0.952	73.16	0.952	37.27	0.952	37.27	0.952
Xception	179.3	0.945	179.3	0.945	89.7	0.945	46.11	0.932	46.11	0.939

Tabla 1: Resultados de cuantificación con TFLite en Python

3.2.3 JavaScript

Como ya se mencionó, existe una librería para TFLite en JavaScript, por lo que podemos utilizar directamente los modelos generados en Python. Tras las pruebas se ha podido comprobar que JavaScript no tiene las mejoras en TFLite que tienen otros lenguajes como C++, Swift o Java, pero se puede añadir a la ejecución.

Si no se cuantificaban los tensores variables, a la hora de crear el modelo de TFLite, los cálculos que se hacen son de precisiones distintas, por lo que previo al cálculo es necesario cambiar la precisión, lo que provoca un aumento de la latencia. Puede llegar a ser tanta la diferencia, que en las pruebas originalmente (sin hacer la cuantificación en los tensores variables) una inferencia de un modelo cuantificado tardaba 50s, y tras añadir esa conversión, el mismo modelo tarda en torno a 2s.

El problema principal es que, en JavaScript, TFLite está utilizando la CPU del dispositivo sin importar si tiene o no GPU, lo que ralentiza la inferencia, por lo que se buscó la

alternativa de utilizar el conversor de TensorFlow JS [28], que está disponible en Python para transformar los modelos a JavaScript.

No solo es una herramienta sencilla de usar, también puede cuantificar los modelos. Se probó utilizando tanto como librería de Python como comando de Bash, ya que tiene ambas opciones.

- Librería de Python. Es más sencilla de usar y no es necesario llamar al sistema operativo si se quiere hacer un script.

- Comando Bash. Tiene más opciones, como no utilizar modelos de Keras como input o cambiar el tipo de salida que se quiere. También añade la opción de que utilice un modelo solo como inferencia, creándolo como grafo, que es más rápido que un modelo creado como capas.

En nuestro caso, se quiere únicamente para inferencia y no para seguir entrenando el modelo, por lo que se utilizó el comando Bash para la creación de los modelos de JavaScript como grafos y no como capas.

TensorFlowJS utiliza OpenGL para utilizar la GPU con el navegador, y por ello utiliza las shaders de OpenGL, normalmente. También se puede especificar otros backend para el cálculo, como WASM o la CPU del dispositivo. Como la primera vez, utilizando OpenGL, que se hace una inferencia no se tiene shaders de ningún tipo, se tienen que generar y compilar, un proceso que ralentiza la inferencia. Por eso, se puede realizar una inferencia previa mientras carga la página o aplicación web, utilizando un input vacío, para hacer una carga de esos shaders necesarios para la inferencia rápida. De esta forma, las siguientes ejecuciones tendrán un transcurso normal.

Dentro del código de JavaScript solo es necesario cargar el modelo y llamar a la función "predict" para hacer la inferencia. Si fuese necesario, la librería tiene funciones para redimensionar el input que se necesite, ya sea una imagen u otro tipo de datos.

3.2.4 Resultados y comparaciones

Aunque ya se mencionó que TFLite no está optimizado para JavaScript, se ha hecho la comparativa entre este y el conversor de TensorFlowJS, comparando la velocidad, ya que, en términos de peso, la cuantificación es del mismo tipo.

El modelo de cuantificación de TFLite con enteros de 8 bits, no se pudo ejecutar ya que espera una entrada de este tipo, y en TensorFlow JS no está implementado por el momento tensores con ese tipo como elemento.

La ejecución en ordenador ha sido hecha en Firefox y la ejecución en móvil en Google Chrome, pero el navegador no afecta normalmente.

Modelos	Original	Float 16	Int 8	TFLite	TFLite Float 16	TFLite Rango Dinámico
VGG19	46	41	50	2791	2739	3476
ResNet50	70	62	60	638	573	730
Inceptionv3	147	97	75	883	825	1046
Xception	101	63	60	1302	1233	1454

Tabla 2: Resultados de tiempo de ejecución en dispositivo móvil (ms).

Modelos	Original	Float 16	Int 8	TFLite	TFLite Float 16	TFLite Rango Dinámico
VGG19	6	5	6	1724	1683	956
ResNet50	12	10	10	346	343	230
Inceptionv3	15	14	12	480	473	317
Xception	8	8	7	689	687	492

Tabla 3: Resultados de tiempo de ejecución en PC (ms)

En las tablas 3.2.4 - 1 y 3.2.4 - 2, se pueden observar en las tres primeras columnas los resultados de la ejecución de los modelos creados mediante el conversor de TensorFlow JS, y en las tres últimas columnas los resultados de modelos creados mediante TFLite. Los modelos cuantificados que no están creados mediante TFLite son más rápidos, ya que como se ha comentado, TFLite no está optimizado para JavaScript. También, que en la mayoría de casos tanto en móvil como PC, estos modelos son más veloces de media en su forma cuantificada, y cuanto mayor sea el modelo y más se tarde en hacer la inferencia (Inceptionv3, Xception), mayor es la diferencia con sus respectivas partes cuantificadas.

Curiosamente la ejecución de los modelos de TFLite de rango dinámico en PC es más veloz que los otros modelos de TFLite, aunque en móvil ocurre al contrario. Podría deberse a las diferencias de arquitectura (ARM vs x64) y su distinta gestión de precisión.

3.3 Knowledge Distillation

Para utilizar esta técnica, se ha utilizado el ejemplo con la clase “Distillation” del tutorial de Keras sobre Knowledge Distillation [22].

3.3.1 Original con Keras

Como se acaba de mencionar, utilizamos la clase "Distillation" que modifica los pasos de entrenamiento y test de los modelos de Keras para realizar el procedimiento ya explicado en el apartado 2.3.

Para el modelo profesor se utilizó como prueba uno de los modelos generados por el grupo MIAG, uno basado en la arquitectura VGG19. Para el estudiante se probaron varios modelos pequeños, que constaban de entre 2 y 5 capas convolucionales y una capa totalmente conectada previa a la salida.

Hay que tener en cuenta también otros hiperparámetros, aparte de los que ya conocemos, como el número de épocas de entrenamiento, la tasa de aprendizaje o el número de neuronas no visibles. El peso a repartir entre la pérdida con el profesor y la realidad, y la temperatura usada en la Divergencia de Kullback-Leibler para la suavización de la distribución, son también importantes para el resultado del entrenamiento.

Por ello, los resultados que se van a comparar vienen de un mismo profesor, el modelo con la arquitectura basada en VGG19 que se mencionó previamente, con un acierto de 91,89%; con un mismo estudiante, una red neuronal con dos capas convolucionales, de 32 y 64 filtros con un kernel de 3x3, y una capa totalmente conectada de 8 neuronas. La parte variable a comparar van a ser los hiperparámetros del peso a repartir, siendo α el peso a repartir para la pérdida de la parte real y 1 - α el peso que se le da a la pérdida con el profesor, y también la temperatura.

El modelo estudiante originalmente, sin utilizar la destilación, consiguió un acierto de 0.8896 tras un entrenamiento de 10 épocas. El modelo estudiante con destilación y mismas épocas obtuvo los siguientes resultados.

En la figura 3.3.1, se observa una gráfica donde se muestra el porcentaje de acierto dependiente de alfa, el valor que marcaba los pesos entre la realidad y el profesor. También cada línea representa una temperatura distinta para la suavización de la distribución entre profesor y estudiante. Los valores más pequeños de alfa generan los resultados mayores, incluso llegando a tener la precisión del profesor. En cambio, la temperatura no afecta en gran medida, exceptuando casos que se salen de lo común, posiblemente por la aleatoriedad de los entrenamientos.

Los resultados muestran que siendo menor el alfa se obtienen mejores resultados, lo que confirma, que al tener el profesor como "guía"; y teniendo éste mucha influencia sobre los gradientes del entrenamiento; genera buenos resultados.

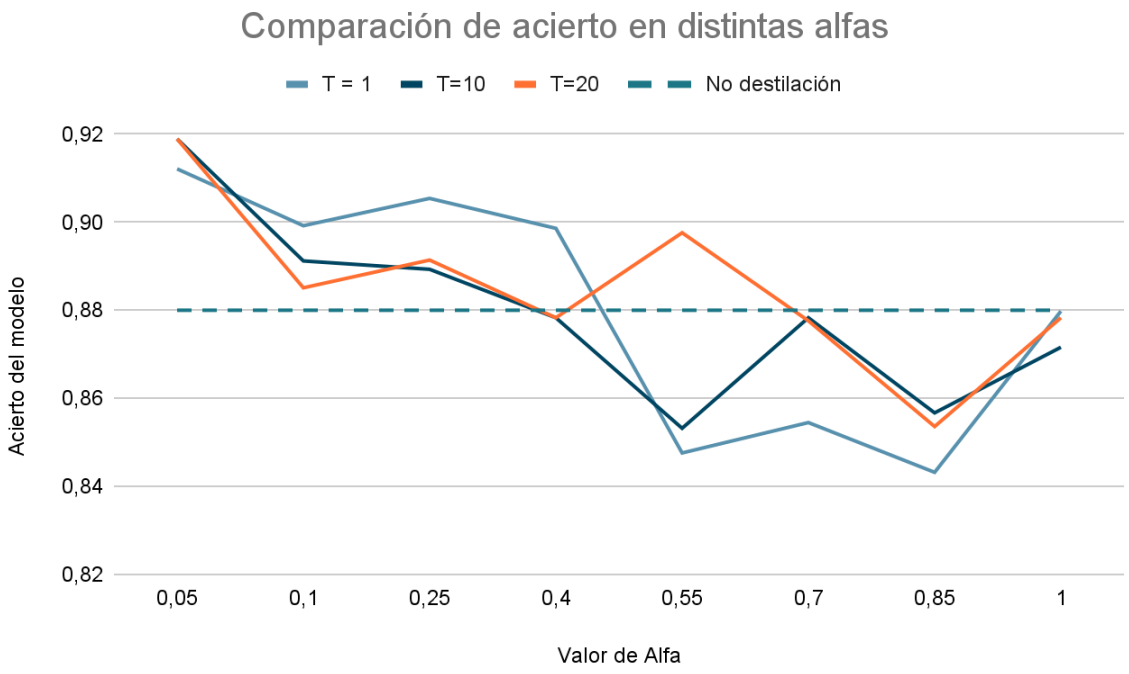


Figura 2: Gráfica de relación de acierto y alfa

3.3.2 Prueba Oracle Knowledge Distillation

En Oracle Knowledge Distillation [18] se propone una mejoría en el entrenamiento del estudiante. Se propone eliminar las predicciones erróneas por parte de los profesores e introducir la realidad.

Resumiendo un poco la explicación, a la hora de recibir las predicciones de parte de los profesores, se compara con la realidad y solo se hace la media de esas predicciones con las que son correctas. Es decir, si en un clasificador de imágenes pasamos una entrada y solo 2 de 3 profesores clasifica bien esa imagen, se utilizará la media de los dos profesores que han hecho correctamente la clasificación para la parte de comparación con el estudiante. Si ningún profesor acierta, se utiliza una función entropía cruzada con la realidad, en vez de la divergencia de Kullback-Leibler.

Por qué utilizar esta modificación, el limitante que tiene la versión original, es que el modelo estudiante, probablemente no supere nunca al profesor, ya que estamos haciendo la tarea de igualar al profesor. Con la modificación es posible tener una mejoría sobre el profesor, con un modelo más pequeño.

Se intentó realizar la modificación, pero no se completó la integración. El principal problema fue un problema de memoria, que se generaba al actualizar los gradientes del modelo estudiante, algo que no se pudo solucionar con la modificación que se quería. Se buscó una alternativa que era sustituir de las predicciones originales de los profesores, las erróneas por la verdad y utilizar todas las entradas en la función de divergencia de Kullback-Leibler, pero los resultados se mantenían en un 50% en las primeras épocas y no llegaba a haber una mejora, ni un aprendizaje por parte del modelo estudiante.

Tras no encontrar el error, se abandonó el uso de estas técnicas para continuar con la integración del modelo original, o del modelo con varios profesores, incluyendo los errores en la media.

3.3.3 Ejemplo de uso. Modelo de clase única.

Uno de los objetivos que se tenía era crear un modelo pequeño para la clasificación de imágenes, que comprobase si una imagen es una retinografía o no. Para un problema así, se puede utilizar una arquitectura que solo tiene una clase como salida, siendo que la imagen a clasificar pertenece a la categoría, en nuestro caso una retinografía, o no.

Esta arquitectura se conoce como One-Class [23], y parte de la necesidad de entrenar una red neuronal teniendo únicamente datos de la clase que queremos, ya que lo que no pertenece a esta clase, es el resto de casos, y si se crease esa información de objetos no pertenecientes a la clase objetivo, es posible crear sesgos y empeora el acierto.

La arquitectura solo modifica el entrenamiento del modelo. Primero se necesita, si se quiere utilizar para clasificación de imágenes, un buen extractor de características de una imagen. Este extractor de características puede ser arquitecturas que ya hemos mencionado, como VGG19 o Inception.

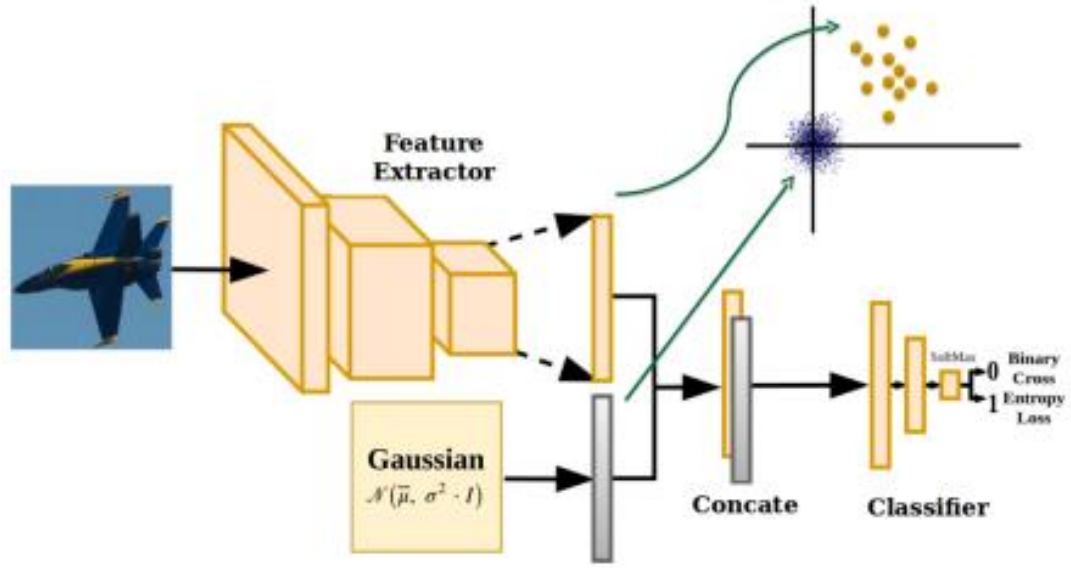


Figura 3: Esquema de OneClass (imagen original de One-Class Convolutional Network [23])

También necesitamos un generador de ruido Gaussiano que creará unas características falsas que se tomaran como la NO clase. Finalmente se concatenan la salida del extractor de características y las características de ruido, y se utilizan en un clasificador.

Como se puede observar en la figura 3.3.3 - 1, las características gaussianas estarán fuera del conjunto de características "buenas", lo que forma la separación entre la clasificación de la clase real (características parecidas a las buenas) y el resto (más parecidas a las gaussianas).

En nuestro caso, el grupo del MIAG, ya tenían un modelo generado, que utilizaba VGG19, y junto con esta técnica de One-Class, tenía resultados cercanos al 100% de acierto en sus test. El problema de este modelo es que ocupaba en torno a los 500 MB de peso, por lo que sería imposible desplegarlo directamente en web o en dispositivos móviles.

Para reducir este tamaño se intentó primeramente el uso de cuantificación, pero el modelo era tan grande que la RAM de Google Colab se quedaba sin memoria para hacer la conversión. Por eso se optó por crear un modelo muy pequeño que aprendiese del modelo grande, que seguidamente se pudo cuantificar para reducir aún más su peso.

Este pequeño modelo, consistía únicamente de 3 capas convolucionales de 64 y 128 filtros, y la capa softmax de clasificación. En términos de espacio, ocupa 4MB. Cuando aplicamos como prueba, a este modelo, la técnica de One-Class para comparar, el resultado de acierto no se acercaba al 65%.

Para el entrenamiento de Knowledge Distillation, se utilizaron las imágenes de test que se utilizaron en One-Class, un conjunto de imágenes de retinografías e imágenes aleatorias que no estaban conectadas. Y para el testing del modelo después de la clasificación, se creó un conjunto de 9250 imágenes, de las cuales 2351 son de retinografías y 6899 no son retinografías. Estas imágenes provienen de las bases de datos públicas de retinas que hemos mencionado al comienzo y de la base de datos de imágenes naturales de Kaggle [24], que es utilizada para este tipo de casos.

Tras el entrenamiento de únicamente 10 épocas, el modelo estudiante consiguió un acierto de un 94.87% en las 9000 imágenes.

En la figura 3.3.3 - 2, podemos observar la matriz de confusión de este test, concretamente se puede observar que el mayor error viene de clasificar algunas imágenes de retinografía como no retinografías. La mayoría de estos errores vienen de que las imágenes utilizadas en el entrenamiento eran todas de retinografías completas, y al no tener suficientes de este tipo para la parte de testeo, se incluyeron muchas imágenes de retinografía recortadas en el disco óptico de la retinografía. Se pensó como una comprobación de la eficacia de este método, ya que los recortes en el disco óptico en retinografías, mejoran la clasificación del diagnóstico del glaucoma, como se comenta en [5].

La idea principal con este modelo pequeño, es utilizarlo como una guía en la página web, para filtrar las imágenes que no son retinografías, no impediría que el usuario envíe una imagen errónea a la clasificación, pero sí que se avisa en caso de error, al usuario por una confirmación de que la imagen es o no una retinografía, por si el usuario ha enviado la imagen que no es.

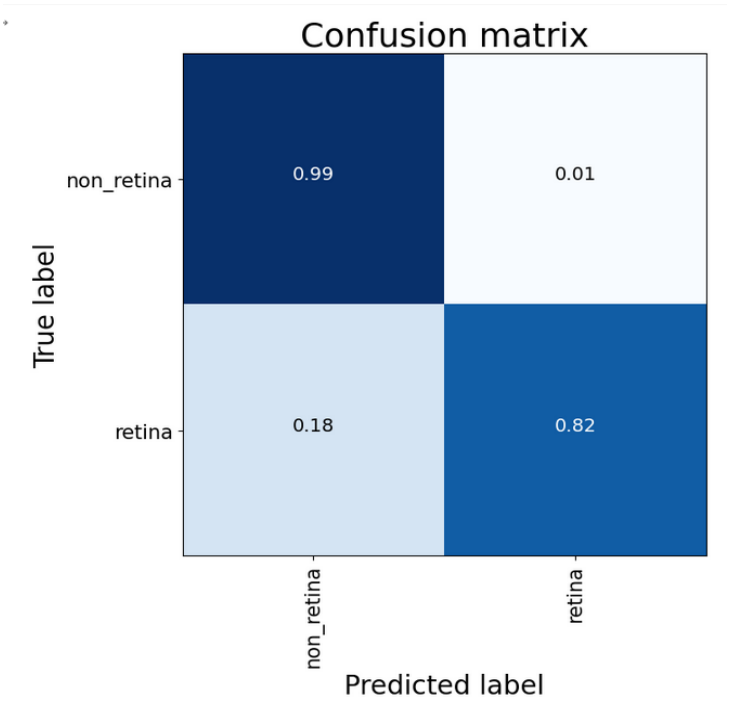


Figura 4:Matriz de confusión del modelo estudiante en la destilación de One Class

3.3.4 Generación de arquitecturas de redes neuronales convolucionales

La idea viene de la posibilidad de que el modelo pequeño puede ser o no el mejor para un profesor y para otro. Por eso se decidió crear el sistema de generación. Además, la búsqueda de arquitecturas es una técnica que se usa actualmente para distintos problemas [26][27].

Al comenzar esta parte, se quiso utilizar AutoKeras, una librería que trabaja sobre Keras y crea modelos adaptándolos al problema actual, mejorando el acierto del modelo que genera poco a poco. Al realizar las primeras pruebas, vimos que era bastante lento a la hora de generar un modelo y consumía mucha GPU, algo que estaba limitado por Google Colab, por lo que se cambió de estrategia.

La estrategia por la que se optó fue seguir la idea de un algoritmo genético. La idea es ir creando por bloques las redes neuronales, dentro de un espacio de búsqueda. Inicialmente se quería modificar la arquitectura del modelo tanto en su parte de extracción de características (parte de capas convolucionales) como en la parte de clasificación (parte de capas de neuronas totalmente conectadas). Pero tras observar que el espacio de búsqueda era muy grande, y que la complejidad de los posibles modelos iba a ser muy grande, se decidió únicamente modificar la parte de extracción de características, que es la más importante.

Algunos problemas que se encontraron al modificar los modelos anteriores, es la necesidad de crear nuevas capas iguales de clasificación, ya que las del modelo anterior tenían un input distinto al nuevo. Esto generaba la necesidad de darle un nombre a cada capa, ya que Keras

al crear capas les da un nombre para diferenciar, pero este nombre solo se crea para el ámbito, por lo que varias veces se generaba un error de dos capas que se llaman iguales en el mismo modelo, algo que Keras no permite.

Los bloques utilizados para la modificación de los modelos son bloques de 1 capa convolucional, una posible capa de normalización del batch y una capa de pooling. Dentro de este bloque se incluirá un número de filtros aleatorio, entre 16 y 512 filtros, siempre siendo potencias de 2. Al realizar la modificación, se introducirá este bloque como última capa convolucional de la parte de extracción de características, para simplificar más la búsqueda.

Originalmente se tendrán 5 modelos iguales, que ya han sido entrenados, formados por una capa convolucional de 16 filtros y una capa densa (totalmente conectada) de 16 neuronas. De estos 5 modelos, se escogen 3 aleatoriamente y se les añade un bloque aleatorio. Se entrena a los modelos modificados, utilizando Knowledge Distillation, y se almacenan el acierto tras el testing y su peso total en MB. Cuando se acabe el entrenamiento, estos tres modelos se añaden con los 5 anteriores, y se escogen los 5 con mejor acierto. Se repite el proceso con los mejores modelos. Véase la figura 3.3.4.

Es un proceso sencillo, pero que puede resultar eficaz.

Se han utilizado cinco profesores, donde todos son modelos basados en VGG19 con la tarea de clasificación de glaucoma y con un tamaño de 88.69 MB, y únicamente 5 épocas para el entrenamiento de los modelos pequeños. Con esta sencilla estructura, se dejó ejecutar hasta conseguir un porcentaje de acierto mayor que el 90%. Se eligió este resultado, porque primeramente puede considerarse buen resultado en un modelo pequeño y no se utiliza el peso, porque varía mucho y al añadir una capa convolucional nueva, es posible reducir el tamaño del modelo, ya que las conexiones con las capas de clasificación, la capa densa, puede verse reducida, lo que disminuye el peso.

Tras media hora, se paró manualmente el proceso para observar resultados, y se generaron 2 modelos que superan el 85% de acierto y otros 3 que superan el 75%. Aunque no se hayan conseguido resultados por encima del 90%, únicamente con 5 épocas de entrenamiento se consiguieron modelos que ocupaban entre 3 y 15 MB que llegaban a más de un 75% de acierto.

Cabe destacar que los modelos generados tenían una estructura muy parecida a las que se utilizan comúnmente en las redes convolucionales, al principio un menor número de filtros y a medida que aumenta el número de capas, aumenta el número de filtros.

El sistema puede mejorarse, utilizando, por ejemplo, técnicas de Fine Tuning como entrenar de nuevo con una tasa de aprendizaje menor y congelando las primeras capas convolucionales, para que no se vean modificadas y el porcentaje de acierto pueda mejorar o se podría utilizar una función heurística para que los bloques que se introducen al modificar a los estudiantes sean más parecidos a los modelos que se usan actualmente, aumentando el número de filtros a medida que se aumenta el número de capas convolucionales, por ejemplo. Aumentar el número de épocas es una opción que mejorará notablemente los resultados, pero también ralentiza la ejecución.

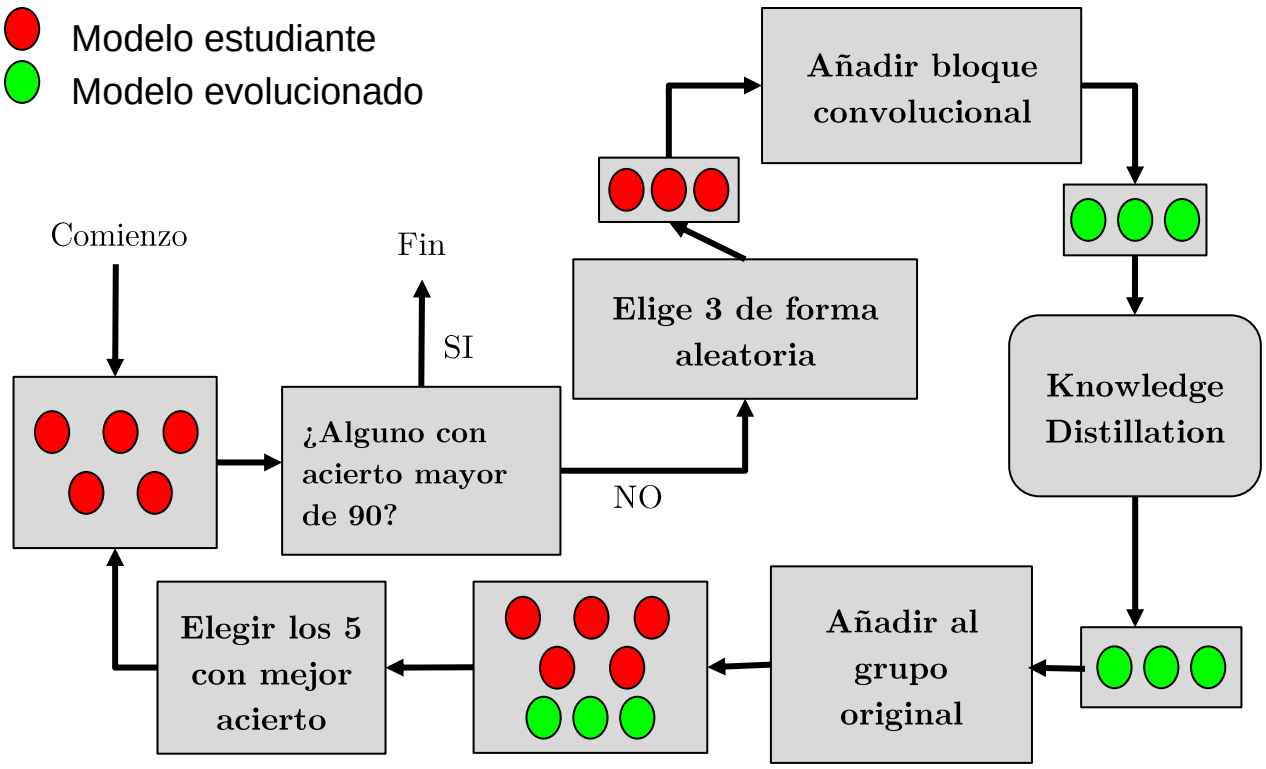


Figura 5: Esquema visual del sistema de generación de redes convolucionales usando Knowledge Distillation

Capítulo 4 Conclusiones y líneas futuras

Las técnicas desarrolladas en este trabajo son sencillas de aplicar, gracias a la cantidad de herramientas y documentación que hay online. Teniendo estas herramientas disponibles, modelos complejos de grandes tamaños pueden integrarse como modelos más pequeños, véase el caso expuesto de clasificación de si es una retinografía o no. Un modelo que sería imposible de integrar en web por su tamaño, pasa a ocupar 4MB, sin llegar a cuantificar después ese modelo resultante.

El modelo se ha puesto a prueba en una aplicación web sencilla y filtra correctamente la mayoría de fotos que no son una retinografía, lo que permite seguidamente mandar esa misma imagen para su diagnóstico si es una retinografía. También podría mejorar la experiencia de usuario, ya que se hace una comprobación previa, descartando imágenes erróneas antes de llegar al análisis, evitando el consumo.

Esta pequeña prueba demuestra que los modelos pequeños pueden utilizarse en ámbitos reales. Es cierto que TFLite en JavaScript no funciona, pero todavía está en desarrollo, ya que es reciente. Sería interesante comprobar el funcionamiento en otros lenguajes, como puede ser C++, y su funcionamiento en componentes de un Arduino, por ejemplo.

También hay que tener en cuenta el trabajo necesario para utilizar Knowledge Distillation, ya que entrenas otra red neuronal, y si quieres resultados más refinados, vas a necesitar un mayor número de épocas entrenando con el profesor, aunque normalmente no es necesario llegar al mismo número de épocas que el entrenamiento del profesor.

Como comparativa entre ambas técnicas, se puede observar que la pérdida de precisión en la cuantificación es minúscula, pero la mayor reducción de tamaño conseguida es de un 75%, mientras que el límite de reducción con Knowledge Distillation es mayor (en el ejemplo de uso se disminuyó el tamaño a 4 MB, una reducción del 99.2 % del total) pero también con una pérdida mucho mayor, el mejor tenía una pérdida del 5%. Podemos concluir que para modelos muy grandes puede ser mejor usar Knowledge Distillation y para modelos más pequeños, se puede utilizar cuantificación.

Como líneas futuras, lo que sería más interesante es conseguir integrar modelos creados con estas técnicas en la aplicación web del grupo MIAG y el HUC, y realizar las pequeñas tareas que ayudan al correcto diagnóstico del glaucoma.

Por otros temas menos relacionados, comprobar el resultado de estas técnicas en otro tipo de tareas como puede ser procesamiento del lenguaje natural o clasificación sin utilizar redes convolucionales, ver en cambios en el comportamiento de la red original y las derivadas. Por otro lado, sería interesante mejorar el sistema de generación de redes convolucionales que se ha expuesto, ya que es uno muy simple, que no termina de utilizar algoritmos de evolución o genéticos, que era la idea original detrás de ese sistema, aparte de perfilar los resultados.

En conclusión, el uso de estas herramientas puede generar un trabajo extra, pero merece la pena por la facilidad que dan para desplegar estos modelos más pequeños, que consumen menos, algo muy importante en la actualidad, tienen una mayor velocidad en muchos casos y dan más versatilidad a la hora de utilizar dispositivos con menos recursos computacionales. En entornos médicos la velocidad es crucial, posiblemente para nuestro caso no, pero sí que mejora la experiencia de usuario si se usa en una página web, pero en general es necesario mejorar la velocidad de los modelos si queremos utilizar inteligencia artificial en la medicina.

Capítulo 5 Summary and Conclusions

Glaucoma is an eye disease that causes permanent spot blindness. It is normally caused by an increase in the pressure inside the eye, damaging the optical nerve. It is hard to notice the effects at the start, and once that is noticeable, it cannot be reversed. This is why it is important to have efficient Deep Learning models that can correctly diagnose this disease.

The team formed by the Medical Image Analysis Group (MIAG) from the ULL University and Hospital Universitario de Canarias (HUC) have already developed a web app that integrates Deep Learning models for Glaucoma diagnosis.[1] These models occupy a lot of space and cannot be used directly on the web app, from the device, or in devices that have less computing power or directly have no GPU.

Recently, thanks to the recent ground-breaking discoveries in Artificial Intelligence and Deep Learning and increasing interest in them, multiple companies, like Google, have made tools and libraries used for creating and optimizing Deep Learning and Machine Learning models. TensorFlow [2] is one of them, with TensorFlowJS [3] and TensorFlowLite [4].

Even though these tools are available online, there are no optimization techniques being used on Deep Learning for Glaucoma diagnosis. The goal of this study is to use a few of them and see the results to integrate them in a real environment, for example, a web application.

All the models that have been already trained and the databases of the image of the retina, were given by the MIAG for its use. Also, all the techniques are used on Python, using Keras.

The first technique used was Quantization, using TensorFlowLite. The technique is to reduce the precision of the types inside the model. For example, changing from float 32-bits to integer 8-bits. This can reduce the model size up to 25% of the total size, while losing low to no accuracy (see table from chapter 3.2.2) and increasing speed on GPU and CPU. For using certain devices, it is needed to do a full quantization to integer 8, even the input, output, activation function and biases.

We also wanted these models to work on JavaScript, and there exists a library for TensorFlowLite in JavaScript (it is still in alpha) t, so models made in Python can be passed to JavaScript through TFlite.

Even though there are two main uses, only one reached for results. Quantization aware training is a technique used to emulate quantization while training, but only on forward propagation, backpropagation is with full precision. When the model finishes training will be quantized and will not lose accuracy. This did not work for us, as there was a mismatch between two layers shapes while training, but not before, so the training could not be finished.

Post-Training quantization worked well. It was important to not only make quantization of the weights, but also from the activation functions, so there were no type conversions on operations that could slow down inference.

This is not the only way to transform from Python to JavaScript, while making quantization of the model. There is another library, tensorflowJS_converter [28], that can be used. This library converts the models to TensorFlowJS.

Looking at the tables at chapter 3.2.4 we can see that TFLite models do not work at the same speed as the models made with TensorFlowJS_converter, so at least in JavaScript it is better to use the second tool, as TFLite is still on alpha.

The other technique that we used is Knowledge Distillation [17]. It is based on the training of a smaller model, known as the student, while using a bigger model to "guide" training, known as the teacher, which can be more than one. The idea is to get the smaller model to be like the bigger ones, instead of using the ground truth only.

Using Keras and its example on Knowledge Distillation [22], a few smaller models were created using KD and compared to the same models, most reached similar accuracy to the teacher model.

We tried to use a variation on this technique, called Oracle Knowledge Distillation, which uses the teachers' predictions only when they are right, and when none is right, uses the ground truth classification. But it could not be completed, as Google Colab sometimes ran out of memory, and when not, the accuracy while training was not rising, and the smaller model was not learning.

In a practical case, we used Knowledge Distillation to use a 500MB model (trained as a One-Class model [23] to differentiate between retina images and anything else) to train a 4MB model, having a 94% of accuracy at the end of it. This model could also be used on a web application, as the bigger one was not possible.

This small model was made by hand, no using other architectures like VGG. But we are not sure if this is the best model for this case, so a system was made to make smaller models via "evolution". Basically, from a plain model with a single convolutional layer, we make 5 copies, and then pick 3 from these and randomly add a convolutional layer between the last convolutional layer and the classification part of the model (fully connected layer).

After this "evolution" the models are trained using Knowledge Distillation with a Glaucoma diagnosis model as the teacher, and tested. Then these are added to the previous five models, ordered by accuracy, and only the five bests are picked to do this process again.

After 30 minutes, with only 5 epochs in training there were 5 small models with an accuracy greater than 75%, 2 of them greater than 85%. Even though these results are not at the same level as the original, genetic, and evolutionary algorithms are used on architecture searching [26][27].

Optimizing big models is more work most times, and usually needs more time, to train the perfect small model, using Knowledge Distillation with more epochs. But as seen, these smaller models are not only faster and efficient, but can be integrated on mobile devices and

even in devices with less computing power or no GPU at all.

In conclusion, these techniques are easy to use, most of them, if not all, are free to use and have a lot of documentation online. Smaller models have a lot of advantages and are needed to reduce the energy consumption that bigger models have. Also, faster models mean faster diagnosis in our case, maybe not crucial for something like an eye test, but could be used on other instances.

Capítulo 6 Presupuesto

6.1 Tareas

Tareas necesarias para el proyecto.

Tipos	**Descripción**	**Horas**	**Coste por Hora**	**Coste total**
Tareas de investigación	Investigación de técnicas actuales de optimización de modelos y cómo utilizarlas	24 h	20€	480 €
Desarrollo Cuantificación	Desarrollo y testeo del código necesario (Python)	40 h	20€	800 €
Desarrollo Knowledge Distillation	Desarrollo y testeo del código necesario (Python)	112 h	20€	2240 €
Desarrollo Sistema de generación	Desarrollo y testeo del sistema de generación utilizando Knowledge Distillation	80 h	20€	1600 €
Mejora del sistema de generación	Mejora del sistema ya creado, utilizando Fine tuning y otras técnicas.	32 h	20€	640 €
Total tareas		288 h	-	5760 €

Tabla 4: Tareas y sus costes

6.2 Medios materiales

Medios materiales utilizados, alguno es opcional, podría acelerar el flujo de trabajo. También, los modelos y las imágenes utilizadas para la optimización vienen dados por el cliente, si no se proporcionan habría que utilizar repositorios públicos de imágenes y también modelos, o agregar un coste por ello.

Tipos	**Descripción**	**Coste total**
Google Colab Pro	Suscripción a Google Colab Pro, durante 4 meses, para mejorar la velocidad de GPU (OPCIONAL)	40 - 240€
Ordenador Portátil	Ordenador en el que se desarrolla de gama media/alta	900 €
Total tareas		900 - 1140€

Tabla 5: Costes de materiales

Capítulo 7 Código relacionado

7.1 Sistema de generación de arquitecturas de redes neuronales convolucionales (Python)

El siguiente código no está completo, falta la carga de los modelos originales y de los profesores, solamente está el proceso de la generación de los modelos finales. Se utiliza TensorFlow y Keras.

```
 import random
import tensorflow as tf
from tensorflow import keras
import os
import numpy as np


input_size = 224
filter_size = [ 512, 256, 128, 64, 32, 16]
# Código del nombre, para marcar el nombre las capas
name_n = 1
# students es la variable donde se guardan los modelos que se van
a “evolucionar”
while (all([mod.acc < 90 for mod in students])):
  evol_models = random.sample(students,3)
      # Para cada modelo escogido
  for model in evol_models:
     layers = [l for l in model.st_model.layers] # Se sacan las
capas del modelo actual
     layer_id = len(layers) - 5 # La última capa del bloque
convolucional es la 5 desde el final
        # Se puede generalizar buscando la capa Flatten, y la
anterior sería la última
     x = layers[layer_id].output
    # Se escoge aleatoriamente si agregar o no una capa de
```

```
Normalizacion
     number = random.randrange(0,2)
     if (number == 0):
         x = NConvPoolBlock(x,3, random.choice(filter_size),
name="block_" + str(name_n))
     else:
           x = NConvBatchPoolBlock(x,3,
random.choice(filter_size), name="block_" + str(name_n))
   # Modificando el número del bloque
     name_n += 1


    # Se regeneran las capas siguientes al bloque, ya que el
output del nuevo bloque tiene una forma distinta
     for i in range(layer_id + 1,len(layers)):
       if layers[i].__class__.__name__ in FUNCTIONS:
         x = GetResults(x, layers[i], name="block_" + str(name_n))
       else:
         x = layers[i](x)
   # Se crea el modelo
     new_model = tf.keras.Model(inputs=layers[0].input,
outputs=x)
   # Se utilizan los profesores, como estudiante el modelo que se
acaba de crear. Esta clase es la que se encuentra en el tutorial
de Keras sobre Knowledge DIstillation
     distiller = Distiller(student=new_model, teachers=teachers)
     distiller.compile(
        optimizer=tf.keras.optimizers.Adam(),
        metrics=[tf.keras.metrics.CategoricalAccuracy()],
student_loss_fn=tf.keras.losses.CategoricalCrossentropy(from_logit
s=False),
        distillation_loss_fn=tf.keras.losses.KLDivergence(),
        alpha=0.5,
        temperature=10,
    )
```

```
      distiller.fit(train_dataset, epochs=5)
      [acc, loss] = distiller.evaluate(test_dataset)
      weight = GetMBSize(distiller.student)

    # Se añade al nuevo modelo a los estudiantes
students.append( StudentData(distiller.student,acc,loss,weight))
   # Se elimina el destilador para liberar memoria
      del distiller

 # Tras pasar por todos los modelos, se reordena a los
estudiantes, y se toman 5
  students.sort(key=lambda x : x.acc, reverse=True)
  students = students[:5]
```

Bibliografía

[1] - F. Fumero, T. Diaz-Aleman, J. Sigut, S. Alayon, R. D. Arnay, and Angel-Pereira (2020). "RIM-ONE DL: A Unified Retinal Image Database for Assessing Glaucoma Using Deep Learning." Image Analysis & Stereology, 39(3), 161-167. doi:https://doi.org/10.5566/ias.2346

[2] - M. Abadi et al. "TensorFlow: Large-scale machine learning on heterogeneous systems" Accessed: 26/05/2023. [Online]. Available: https://www.tensorflow.org/, doi: 0.5281/zenodo.4724125.

[3] - TensorFlow Developers." TensorFlow JS" TensorFlow https://www.tensorflow.org/js (accessed May. 26, 2023)

[4] - TensorFlow Developers. "TensorFlow Lite" TensorFlow https://www.tensorflow.org/lite (accessed May. 26, 2023)

[5] - J. Camara, A. Neto, I. M. Pires, M. V. Villasana, E. Zdravevski, and A. Cunha, "Literature Review on Artificial Intelligence Methods for Glaucoma Screening, Segmentation, and Classification," *Journal of Imaging*, vol. 8, no. 2, p. 19, Jan. 2022, doi: 10.3390/jimaging8020019.

[6] - A. Diaz-Pinto, S. Morales, V. Naranjo, T. Köhler, J. M. Mossiand, A. Navea, "CNNs for Automatic Glaucoma Assessment using Fundus Images: An Extensive Validation". (15-Mar-2019). figshare, [Online]. Available: https://figshare.com/articles/dataset/CNNs_for_Automatic_Glaucoma_Assessment_using_Fundus_Images_An_Extensive_Validation/7613135/1. doi: https://doi.org/10.6084/m9.figshare.7613135.v1.

[7] - J. I. Orlando et al. "REFUGE Challenge: A unified framework for evaluating automated methods for glaucoma assessment from fundus photographs" (2020). Medical Image Analysis, Volume 59, Accessed: 26/05/2023. [Online] Available: https://www.sciencedirect.com/science/article/pii/S1361841519301100, doi: https://doi.org/10.1016/j.media.2019.101570.

[8] - Z. Zhang et al. "ORIGA(-light): an online retinal fundus image database for glaucoma analysis and research" (2010) Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual International Conference, 3065–3068. doi: https://doi.org/10.1109.

[9] - M. S. Haleem, L. Han, B. Li, A. Nisbet, J. Van Hemert, M. Verhoek, "Automatic Extraction of the Optic Disc Boundary for Detecting Retinal Diseases." (2013) Proceedings

of the IASTED International Conference on Computer Graphics and Imaging, CGIM 2013. doi: 10.2316/P.2013.797-015.

[10] - J. T. O'Neill, "An overview of Neural Network Compression" (2020) Accessed: 26/05/2023. [Online]. doi:https://doi.org/10.48550/arXiv.2006.03669.

[11] - Y. L. Cun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. San Francisco, CA: Morgan Kaufmann, 1990, pp. 598--605.

[12] - D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the State of neural network pruning?", arXiv.org, (2019) Accessed:26/05/2023 [Online]. Available: https://arxiv.org/abs/2003.03033. doi: https://doi.org/10.48550/arXiv.2003.03033.

[13] - H. Tanaka, D. Kunin, D. L. K. Yamins, S. Ganguli "Pruning neural networks without any data by iteratively conserving synaptic flow", arXiv.org, (2020) Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/2006.05467. doi: https://doi.org/10.48550/arXiv.2006.05467.

[14] - B. Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", arXiv.org, (2017) Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/1712.05877. doi: https://doi.org/10.48550/arXiv.1712.05877.

[15] - C. Bucilua, R. Caruana, and A. Niculescu-Mizil, 'Model Compression', in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Philadelphia, PA, USA, 2006, pp. 535–541. doi: https://doi.org/10.1145/1150402.1150464

[16] - F. Ruffy and K. Chahal, "The State of Knowledge Distillation for Classification", *arXiv*. (2019). Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/1912.10850. doi: https://doi.org/10.48550/arXiv.1912.10850.

[17] - G. Hinton, O. Vinyals and J. Dean, "Distilling the Knowledge in a Neural Network", *arXiv*. (2015). Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/1503.02531. doi: https://doi.org/10.48550/arXiv.1503.02531.

[18] - M. Kang, J. Mun and B. Han, "Towards Oracle Knowledge Distillation with Neural Architecture Search", *arXiv*. (2019). Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/1911.13019. doi: https://doi.org/10.48550/arXiv.1911.13019.

[19] - K. Zhang, C. Zhang, S. Li, D. Zeng and S. Ge, "Student Network Learning via Evolutionary Knowledge Distillation" *arXiv*. (2021). Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/2103.13811. doi:

https://doi.org/10.48550/arXiv.2103.13811.

[20] - TensorFlow Developers.” TensorFlow: Model Optimization”. TensorFlow, https://www.tensorflow.org/lite/performance/model_optimization#quantization (accessed May. 26, 2023)

[21] - F. Chollet, “The Functional API” Keras, https://keras.io/guides/functional_api/ (accessed May. 26, 2023)

[22] - K. Borup, “Implementation of Classical Knowledge Distillation” Keras, https://keras.io/examples/vision/knowledge_distillation/ (accessed May. 26, 2023)

[23] - P. Oza and V. M. Patel “One-Class Convolutional Neural Network” *arXiv*. (2019). Accessed: 26/05/2023 [Online]. Available: https://arxiv.org/abs/1901.08688. doi: https://doi.org/10.48550/arXiv.1901.08688.

[24] - P. Roy, S. Ghosh, S. Bhattacharya, and U. Pal, ‘Effects of Degradations on Deep Neural Network Architectures’, *arXiv preprint arXiv:1807. 10108*, 2018.

[25] - “Express JS” Express https://expressjs.com/es/ (accessed May. 26, 2023)

[26] - Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen and K. C. Tan, "A Survey on Evolutionary Neural Architecture Search," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 2, pp. 550-570, Feb. 2023, doi: 10.1109/TNNLS.2021.3100554.

[27] - K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," in *Evolutionary Computation*, vol. 10, no. 2, pp. 99-127, June 2002, doi: 10.1162/106365602320169811.

[28] - TensorFlow Developers. “TensorFlow JS: Model Conversion” TensorFlow https://www.tensorflow.org/js/guide/conversion (accessed May. 26, 2023)