APLICACIÓN ANDROID PARA LA IDENTIFICACIÓN DE ENFERMEDADES DERMATOLÓGICAS

Grado en Ingeniería Informática

Autor: Nahuel Sánchez Rodríguez

Tutor: Francisco José Santana Pérez

07/2020

ULPGC
Universidad de Las Palmas de Gran Canaria
Escuela de Ingeniería Informática

Tabla de contenido

Introducción

El archipiélago canario es elogiado en numerosas ocasiones como “Islas Afortunadas” entre otras razones por su clima primaveral durante casi la totalidad del año. Contamos con un clima subtropical que nos permite disfrutar largas temporadas de sol y buenas temperaturas. Sin embargo, estas condiciones tan atractivas producen a su vez efectos negativos en la población, especialmente los jóvenes, pues infravaloran los perjuicios de la radiación solar, lo cual se traduce en futuros problemas dermatológicos de diversa gravedad.

Por regla general, la mayor parte de la población afirma no acudir con frecuencia a un centro especializado para ser revisado por un profesional de la materia y prevenir cualquier incidencia cutánea. Esta falta de compromiso se debe al desconocimiento sobre la gravedad que pueden conllevar determinadas enfermedades, así como las largas listas de espera a las que tenemos que enfrentarnos para poder acudir a través de la Seguridad Social.

Por otro lado, desde muy pequeño empecé a tener intereses acerca de la tecnología, especialmente ordenadores y móviles. Con el paso de los años lo que inicialmente era una afición se terminó convirtiendo en una vocación y decidí que quería dedicar mi futuro profesional al mundo de la informática. Sin embargo, esta ambición no terminaba ahí, pues quería adquirir conocimientos para posteriormente poder mejorar la calidad de vida de otras personas. Con esas premisas, tras largas búsquedas de fuentes de inspiración para afrontar el último escalón de la etapa universitaria, llegué a la conclusión de que quería crear un producto software innovador a la vez que simple y, sobre todo, que pudiera beneficiar a nuestra sociedad. De esta manera crearía una aplicación móvil que acompañada de Inteligencia Artificial pudiera ayudar a las personas a conocer si estas eran susceptibles de padecer una enfermedad en su piel. Además, con el objetivo de luchar contra la propagación de información falsa que se produce en la red a diario, he volcado en la aplicación información sobre cada una de las enfermedades que el sistema puede reconocer. Dicha información ha sido obtenida a través de una organización médica de gran prestigio y sin ánimo de lucro llamada Mayo Clinic.

En resumen, tras lo expuesto anteriormente, pretendo crear un software nativo para el sistema operativo Android que permita a los usuarios tomar una fotografía de una mancha cutánea y que, mediante una conexión a un servidor remoto y una red neuronal alojada en él, la aplicación devuelva al usuario la enfermedad que es posible que padezca e información relevante de esta. Así mismo, dentro de este producto podrán mantener un historial de todas las manchas fotografiadas y buscar centros médicos especializados próximos a su ubicación actual o más reciente.

Estado del Arte

Aunque el origen de la inteligencia artificial como concepto se remonta a 1956, en los últimos años hemos vivido una época dorada para la misma, pues se han logrado increíbles avances permitiendo al ser humano aplicarla en multitud de contextos. Por ejemplo, hoy en día vivimos rodeados de dispositivos dotados de esta inteligencia, véase los smartphones, electrodomésticos, vehículos, etcétera. Pero este fenómeno no acaba aquí, sino que su uso también se destina al análisis de mercado, previsión de ventas de un producto, detección de estafas e, incluso, cada vez es más frecuente llamar a un número de atención al cliente y ser atendido por un ordenador inteligente. Como vemos, esta rápida evolución ha desembocado en la creación de multitud de soluciones software y hardware que ayudan a resolver problemas de diversa índole como, por ejemplo, la sanidad. Por ello, si realizamos unas pocas búsquedas en la tienda de aplicaciones de Android, podemos encontrar cuatro productos similares a este Trabajo de Fin de Título. Sin embargo, todas tienen algunas carencias que son solventadas en mi aplicación y, a su vez, disponen de características que podrían ser interesantes añadir en un futuro.

- La primera aplicación que analizaremos será "**Medgic - Analiza y detecta enfermedades de la piel**" que cuenta con más de 100.000 descargas y una calificación de cuatro estrellas sobre cinco y sin coste de adquisición. De todas las opciones que analizaremos, quizás esta sea la más profesional, pues cuenta con una gran variedad de lesiones dermatológicas que es capaz de reconocer y, por otro lado, se recalca la importancia de acudir a un dermatólogo, aunque se esté empleando su propio software. Como característica destacable ofrece una serie de preguntas cuando analizamos una nueva foto para saber desde cuándo tenemos la mancha cutánea y así recabar más información. El mayor inconveniente que encuentro al sistema es su dificultad inicial pues si bien no es excesiva, para personas mayores o de escaso conocimiento tecnológico puede suponer una curva de aprendizaje algo pronunciada. Más adelante, veremos que en mi propuesta software he intentado en todo momento combinar una solución moderna y eficiente, con la simpleza de poder realizar cualquier acción en menos de cuatro clics.

- La siguiente aplicación encontrada recibe el nombre de "**DermIA - Análisis Cáncer Piel manchas con Cámara**". Siendo nuevamente gratuita y con más de 10.000 descargas y una nota media de 4.4, esta aplicación se describe a sí misma como una herramienta totalmente fiable para saber si padecemos un cáncer de piel. Ahora bien, comencemos a describir los puntos fuertes y débiles de la misma. Como aspecto positivo destacaría la posibilidad de reflejar en un gráfico del cuerpo humano a qué parte de este se corresponde la imagen analizada, pero, por el contrario, presenta ciertas flaquezas que podrían ser consideradas incluso peligrosas para la salud. El software indica en forma de porcentaje la probabilidad de padecer un cáncer, de manera que afirma la existencia de una patología de extrema gravedad, en lugar de recalcar la importancia de acudir a un centro especializado donde poder confirmar la predicción. Por otro lado, la aplicación se limita a predecir si eres positivo o negativo, no realiza un análisis de la imagen con el objetivo de intentar predecir qué enfermad concreta pudieras estar padeciendo. Como último

punto y menos relevante, hay que destacar el diseño de la aplicación que es demasiado colorida para mi gusto, provocando un rechazo visual durante su uso. Sin duda, considero un fallo el hecho de no advertir al usuario de que el uso de su aplicación no sustituye a ningún especialista ni tiene validez como informe médico. En *SkinScanner* el usuario visualiza en la primera pantalla un mensaje permanente de aviso donde se informa que está utilizando un software experimental y que debe acudir a un centro dermatológico.

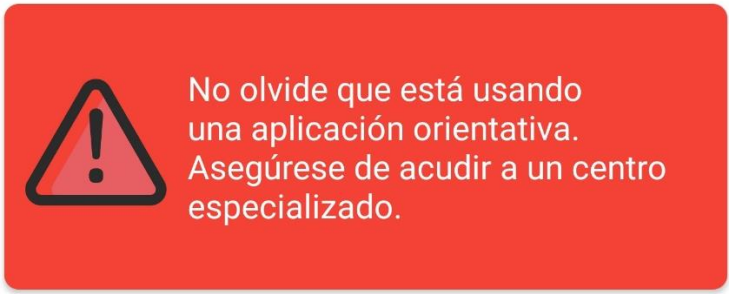


Ilustración 1. Mensaje de precaución en SkinScanner.

- En tercer lugar, tenemos "**Miiskin - Melanoma Skin Cancer**" la cual a pesar de ser gratuita cuenta con multitud de críticas pues desde el primer momento de uso se nos obliga a abonar una importante cuota mensual para no perder las funcionalidades de esta. Además, como su nombre indica, está limitada a reconocer un único tipo de lesión, el melanoma. Si bien su diseño es moderno y actual, carece de traducciones completas del sistema lo cual afea la experiencia del usuario final. Considero que este producto a pesar de contar con más de 100.000 descargas ofrece un servicio basado en publicidad engañosa y su uso está demasiado limitado al únicamente reconocer un cierto tipo de enfermedad.

- Para finalizar el análisis de las aplicaciones disponibles en el mercado, concluiremos simultáneamente con "**Smart Skin Cancer Detection**" y "**Visus: Skin Cancer Detection**". Ambas han sido descargadas por menos de 10.000 usuarios y poseen malas calificaciones sin apenas reseñas. La primera de ellas cuenta con un diseño completamente obsoleto y únicamente diferencia entre tres tipos de enfermedad. Además, ambas junto con la ya analizada "**Miiskin**", cargan los modelos en las propias aplicaciones lo cual incrementa notablemente el peso de estas llegando una de ellas a ocupar en nuestro dispositivo un mínimo de 134 megabytes. Por todas estas razones considero que son peores opciones que las anteriores y, por ello, cuentan con un número de descargas muy inferior.

Una vez analizadas las aplicaciones que considero más relevantes, procedo a exponer las ventajas de mi software frente a estas:

✓ Nuestra aplicación permite desde un menú de navegación inferior muy simple acceder a todas las opciones disponibles, logrando una navegación veloz y sencilla.

- ✓ También podemos buscar los dermatólogos más cercanos en función de nuestra localización, pues considero que para una persona preocupada por una mancha en su piel es de gran utilidad conocer los centros más próximos sin tener que salir de la aplicación. Además, al mostrar este listado incitamos de forma indirecta al usuario a acudir a un especialista en vez de conformarse con la predicción realizada por nuestra red neuronal.

- ✓ Por otra parte, hemos creado una infraestructura muy eficiente pues contamos con un servidor en **Google Cloud Platform**, el cual explicaremos en detalle más adelante, que nos permite cargar nuestro modelo en él y realizar todas las predicciones mediante conexiones al mismo. De esta manera logramos reducir en gran medida el peso de la aplicación, así como hacer un uso más eficiente de los recursos del dispositivo móvil. Bajo mi punto de vista esta es una de las características más importantes de mi producto, pues siempre he considerado que una aplicación pesada y cuyo funcionamiento no es fluido produce una sensación negativa en el usuario y es más que probable que sea desinstalada en cuestión de minutos.

- ✓ Para concluir, una vez se analiza una imagen, nuestro software proporciona información sanitaria verificada para que se puedan adquirir conocimientos sobre la posible patología que se padece, además de mostrar algunos consejos para reconocerla. De esta manera evitamos que el usuario tenga que acceder a su navegador para obtener información que, además, pudiera provenir de una web sin ningún tipo de garantía, contribuyendo a la difusión de información falsa.

Objetivos

Para la realización completa de este proyecto se han fijado cuatro objetivos principales que serán posteriormente unificados y materializados en forma de aplicación móvil.

- En primer lugar, para lograr un reconocimiento eficiente de las imágenes que aporte el usuario debo **entrenar una red neuronal** hasta alcanzar unos niveles de precisión adecuados. Debido a la aleatoriedad de las manchas cutáneas dicha tarea requerirá muchas horas para encontrar el modelo que mejor se adapte a nuestras necesidades y realizar el pertinente entreno de este.

- Por otra parte, nos encontramos en la disyuntiva de cómo utilizará el usuario final nuestra inteligencia artificial. Por un lado, el modelo podría haber sido cargado en la aplicación localmente, pero esto, como hemos visto, conllevaría un aumento notable de espacio requerido en la memoria de los dispositivos, así como un consumo elevado de los recursos. Por ello, para lograr una solución más eficiente surge nuestro segundo objetivo, el cual consistirá en **montar un servidor externo**, accesible desde cualquier parte del mundo, para que almacene y cargue el modelo neuronal. Además, dicho servidor será el encargado de recibir las peticiones de los usuarios para procesar sus fotos y obtener la predicción correspondiente.

- Una vez tenemos un sistema de reconocimiento de enfermedades dermatológicas y su pertinente infraestructura de alojamiento, podemos centrarnos en la **localización de dermatólogos próximos a la ubicación del usuario**. Esta funcionalidad permitirá acceder desde la misma aplicación a un listado de centros especializados, ordenados en función de la distancia a los mismos, incluyendo información relevante como la calificación media de otras personas, su dirección y un acceso directo a la aplicación Google Maps para poder conocer el trayecto más corto al lugar en cuestión. Este objetivo será posible gracias al uso de una API de Google llamada “Google Places API”.

- Finalmente llegamos a nuestro último objetivo: crear una **aplicación Android nativa**, desarrollada en Kotlin, que permita unificar todas las funcionalidades descritas en un producto software sencillo de utilizar, actualizado con las últimas librerías, modelos y patrones de diseño. Dicho producto deberá ser capaz de interactuar con nuestra API, así como realizar la conexión con nuestro servidor externo para el procesamiento de imágenes. Adicionalmente, tal y como se indicó en la introducción, los usuarios tendrán acceso a un historial de todas las predicciones realizadas hasta el momento, para lo cual se hará también uso de bases de datos locales mediante modernas librerías presentadas por Google recientemente. De esta manera el usuario tendrá la tranquilidad de que sus imágenes solo se almacenan en su dispositivo, mejorando así la **privacidad** y fluidez de la aplicación.

En el momento en que se hayan alcanzado todos los objetivos se habrán trabajado profundamente las competencias descritas en la propuesta de TFT (TFT-01). Estas comprenden la capacidad de elegir concienzudamente las mejores herramientas

software para nuestra aplicación, así como la creación de soluciones software robustas, a prueba de errores y teniendo en cuenta aspectos éticos como la privacidad de los usuarios. Todo ello bajo los lenguajes de programación más actuales y eficientes como son Kotlin y Python, a la vez que se emplean las estructuras de datos más ventajosas para nuestra finalidad, mediante librerías que serán descritas en un futuro.

Por último, se habrá demostrado tener la suficiente capacidad de desempeñar con éxito todas las fases que comprenden el desarrollo de un proyecto, desde la lluvia de ideas y creación de los casos de uso, hasta la realización de pruebas para comprobar si se cumplen los requisitos de usuario, pasando por la construcción del software aplicando algunos de los patrones de diseño vistos durante la carrera como podría ser el **patrón Singleton** para el acceso a la base de datos local mediante una única instancia; o el **patrón arquitectónico MVVM** aplicado para la visualización del contenido de la misma.

¿Por qué resulta conveniente contar con una única instancia para acceder a nuestra base de datos? La razón principal tiene que ver con los recursos del smartphone, ya que el acceso a estas fuentes de datos tiene un cierto coste y, dada la naturaleza de nuestro proyecto, el usuario puede estar continuamente accediendo a registros almacenados de manera que, si creásemos nuevas instancias en cada acceso, el consumo de recursos sería desorbitado causando un empeoramiento del rendimiento de la aplicación y el consiguiente descontento del usuario. Con el patrón Singleton nos aseguramos de que cada vez que deseamos instanciar la base de datos, se verifica primero si existe o no una instancia ya creada. En caso afirmativo se devuelve esta o, de lo contrario, se crea.

¿Y en qué consiste el patrón MVVM? Pues bien, este patrón denominado Modelo-Vista-VistaModelo es una variación del más que conocido Modelo-Vista-Controlador, donde se separan los datos de la aplicación, la interfaz de usuario y la lógica de negocio. Cuando se produce cualquier modificación es necesario que el controlador notifique y actualice la vista. En cambio, en MVVM, a pesar de que también hay una clara separación entre la vista y los datos, todo cambio de estos se refleja de forma automática en la interfaz y viceversa. Esto es una clara ventaja para nuestra aplicación pues el listado del historial de muestras cutáneas será actualizado automáticamente cuando haya un nuevo registro o se elimine alguno existente. De igual manera será aplicado para el listado de dermatólogos próximos a la localización del usuario.

Recursos necesarios

Software

Android Studio es el entorno de programación para Android por excelencia desde el año 2014. Está basado en IntelliJ IDEA de JetBrains y es totalmente gratuito. Permite crear aplicaciones para todo tipo de dispositivos con sistema operativo Android como smartphones, tabletas, relojes, televisiones, etcétera. Permite crear software robusto y combinar en una misma pantalla el desarrollo de la interfaz de usuario con la clase responsable de la lógica correspondiente. En mi caso, Android Studio ha sido empleado para el desarrollo de la aplicación final a través del lenguaje de programación Kotlin.

PyCharm es otra de las soluciones software creadas por la empresa JetBrains y enfocada al desarrollo de todo tipo de proyectos bajo el lenguaje de programación Python. En mi experiencia como estudiante ha sido siempre de gran utilidad para trabajar con redes neuronales. En esta ocasión, PyCharm fue clave a la hora de elaborar un servidor que recibe una imagen a través de una petición HTTP y la procese con nuestro modelo neuronal para devolver una predicción.

Para entrenar dicho modelo he empleado **Google Colaboratory**, una herramienta disponible para todos donde también podemos crear nuestros proyectos de Python y, además, contar con acceso gratuito a GPUs remotas que permiten realizar procesos costosos en menor tiempo del que emplearíamos usando nuestro propio ordenador.

Hardware

Para las pruebas de funcionamiento de la aplicación nativa he empleado mi teléfono personal, un **smartphone** con Android 10 y una pantalla de 6.67 pulgadas.

Por otro lado, para el alojamiento y gestión del servidor he empleado **Google Cloud Platform**, una plataforma que recoge todas las aplicaciones web que ofrece la compañía estadounidense para desarrolladores. Afortunadamente, a los nuevos usuarios se nos otorga crédito gratuito para poder trabajar con sus herramientas por tiempo limitado, de manera que, si deseamos tener acceso ilimitado en el tiempo y con mayor número de funcionalidades disponibles, deberemos abonar una cuota mensual. Sin embargo, este periodo de prueba será más que suficiente para poder tener mi propia máquina virtual Linux en la que almacenar los archivos necesarios tanto de la red como del servidor mencionado. El hecho de emplear una máquina virtual con crédito de prueba ha dificultado algunas tareas como veremos más adelante y también ha requerido que cada vez que finalizaba el trabajo sobre el servidor, desconectara la máquina para evitar el gasto innecesario de crédito y mantener un margen suficiente de cara a la exposición del proyecto.

Librerías

Para la creación de la red neuronal, tras numerosas pruebas con diferentes alternativas y modelos de creación propia, se adoptó el uso de una librería conocida como **FastAi**. Esta herramienta es de código abierto y está construida sobre **PyTorch**, un paquete que permite la realización de grandes cálculos a través del uso de la GPU, siendo ampliamente utilizado en el mundo de la inteligencia artificial. FastAi nos permite emplear modelos clásicos como DenseNet, MobileNet, Xception, entre otros. Sin embargo, me decanté por esta herramienta debido a sus increíbles funcionalidades que permiten realizar un entrenamiento progresivo, adaptando así ciertos hiperparámetros según las necesidades del modelo tras cada iteración. Además, al emplear esta herramienta nos aseguramos de que se están aplicando las mejores técnicas y métodos de entrenamiento actuales y, lo más importante, de forma semi-autónoma. Por si estas características no fueran suficientes puntos a favor, también nos ofrece una serie de métodos para visualizar gráficamente la evolución de las pérdidas del modelo tanto del entrenamiento como de la validación, facilitando enormemente la labor de análisis de los resultados.

Ahora bien, de nada nos serviría contar con las mejores herramientas para trabajar con redes neuronales si no disponemos de un dataset para entrenarlas. Es aquí donde entra en juego **Kaggle**, una plataforma que permite a cualquier persona interesada en redes neuronales encontrar repositorios de datos con los que poder desarrollar sus propios modelos, así como explorar y comentar los creados por otros usuarios. Resulta de gran utilidad poder conocer qué pasos han seguido otros desarrolladores, para obtener buenos resultados de manera que, a la hora de llevar a cabo tu propio entrenamiento, no es necesario empezar de cero, sino que cuentas con una base mínima que te sitúa en el camino correcto para encontrar la mejor red neuronal. Para este proyecto se ha empleado un dataset de diez mil imágenes denominado “Skin Cancer MNIST: HAM10000”, que nos aporta dicha cantidad de muestras, todas entremezcladas, en dos carpetas y, a través de un documento csv debemos separarlas en las siete categorías en las que se dividen. Este proceso será explicado en siguientes apartados del documento.

El empleo de la librería FastAi supondrá a su vez ciertas dificultades para la implementación en nuestro servidor Linux, pues su instalación supone un elevado consumo de memoria, capacidad muy limitada en las máquinas virtuales que nos ofrece Google en su plan de prueba. Para solventar este inconveniente, entra en juego una distribución abierta de Python y ampliamente conocida: **Anaconda**, la cual está enfocada al aprendizaje automatizado y a la ciencia de datos. Permite simplificar en gran medida la instalación de paquetes software y, gracias a ello, podré instalar exitosamente FastAi en mi máquina virtual remota.

Una vez tenemos la infraestructura preparada para alojar nuestro servidor Python, encargado de ejecutar el modelo y realizar las predicciones, será necesario construir dicho servidor. Su codificación manual resultaría una tarea extremadamente ardua, pero, por suerte, existe una librería minimalista denominada **Flask** que simplifica la labor. Nos permite crear aplicaciones web rápidamente y sin necesidad de codificar enormes bloques de código. Esta librería también deberá ser instalada en la máquina virtual para el correcto funcionamiento del servidor.

En último lugar, me gustaría mencionar las librerías empleadas para el óptimo funcionamiento de la aplicación móvil. Comenzaremos con una de las más importantes: **Room**, la cual nos ofrece utilidades para la creación de bases de datos locales, almacenando cualquier objeto desarrollado en el proyecto. Su estructura se divide en tres componentes: la base de datos que es el punto de partida donde se define la identidad de esta y su conexión con los datos; las entidades, que son cada una de las tablas que se almacenan en dicha base de datos y, por último, el acceso a los datos (DAO) que posee todos aquellos métodos necesarios para acceder a los datos, insertar nuevos o modificar los ya existentes. En mi caso decidí crear dos tablas, destinando una de ellas al alojamiento de las predicciones realizadas, junto con una serie de datos adicionales, como la ubicación de la imagen correspondiente, fecha, hora y anotaciones. La segunda tabla se destina a almacenar la información relevante de los dermatólogos encontrados por la API de Google, Places API, la cual es otro recurso fundamental en nuestro proceso de creación.

Places API nos permite realizar peticiones HTTP a través de las cuales obtendremos información acerca de lugares de interés. He empleado este recurso para obtener un listado de lugares categorizados como centros dermatológicos y mostrarlos en la aplicación. La respuesta de la API se recibe en formato JSON, pudiendo ser convertida a GSON por una librería del mismo nombre que, además, nos permite crear clases y métodos que realicen la transformación de una cadena GSON a objetos y clases en lenguaje Kotlin. De esta manera conseguimos tener entidades manejables y que pueden ser almacenadas en la base de datos mencionada en los apartados anteriores.

Otra librería empleada es **Glide**, que nos permite lograr una gran fluidez en las interacciones con la interfaz, a pesar de que estemos manejando gran cantidad de imágenes, pues se encarga, de forma autónoma, de almacenarlas en caché de manera que no realizamos continuos accesos al disco del dispositivo para visualizarlas. Su curva de aprendizaje es muy leve y rápidamente podemos integrar la utilidad en nuestra interfaz de usuario.

Análisis, diseño y desarrollo

En este apartado vamos a describir detalladamente todo el proceso desde la concepción de la idea hasta la obtención de un producto software viable. Se especificará cómo ha sido el desarrollo, los errores cometidos, así como las decisiones tomadas para solventar los obstáculos encontrados en el camino.

Análisis

Desde el comienzo del curso dediqué mucho tiempo a la búsqueda de ideas y fuentes de información sobre las que basar mi proyecto de TFT. También mantuve reuniones con mi tutor para conocer su experiencia en anteriores proyectos y aprender de sus consejos a la hora de decidir en torno a qué idea giraría mi trabajo. Hasta entonces, únicamente tenía claro que quería combinar la programación para dispositivos móviles con la inteligencia artificial, ya que como he explicado en la introducción, son dos elementos indispensables en nuestro día a día. De esta manera, tras navegar por la web de Kaggle durante varios días encontré un tema que siempre me había parecido interesante, las enfermedades dermatológicas. A raíz de este hallazgo comenzó un periodo de lluvia de ideas donde apuntaba toda aquella que consideraba enriquecedora para el proyecto, aunque desgraciadamente varias de ellas tuvieron que ser descartadas por falta de tiempo para su realización debido a la excesiva complejidad técnica.

Para no centrarme exclusivamente en mis propias ideas decidí consultar con mi entorno acerca de esta propuesta y qué utilidades creían necesarias en una aplicación de este tipo y cuáles prescindibles. Esta acción fue realizada en un primer momento como simple curiosidad y resultó de gran ayuda para lograr un producto software completo y adaptado a todos los públicos, desde personas con experiencia en el mundo digital, hasta personas de cierta edad y con escasa soltura en este campo. Por ejemplo, algo tan enriquecedor como la búsqueda de centros médicos próximos surgió a raíz de una charla informal con un familiar, el cual acude con relativa frecuencia al dermatólogo.

Al mismo tiempo que iba perfilando un primer concepto, continuaba manteniendo charlas distendidas con mi tutor para conocer su opinión ante mis sugerencias y aplicar las recomendaciones que me indicaba. De este modo llegamos a la base, a partir de la cual se construiría toda una infraestructura software que enlazaría campos tan distintos como la salud y la informática. Crearía una aplicación capaz de predecir si una mancha en la piel podía ser susceptible de corresponderse con una enfermedad dermatológica, además de mostrar al usuario los centros dermatológicos más cercanos por si necesitara acudir a uno. Ahora bien, aunque los objetivos estaban claros, su implementación distaba mucho de estarlo. Iban a ser necesarias bastantes horas de investigación para resolver todos los problemas a los que tenía que enfrentarme.

Casos de uso

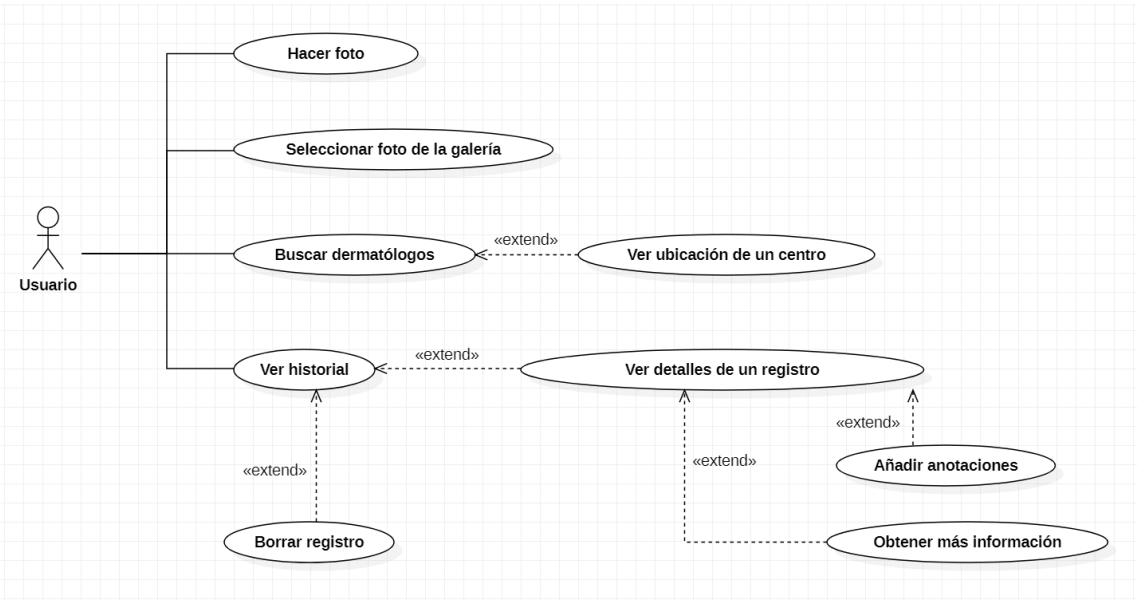


Ilustración 2. Diagrama de casos de uso

Antes de comenzar la etapa de investigación decidí elaborar un diagrama de casos de uso (*Ilustración 2*) para reflejar las posibles interacciones del usuario con la aplicación, mostrando así todas las funcionalidades que deberían estar disponibles al finalizar el desarrollo. En este diagrama podemos distinguir cuatro casos de uso principales en torno a los cuales giran el resto de los casos: tomar o seleccionar foto, buscar dermatólogos y ver historial. Cada uno de ellos tiene sus peculiaridades, de manera que, para poder comprenderlos correctamente, realizaré sus respectivas especificaciones.

Caso de uso: Tomar foto.

Descripción: El usuario realiza una fotografía de la mancha cutánea para que sea analizada.

Actores: Usuario.

Precondiciones: Permisos de acceso a la cámara y al almacenamiento concedidos.

Flujo normal:

- **Paso 1:** Hacer clic en el botón “Hacer foto”.
- **Paso 2:** Enfocar la superficie a fotografiar.
- **Paso 3:** Tomar la fotografía.
- **Paso 4:** Confirmar la imagen capturada.

Postcondiciones: La imagen es guardada en la memoria del teléfono y en el historial se crea un nuevo registro.

Excepciones: Si no hay conexión a internet o el servidor no está operativo, se muestra una ventana flotante informando al usuario.

Caso de uso: Seleccionar foto de la galería.

Descripción: El usuario selecciona una fotografía almacenada en la memoria de su dispositivo.

Actores: Usuario.

Precondiciones: Permisos de acceso a la cámara y al almacenamiento concedidos.

Flujo normal:

- **Paso 1:** Hacer clic en el botón "Seleccionar foto de la galería".
- **Paso 2:** Elegir la imagen deseada.

Postcondiciones: En el historial se crea un nuevo registro.

Excepciones: Si no hay conexión a internet o el servidor no está operativo, se muestra una ventana flotante informando al usuario.

Caso de uso: Buscar dermatólogos.

Descripción: El usuario descubre los centros dermatológicos próximos a su localización geográfica actual o más reciente.

Actores: Usuario.

Precondiciones: Permiso de acceso ubicación concedido.

Flujo normal:

- **Paso 1:** Navegar a la vista "Dermatólogos".
- **Paso 2:** Hacer clic en el icono de actualizar.

Extensiones:

- **Paso 3:** El usuario desea ver la localización exacta de un centro.
  - Caso de uso: Ver ubicación de un centro.

Postcondiciones: El listado queda actualizado y en la parte superior se muestra el día y hora de la última búsqueda.

Excepciones: Si no hay conexión a internet se muestra una ventana flotante informando al usuario.

Caso de uso: Ver ubicación de un centro.

Descripción: El usuario puede acceder directamente a Google Maps para ver la posición exacta del centro dermatológico.

Actores: Usuario.

Precondiciones: Tener instalada la aplicación Google Maps en el dispositivo.

Flujo normal:

- **Paso 1:** Hacer clic en el botón “Abrir en Mapas”.

Postcondiciones: Se abre la aplicación Google Maps y nos muestra la ubicación del centro, así como imágenes e información relevante.

Excepciones: Si no hay conexión a internet se muestra una ventana flotante informando al usuario.

Caso de uso: Ver historial.

Descripción: El usuario visualiza un listado con todas las imágenes analizadas hasta la fecha.

Actores: Usuario.

Precondiciones: Permiso de acceso al almacenamiento concedido.

Flujo normal:

- **Paso 1:** Navegar a la vista “Historial”.

Extensiones:

- **Paso 2.1:** El usuario desea eliminar uno de los registros listados.
  - Caso de uso: Borrar registro.
- **Paso 2.2:** El usuario desea conocer más información acerca de un registro concreto.
  - Caso de uso: Ver detalles de un registro

Postcondiciones: Se muestra el historial ordenado del registro más reciente al más antiguo.

Caso de uso: Borrar registro.

Descripción: El usuario elimina un registro del historial.

Actores: Usuario.

Flujo normal:

- **Paso 1:** Hacer clic en el botón “Borrar”.

- **Paso 2:** Hacer clic en el botón “Confirmar” dentro de la ventana de confirmación.

Postcondiciones: El registro en cuestión desaparece del historial.

Caso de uso: Ver detalles de un registro.

Descripción: El usuario accede a una nueva vista para obtener más información acerca de un registro del historial.

Actores: Usuario.

Flujo normal:

- **Paso 1:** Hacer clic en el botón “Detalles”.

Extensiones:

- **Paso 2.1:** El usuario desea informarse acerca de la enfermedad predicha por la red neuronal.
  - Caso de uso: Obtener más información.
- **Paso 2.2:** El usuario quiere añadir una anotación al registro que está visualizando.
  - Caso de uso: Añadir anotaciones.

Postcondiciones: Se accede a una nueva vista donde se muestra más información acerca del registro seleccionado.

Caso de uso: Obtener más información.

Descripción: El usuario accede a información acerca de la enfermedad que ha predicho la red neuronal.

Actores: Usuario.

Flujo normal:

- **Paso 1:** Hacer clic en el botón “Obtener más información”.

Postcondiciones: Se muestra la nueva vista con la información correspondiente.

Caso de uso: Añadir anotaciones.

Descripción: El usuario añade anotaciones al registro que está visualizando.

Actores: Usuario.

Flujo normal:

- **Paso 1:** Hacer clic en el botón con el símbolo “+”.
- **Paso 2:** Escribir el texto que se quiera guardar como anotación.
- **Paso 3:** Hacer clic en el botón “Guardar”.

Postcondiciones: Se muestra la vista del registro con la nueva anotación añadida o modificada.

Observaciones: Se establece un límite de 240 caracteres para las anotaciones de cada registro.

Durante la lectura de estas especificaciones queda de manifiesto la simpleza de uso que tendrá la aplicación, donde se ocultará todo el proceso de análisis de la imagen para que el usuario únicamente deba preocuparse por escoger una fotografía y, posteriormente, acceder al resultado de la predicción.

Investigación

Ha llegado el momento de acercarme a diferentes recursos y tecnologías que nunca he empleado, necesitando así varias jornadas de aprendizaje para poder afrontar de manera sólida el desarrollo del proyecto. El punto de partida fue aprender a crear un servidor Python donde poder alojar mi red neuronal y a la que acceder mediante peticiones para analizar una imagen y recibir la predicción correspondiente. Debido a que nunca había creado una herramienta similar desconocía por completo cómo podía llevarla a cabo, pero gracias a la búsqueda de información en un blog oficial de Keras descubrí un framework denominado Flask que, como hemos mencionado, permite crear el servidor en unas pocas líneas de código. Consta de tres partes bien diferenciadas, comenzando con un método principal, el cual es ejecutado cuando se arranca el servidor, realizando la puesta en marcha de este y llamando a un segundo método, que será el responsable de cargar el modelo neuronal que se desarrollará en un futuro. Aquí es donde radica una de las principales ventajas de emplear un servidor remoto, ya que cargamos el modelo una única vez en lugar de tener que cargarlo desde la memoria del dispositivo cada vez que el usuario desea usar la aplicación. El código se completa con un tercer método, el cual es llamado mediante peticiones HTTP y se encarga de recibir la imagen como parámetro, analizarla con el modelo y devolver la predicción resultante. La implementación resulta sencilla con ayuda de la documentación mencionada anteriormente y el objetivo es dejar el programa completamente preparado para el último paso, cargar nuestro futuro modelo y realizar el análisis de la imagen que reciba.

A continuación, procedo a buscar información acerca del manejo y utilización de la función de cámara. Si bien tenía nociones en la creación de aplicaciones Android, gracias en buena medida a la realización, un año atrás, de un curso de especialización en la plataforma Udemy, desconocía cómo realizar correctamente la implementación del software de cámara en un proyecto de estas características. Existían dos alternativas, implementar de forma tradicional el software, el cual requería numerosos pasos, o bien, hacer uso de una librería denominada CameraX, la cual presumía de facilitar el desarrollo de una aplicación de cámara. Efectivamente, esta biblioteca de recursos me permitía crear mi propio software fotográfico e implementarlo directamente en una de las vistas de la aplicación, sin embargo, tras su codificación en Android Studio y la realización de diversas pruebas comprobé que las imágenes obtenidas no eran de la

calidad esperada, así como una serie de excepciones producidas aleatoriamente durante el manejo de la herramienta. De esta manera, a pesar de haber creado una vista totalmente personalizada y funcional, esta contaba con pequeños errores que no podían permitirse en una aplicación que busca la calidad y eficiencia por encima de todo.

Por estas razones, tras fallidos intentos por corregir los problemas presentes, decidí rectificar y realizar una implementación tradicional de la cámara, pues no solo me proporcionaría una vista por defecto, con todas las funcionalidades que permite el hardware de cada dispositivo, sino que, además, contaba con amplia y detallada información en la plataforma para desarrolladores de Android. El funcionamiento de esta utilidad es realmente sencillo, pues Google nos ofrece un método que permite lanzar la actividad de cámara como si de una vista más de nuestra aplicación se tratara. Llegado a este punto poseía los suficientes conocimientos para afrontar la implementación completa en la aplicación final, aunque por desgracia, continuaría encontrando algunos errores que iban a ralentizar y entorpecer el progreso del proyecto.

Tras haber superado los dos primeros obstáculos, tocaba adentrarse en el mundo de las APIs, donde escasamente había trabajado anteriormente. Iba a emplear la API de Google denominada Places API para obtener los centros dermatológicos próximos a la ubicación del usuario, tarea que parecía bastante complicada a primera vista. Sin embargo, contra todo pronóstico, la implementación de Places API fue realmente sencilla y únicamente se necesitó crear un proyecto en Google Cloud Platform, adquirir una clave de API para poder usarla en nuestra aplicación y generar la petición URL para, a través de esta, recibir en formato JSON toda la información acerca de los dermatólogos.

Si bien es cierto que en estas pruebas iniciales, al no haber implementado aún la funcionalidad del GPS, las pruebas eran realizadas introduciendo manualmente una latitud y longitud en la query. Además, esta herramienta daba la posibilidad de añadir parámetros adicionales para indicar en función de qué atributo ordenar los resultados y añadir palabras clave que permitieran una búsqueda más precisa. Una vez había creado la URL deseada y comprobado su correcto funcionamiento, procedo a la conversión del resultado JSON a objetos. Para ello empleo una página web denominada “jsonschema2pojo” en la cual introduciendo la cadena JSON, obtenida a través de la petición, me generaba los archivos Java necesarios para dicha conversión, realizada con la ayuda de la librería GSON. De esta manera, cada vez que se recibiera la cadena resultante, nuestro software sería capaz de generar los objetos y así trabajar con estos de forma más sencilla y eficiente.

Llegados a este punto me había encontrado con dos inconvenientes: El primero de ellos y menos grave, venía dado por las clases Java obtenidas puesto que la aplicación Android iba a ser desarrollada en Kotlin, de manera que debí realizar manualmente la transformación de cada clase de un lenguaje al otro, (*véase ejemplos en Anexo II, punto 4*). Por otra parte, siempre que realizamos búsquedas en aplicaciones como Google Maps, la cual se nutre de la misma API, entre otras, obtenemos resultados erróneos pues

muchos negocios están clasificados incorrectamente. En este caso, cuando establecía la búsqueda de dermatólogos obtenía a su vez resultados de peluquerías y centros de estética. Como este comportamiento era natural de la herramienta y los lugares registrados en su base de datos, tendría que desarrollar mi propio filtro para descartar aquellos lugares cuyos nombres contuvieran ciertas palabras indicadoras de que no eran centros dermatológicos. Dicha utilidad sería desarrollada más adelante en la aplicación final.

Aplicación

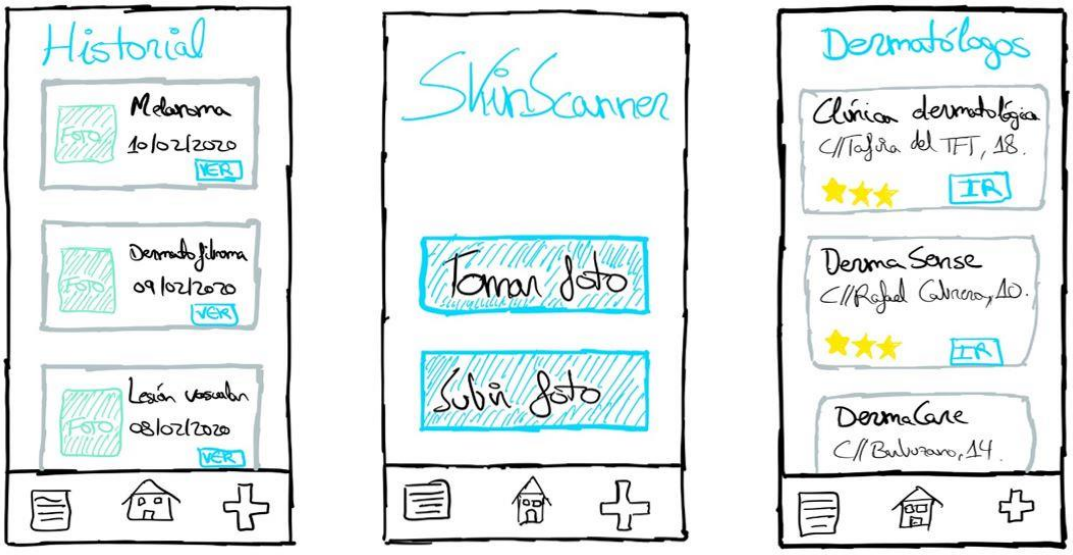


Ilustración 3. Boceto SkinScanner

Una vez he estudiado todas las nuevas herramientas que necesitaría para el desarrollo del proyecto, comienzo a diseñar unos bocetos *(Ilustración 3)* donde recogemos las principales ideas de diseño, siguiendo especialmente las directrices de Google y su diseño Material Design. Estos dibujos fueron realizados a mano en un dispositivo iPad con ayuda del Apple Pencil, permitiendo lograr resultados muy similares a los que se obtendrían en papel, pero con una calidad de imagen mucho mayor. Durante mi paso por varias asignaturas, siempre nos han recalcado la importancia de planificar correctamente un proyecto antes de lanzarnos a programar sin tener las ideas claras. Sin duda, realizar la especificación de casos de uso supone una gran ayuda para no olvidarnos ningún detalle de la aplicación y poder, también, imaginar de manera clara cómo deseo que se vean todas las funcionalidades en la aplicación. También he diseñado en numerosas ocasiones bocetos y mockups donde mostrar cómo será la versión final sin necesidad de implementar código y, aunque en ocasiones pueda resultar un proceso lento y aburrido, cuando tienes que comenzar a codificar agradeces enormemente contar con unas pautas que te faciliten el proceso de diseño y te ayuden a saber cuándo has finalizado correctamente una parte del proyecto.

De este modo llego a uno de los núcleos del TFT: la creación de la aplicación móvil para Android. Antes de nada, debo mencionar que inicialmente la idea era desarrollar el proyecto empleando el lenguaje Java pues, durante estos años, ha sido el protagonista en la mayoría de las asignaturas. Sin embargo, tras la reciente aparición de Kotlin quise aprender este lenguaje, pues era ampliamente conocido que se trataba de un lenguaje mejor optimizado y sencillo de usar de cara a la programación móvil. De esta manera, comencé a formarme mediante cursos online y me dispuse a crear la totalidad de la aplicación en este nuevo lenguaje, decisión que, adelanto, fue todo un acierto.

Ilustración 4. Vista principal

El comienzo de este proceso tuvo lugar con la creación de las tres vistas principales, correspondientes con las expuestas en el boceto, y que serían las encargadas de permitir al usuario tomar o seleccionar una foto de su galería para el análisis de esta, acceder al historial de imágenes ya analizadas, o bien, conocer los centros dermatológicos cercanos a la ubicación en la que se encontrara en ese instante. Cada una de estas se considera un fragmento y en conjunto construyen un sistema de navegación a través de un menú inferior. El fragmento central sería la primera pantalla que visualizará el usuario al entrar a la aplicación, de manera que en ella dispuse los

botones que permiten la captura de imágenes, así como la subida de estas desde la galería del dispositivo. Adicionalmente, para evitar un mal uso de la aplicación se añadió una alerta permanente con el objetivo de recordar el carácter experimental del proyecto y la indispensable recomendación de acudir a un centro especializado *(Ilustración 4).*

Si nos fijamos en la parte superior de la vista, encontramos un logotipo diseñado por mí para este proyecto. Es un elemento que no aporta valor en cuanto a funcionalidad, pero considero que tener una imagen propia con la que identificarse supone un incremento de la calidad percibida. Para su elaboración acudí a la web Canva que es una plataforma de creación donde cualquier persona puede dar rienda suelta a su imaginación y realizar todo tipo de creaciones. Además de la imagen mostrada en la vista, desarrollé una imagen destinada a la pantalla de bienvenida de la aplicación y al icono de esta. En la siguiente *Ilustración* se muestran ambas:

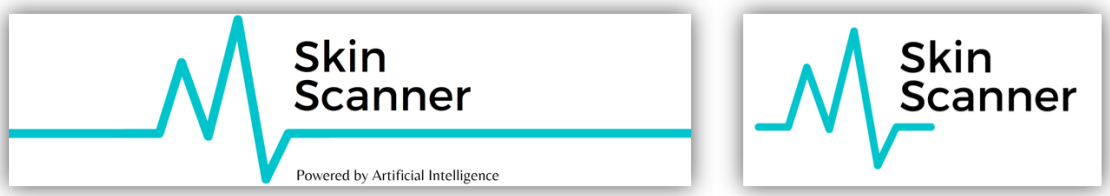


Ilustración 5. Logotipo SkinScanner.

La captura de fotos, tal y como he explicado anteriormente, se llevará a cabo mediante la implementación ofrecida por Google, encontrando inconvenientes únicamente en la creación de los archivos que serán contenedores de dichas imágenes. Comencé solicitando los permisos de almacenamiento y acceso a cámara necesarios para poder realizar fotografías y guardarlas en la memoria del dispositivo. Posteriormente, implementé la llamada a la cámara en sí la cual se realizaba mediante un objeto "Intent" que es utilizado para el desplazamiento entre vistas dentro de la aplicación. Con esto conseguimos que el sistema realice de forma automática la apertura de la vista y cuando el usuario confirma que la fotografía realizada satisface sus necesidades, el Intent genera la imagen y la almacena en una ruta que nosotros debemos haber creado con anterioridad, de forma que podemos darle el nombre que deseemos a cada fotografía. En mi caso, decidí que cada archivo debía recibir el nombre de la aplicación junto con la fecha y hora en la que se realizó la fotografía, (*véase Anexo II, punto 6*). Para solventar el problema del almacenamiento debí realizar una investigación por la web hasta encontrar la causa del error. Descubrí que debía crear dentro del fichero de configuración "manifest.xml" un proveedor para la aplicación de manera que la creación y almacenamiento de imágenes estuviera bajo la responsabilidad de una autoridad propia, (*véase Anexo III, punto 1*).

En segundo lugar, se implementó la subida de imágenes a la aplicación, proceso que no supuso ningún inconveniente pues la documentación oficial estaba muy

detallada. Al igual que para la cámara, debíamos crearnos un Intent con el cual se lanza el selector de imágenes dentro de la galería del dispositivo y, cuando el usuario ha escogido una fotografía, devuelve la ruta completa donde se localiza dicho archivo. De esta manera no necesitamos crear nuevos ficheros para almacenar estas imágenes, lo que generaría duplicados innecesarios y un gasto de memoria absurdo, sino que almacenamos la ruta original y cuando se necesite cargamos la imagen desde dicha ruta.

Hemos llegado a un punto en el cual nuestra aplicación es capaz de tomar fotografías desde su propia interfaz y almacenarlas en la memoria del dispositivo, además de permitir la subida directa desde la galería. Sin embargo, para poder ser analizadas y almacenar los resultados y demás información relevante, necesitaremos una base de datos local donde poder guardar dichas entidades. En un primer momento consideré hacer uso de una librería externa, ya que anteriormente había trabajado con ella y, por tanto, su implementación sería sencilla. En cambio, al haber optado por Kotlin, tras investigar por la red descubrí que junto a este nuevo lenguaje Google había lanzado su propia librería para la creación de bases de datos SQL, denominada Room. Con el objetivo de crear una aplicación actualizada con las últimas herramientas y tecnologías decidí dar una oportunidad a este recurso y seguir las indicaciones de la documentación oficial para su correcta implementación. Además, para hacer un uso responsable de los recursos del dispositivo y lograr un rendimiento óptimo, implementé el patrón Singleton para los accesos a esta base de datos, de manera que, como ya hemos visto, en vez de crear una nueva instancia cada vez que deseamos acceder a la información almacenada, reutilizamos la primera instancia creada. La ventaja principal que ofrece esta herramienta de almacenamiento consiste en que las entidades que guardamos se crean como una clase tradicional, donde cada atributo de la clase es una columna de la tabla, simplificando enormemente el trabajo de almacenamiento de objetos complejos.

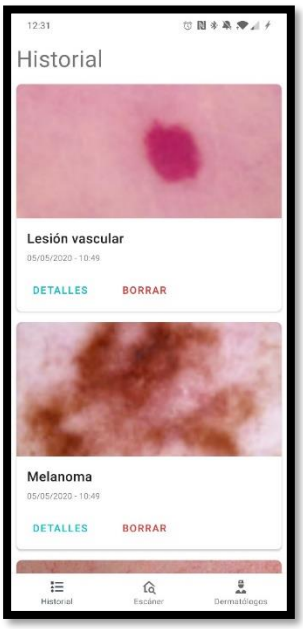


Ilustración 6. Vista del historial

Para el funcionamiento de este sistema elaboré una clase “AppDatabase” la cual extiende de “RoomDatabase” (*véase Anexo II, punto 1*). En ella tenemos una instancia la cual será devuelta a cualquier llamada que requiera acceder a los datos almacenado y, además, empleo anotaciones propias de la librería para determinar a qué clases pertenecen los dos tipos de objetos almacenado. Por otra parte, es necesario crear clases de acceso a datos, “DAO”, para incluir en ellas todas las sentencias SQL que queramos realizar sobre nuestra base de datos. Tanto para los centros dermatológicos como para los registros de imágenes, creo funciones de inserción y extracción, añadiendo para el segundo caso un método de actualización.

Otro aspecto fundamental es la elaboración de clases entidad, que serán dos en este caso. Para ello se vuelven a emplear las anotaciones de Room indicando el nombre que queremos darle a la tabla, que representará cada entidad, así como la declaración de cada columna, indicando su nombre y tipo de datos, (*véase Anexo II, puntos 2, 3*).

Una vez he logrado empaquetar los datos deseados en entidades y estas han sido almacenadas en la base de datos local, el siguiente paso natural es la visualización del historial *(Ilustración 6)* donde se mostrarán todas las imágenes analizadas, cuando esté operativo el servidor y la red neuronal. De momento, se almacenarán resultados prefijados de manera que podamos continuar con el desarrollo de la aplicación sin necesidad de tener lista la infraestructura restante.

Tal y como se reflejó en el boceto, mi idea era disponer cada registro en forma de tarjeta y a su vez mostrarlos en una lista deslizable verticalmente. Para ello emplearé un recurso denominado “RecyclerView” que permite la creación de este componente indicando, mediante otro fichero XML, cómo será la vista de los elementos que se muestren en su interior. Posteriormente, se debe crear un adaptador que será el encargado de recoger cada atributo del objeto a visualizar y mostrarlo en su correspondiente campo dentro de la tarjeta. Para entender de forma sencilla el funcionamiento de un adaptador consideremos un único registro de imagen con su predicción, fecha, hora, localización en la memoria y anotaciones. El adaptador se encarga de tomar aquellos parámetros que le interesan y asignarlos a cada elemento de la interfaz dentro de la tarjeta creada. Adicionalmente, maneja los eventos de los botones que esta incluya, como por ejemplo borrar un registro, mediante el uso del repositorio correspondiente y la llamada al método de eliminación creado anteriormente. Es aquí donde se carga la imagen de cada registro empleando la librería Glide, de manera que se toma la imagen cacheada y se asigna al componente de la vista contenedor de la imagen, (*véase Anexo III, punto 2*).

En este punto, con el objetivo de mejorar la calidad del código, decidí aplicar un patrón arquitectónico recomendado en el desarrollo de aplicaciones Android, denominado MVVM (Modelo, Vista, Vista-Modelo), el cual hemos visto que permite separar la lógica de negocio de las vistas asociadas, generando código limpio, fácil de comprender y escalable. Además, otra de sus ventajas es la persistencia, evitando la pérdida de datos, aunque el sistema cierre nuestra aplicación para ahorrar recursos.

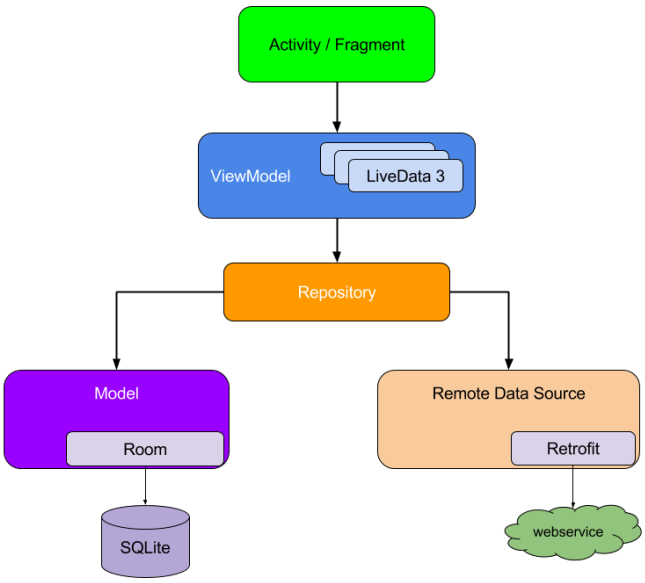


Ilustración 7. Arquitectura MVVM. Fuente: Android Developers

Como podemos observar en la *Ilustración 7,* cada componente únicamente depende del componente inmediatamente inferior a él, de manera que tenemos una estructura sólida y, además, gracias al uso del LiveData conseguimos la actualización en segundo plano de los datos mostrados, sin necesidad de realizar llamadas a la base de datos cada vez que se realiza una modificación o inserción. Adicionalmente, desde Google recomiendan crear una clase repositorio que sirva de enlace entre nuestra Vista-Modelo y el Modelo, directrices también aplicadas en *SkinScanner*. Esta clase será llamada cada vez que se quiera interactuar con la base de datos tanto para insertar, eliminar o mostrar información. La implementación de este patrón conllevó numerosas horas de investigación y aprendizaje pues era la primera vez que lo aplicaba en un software de relativa complejidad. Sin embargo, tras lograr el objetivo, considero que aporta calidad y limpieza al proyecto, siendo nosotros mismos los primeros beneficiarios, pues tener un código correctamente estructurado facilita el desarrollo.

Hasta este momento tenía un software capaz de obtener muestras fotográficas, almacenarlas en una base de datos local junto a otros datos relevantes y, además, mostrar todos los registros en un historial. El siguiente paso sería crear una vista similar para los dermatólogos cercanos a la ubicación del usuario, aplicando el mismo patrón arquitectónico. Comienzo implementando el código probado con anterioridad encargado de realizar la conexión y petición HTTP con Places API de Google, obteniendo, en formato JSON, el listado de centros próximos a la ubicación especificada en función de la latitud y longitud. Recordemos que a partir de esta respuesta obtendremos un objeto GSON para poder generar a partir de él todas las entidades contenidas en la respuesta de la API, (*véase Anexo III, punto 3*). De esta manera, puedo crearme una nueva clase donde incluir solo aquellos parámetros que me interesan y así almacenarlos en nuestra base de datos local, al igual que hice con las muestras dermatológicas. Google nos devuelve gran cantidad de información y por ello consideré que debía realizar una

selección de los aspectos más relevantes para el usuario final, como pudieran ser el nombre del negocio, su dirección y la calificación media de los usuarios de Google Maps. Me hubiera gustado incluir otro atributo que indicaba si el negocio se encontraba abierto en ese instante, pero, a pesar de los intentos para su implementación, decidí descartarlo pues muchos negocios no incluían este dato de manera que la API generaba excepciones a la hora de recopilar la información.

Quizás pueda resultar confuso el hecho de haber creado una tabla en la base de datos para guardar unos objetos que directamente son recibidos de la red, pero el motivo se encuentra, una vez más, en el uso eficiente de los recursos, porque al almacenar los registros puedo mostrar el listado cada vez que el usuario accede la vista en vez de tener que realizar una petición HTTP, con la consiguiente carga de trabajo del dispositivo y del servicio, restando a su vez fluidez a la interfaz. Además, por regla general una persona suele vivir en una misma zona, de manera que es innecesario generar continuas peticiones cuando los resultados serán siempre idénticos.

Uno de los problemas encontrados durante esta tarea requirió de mucho ingenio para su solución. Tal y como vimos al comienzo, cuando realizábamos una búsqueda, la API devolvía un listado en el cual aparecían negocios de estética, peluquerías, dentistas y demás centros no relacionados con la dermatología. Tras investigar las razones, me di cuenta de que era un problema de la propia base de datos de Google, pues muchos locales eran clasificados erróneamente como centros dermatológicos. Este hecho me llevó a barajar y probar diversas soluciones hasta encontrar la única realmente efectiva: filtrar los datos recibidos. De esta manera, tras recibir la respuesta de Places API, realizo un cribado descartando cualquier resultado que contenga palabras clave como “hair”, “rehabilitación”, “estética”, “peluquería”, “dental”, “dentista”, “dermoláser”, “láser”, “dermoshop”, “policlínico”, “podólogo”, etcétera.

Me gustaría mencionar también una idea frustrada durante este proceso, ya que a la hora de ver la interfaz que iba creando me daba cuenta de que sería ideal contar con una imagen de cada centro para así no solo percibir información textual sino gráfica, la cual siempre influye en la toma de decisiones. Sin embargo, tras intentar obtener dichos elementos de la propia respuesta de la API, no conseguí mostrarlos ni almacenarlos en la base de datos local. Resultó bastante frustrante no lograr el objetivo pues, bajo mi punto de vista, no solo habría enriquecido al usuario final en cuanto a la información recibida, sino que incrementaría el atractivo de esta herramienta de búsqueda.

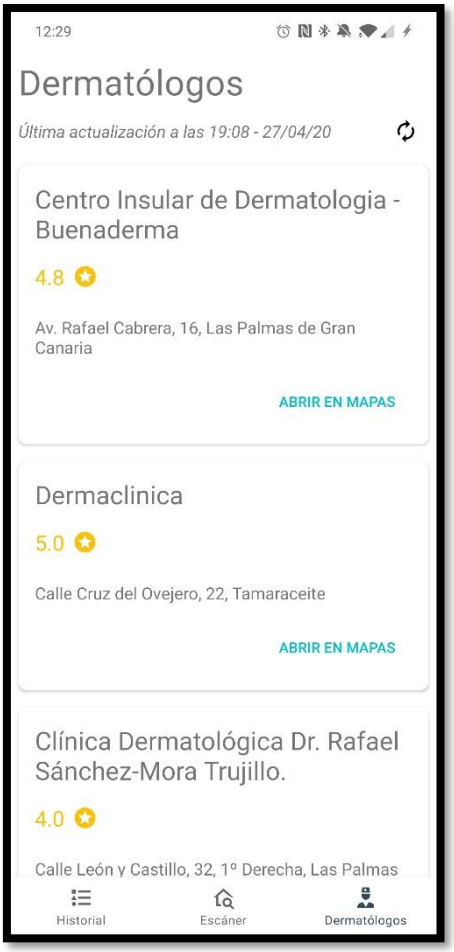


Ilustración 8. Vista de dermatólogos

De esta manera, tras aplicar el cribado mencionado y almacenar en la base de datos los registros válidos, solo quedaba mostrar la información en forma de tarjetas *(Ilustración 8)* dentro de una vista deslizable, de igual modo que hicimos con el historial. Esta tarea fue muy sencilla pues únicamente debíamos seguir el código ya desarrollado, adaptándolo a las características de estas nuevas entidades. Del mismo modo, para la conexión entre el modelo y la vista apliqué el patrón MVVM. Aunque visualmente se habían alcanzado los objetivos y tenía una herramienta de búsqueda bastante eficaz, el sistema todavía no recogía la ubicación actual del usuario, sino que realizaba la petición con unos valores de latitud y longitud preestablecidos. Por otra parte, cada tarjeta debía contar con un botón que permitiría ver en Google Maps la ubicación exacta del centro dermatológico para así poder conocer la ruta más rápida hacia este, por ejemplo.

Para conocer la posición exacta del usuario en un momento concreto, únicamente es necesario implementar una serie de métodos, facilitados por la documentación de Google, así como solicitar los permisos necesarios si es la primera vez que el usuario accede a la funcionalidad de GPS. Una vez han sido concedidos, podemos conocer la latitud y longitud exactas, o bien, si en ese momento no se puede obtener, nos devuelve la última conocida. Una vez tenemos estos dos valores podemos realizar las peticiones a la API indicando la latitud y longitud deseadas, obteniendo el listado de centros próximos para cada usuario, independientemente de dónde se encuentre. Esta

tarea se realiza generando una URL que contiene tanto nuestra clave de API obtenida al comienzo del proyecto, como los atributos que permiten hacer una búsqueda precisa, incluyendo los valores devueltos por el sensor de GPS. Cabe mencionar, que todo el proceso se realiza dentro de una clase asíncrona para que sea ejecutado en segundo plano y acapare los recursos de procesamiento del teléfono, (*véase Anexo II, punto 5*).

Vayamos con el botón de acceso a Google Maps, el cual nos permite entrar de forma instantánea en esta aplicación y ver, sin necesidad de introducir ningún dato, la ubicación del dermatólogo en el que nos hemos interesado. Aunque pueda parecer una actividad bastante compleja de realizar, su desempeño es realmente sencillo porque se basa en el uso del ya mencionado Intent, al cual le añadimos como parámetro una URI con la latitud, longitud y el nombre del negocio. Como curiosidad, para el empleo de este tipo de recursos basados en servicios de Google, es necesario importar algunas librerías relacionadas con los Google Play Services, así como indicar el uso de estos en el documento de configuración "manifest.xml".

Tras los últimos avances me di cuenta de que para el usuario sería ideal conocer la última vez que actualizó el listado de dermatólogos, pues podría encontrarse en una ubicación diferente con motivo de un viaje, por ejemplo, y desconocer si esos lugares que está visualizando se corresponden a dicha posición o a la anterior. Para añadir esta utilidad hago uso de las SharedPreferences de Android, que permiten almacenar datos de tipo primitivo en el dispositivo de manera que esta información se mantiene a salvo, aunque hayamos cerrado la aplicación. Con ello, cuando realizamos la búsqueda de los centros, tomo la fecha y hora del sistema y la guardo, además de actualizar estos datos en la vista. Tras añadir esta utilidad, doy por concluido el desarrollo de las tres vistas principales de la aplicación, pero todavía sería necesario crear dos nuevas vistas que permitieran al usuario acceder a los detalles de una muestra dermatológica ya analizada por la red neuronal, así como, a partir de esta pantalla, poder adquirir más información acerca de la enfermedad identificada por nuestra red.

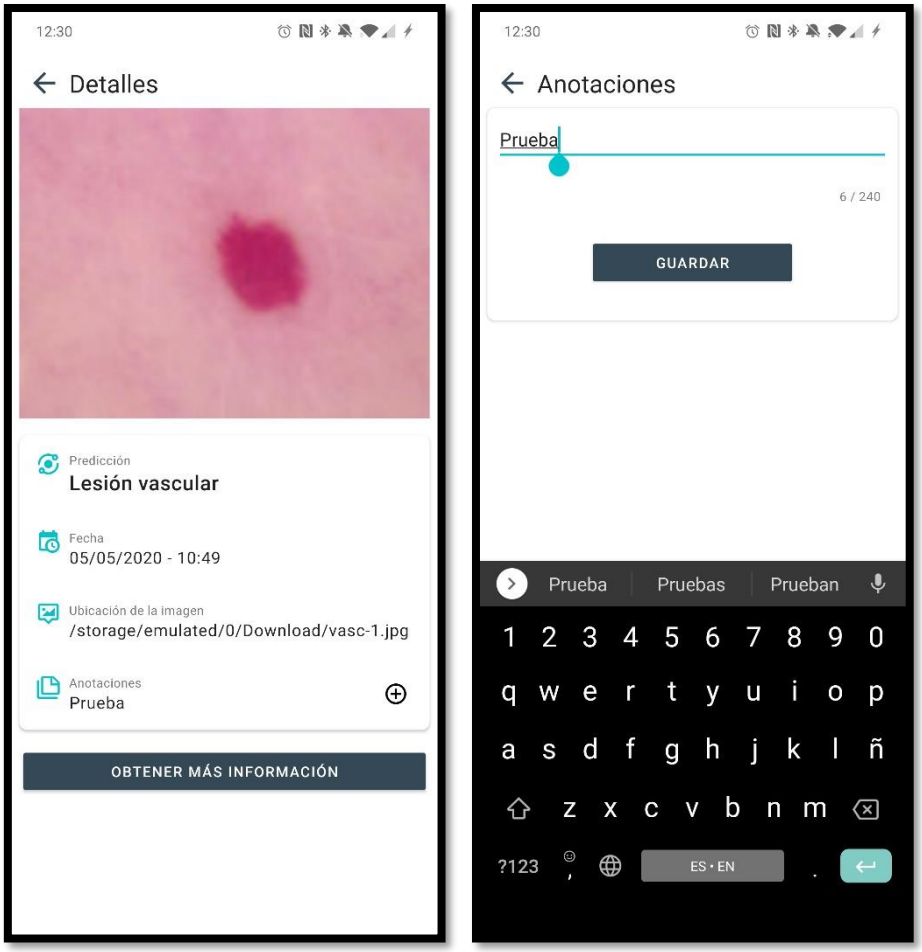


Ilustración 9. Vista de detalles

Tal y como se puede observar en la *Ilustración 9*, mostramos toda la información disponible de la muestra fotográfica aportada por el usuario y procesada por nuestro sistema de Inteligencia Artificial, aún sin desarrollar. La estética se mantiene fiel al formato en tarjeta y dentro de esta incluyo la predicción efectuada, la fecha y hora en que se realizó dicho procesamiento, así como la ubicación donde se encuentra almacenada la imagen protagonista, permitiendo saber si fue cargada desde el dispositivo o tomada desde la aplicación, pues en este caso, se almacena en una carpeta concreta denominada “SkinScanner”. Además, tras consultar entre familiares y amigos, llegamos a la conclusión de que resultaría de gran utilidad contar con un apartado destinado a las anotaciones que el usuario quisiera hacer como, por ejemplo, la parte del cuerpo a la que pertenece la mancha o el día que comenzó a tenerla, ya que no tiene por qué coincidir con el día en que se tomó la fotografía.

La implementación de esta vista fue realmente sencilla en comparación con las anteriores, existiendo una única complejidad relativa a la presentación de la imagen relacionada con el registro de la base de datos que el usuario quiere visualizar. Inicialmente realizaba la carga de esta directamente desde la memoria del dispositivo, pero el rendimiento de la aplicación decaía notablemente pues cuando realizábamos

sucesivos accesos a distintos registros los tiempos se incrementaban ligeramente empeorando el atractivo visual y la fluidez de la aplicación. Para solventar este inconveniente vuelvo a utilizar la librería Glide. Este recurso es empleado de igual manera a la hora de visualizar el historial, logrando unas transiciones rápidas entre las distintas vistas, al mismo tiempo que el desplazamiento vertical es extremadamente fluido. Además, aunque en nuestro caso Glide es empleada únicamente con imágenes, también es compatible con otros formatos como vídeos y GIFs, permitiendo dar diferentes efectos y características a los elementos mostrados.

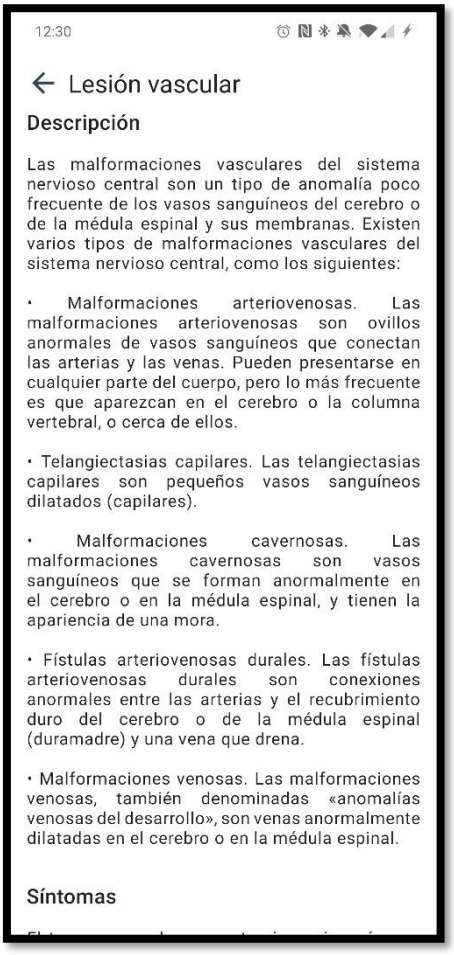


Ilustración 10. Vista de información adicional

Hemos llegado a la última vista de la aplicación (*Ilustración 10*), en la que el usuario podrá ver de forma inmediata información acerca de la enfermedad que la red le ha indicado que podría padecer. Esta información está almacenada en la propia aplicación de forma que siempre estará accesible sin necesidad de conexión a Internet. En este punto me encontré con un inconveniente importante pues, aunque el desarrollo de esta fase tuvo lugar sin ningún percance, la obtención de dicha información no pudo realizarse tal cual estaba prevista debido a la excepcional situación en la que nos encontrábamos por la COVID-19. Inicialmente queríamos visitar diferentes centros

dermatológicos para intentar recabar la máxima cantidad de datos acerca de las siete enfermedades diagnosticables de *SkinScanner,* pero, por desgracia, como causa del confinamiento establecido resultó imposible realizar esta tarea. El objetivo era dotar a la aplicación de información obtenida directamente de un profesional, así como obtener muestras fotográficas diagnosticadas en sus centros, para poder realizar pruebas en nuestra red neuronal.

A pesar de los inconvenientes, logré encontrar una web dedicada a la divulgación científica en la cual se recogían todas nuestras patologías, acompañadas de pruebas gráficas. Dicha web es propiedad de la Organización Mayo Clinic, una entidad internacional de gran prestigio y cuya información es redactada por expertos en la materia. Con ello, aunque me hubiera gustado seguir el plan inicial, obtuve una valiosa fuente de datos fiables, ideales para la visualización en nuestra aplicación. Además, almacené las imágenes disponibles para las posteriores pruebas de precisión de la red neuronal que debía crear.

Por último, me gustaría mencionar algunos detalles adicionales incluidos y que, bajo mi punto de vista, aportan mayor calidad al producto software elaborado. Por un lado, la aplicación es totalmente funcional en dos idiomas, español e inglés, para así poder llegar a un público mucho más amplio. Por otro lado, durante mi formación como ingeniero, siempre se nos ha recalcado la importancia de crear software responsivo, capaz de adaptarse al dispositivo y las preferencias del usuario final, de manera que *SkinScanner* es una aplicación diseñada para usar tanto en modo vertical como horizontal, modificando la distribución y tamaño de los elementos según convenga.

Red neuronal

He finalizado el desarrollo de la aplicación, pero continúo lejos de alcanzar el final del proyecto pues ahora debo crear una red neuronal que sea capaz de distinguir suficientemente bien las siete clases de enfermedades dermatológicas que nos aporta el dataset obtenido de la web Kaggle. Esta fase del trabajo resultó infinitamente más compleja de lo que inicialmente había previsto, requiriendo sucesivos cambios en la planificación en cuanto a la infraestructura a emplear para el despliegue, así como el propio modelo de la red. Por ello, voy a describir todos los pasos realizados, incluyendo aquellos que resultaron no tener éxito para la creación del modelo finalmente implementado en el proyecto.

Comencé creando un proyecto en PyCharm en el cual alojaría todos los ficheros Python necesarios tanto para la creación de la red neuronal, como el servidor donde se cargará. Pretendo diseñar un modelo totalmente propio para observar los resultados que obtengo y decidir si puedo lograr un modelo competente, o bien, entrenar con un modelo ya creado como los conocidos VGG o DenseNet, que cuentan con gran número de capas y, por tanto, más potencia para analizar imágenes tan peculiares y aleatorias como son las muestras cutáneas.

El proceso de entrenamiento requiere primeramente la obtención y procesamiento de los datos a emplear. En este punto cuento con las imágenes de todas las categorías mezcladas entre sí en dos carpetas, junto con un archivo CSV donde se encuentra el identificador de cada imagen, así como la categoría a la que corresponde, entre otros detalles. Para poder ser empleadas necesito contar con dos carpetas, una destinada a la fase de entrenamiento y otra a la fase de validación, donde en el interior de cada una haya siete carpetas representando cada una de las patologías. Carecía de experiencia para efectuar esta reorganización de los archivos por lo que tuve que investigar por la red hasta encontrar pequeñas porciones de código que me permitieron crear mi propio archivo Python clasificador de muestras.

El funcionamiento consiste en leer el archivo CSV y agrupar los datos en función del ID de la lesión para, posteriormente, identificar imágenes duplicadas y marcarlas en una nueva columna de forma que sean ignoradas en el momento de repartirlas entre las carpetas de entrenamiento y validación. A continuación, realizamos el reparto de muestras entre dichas carpetas, destinando un 17% de las fotografías a la carpeta de validación y el 83% restante a la de entrenamiento. Una vez tenemos asignado el destino de cada imagen procedemos a insertarlas en las carpetas que corresponde, realizando dos veces dicho proceso pues recordemos que el total de imágenes iniciales estaba repartido entre dos carpetas.

Aunque ya tenemos nuestras imágenes listas para ser utilizadas, podemos aplicarles diferentes transformaciones para ampliar el tamaño del dataset y contemplar nuevas situaciones como rotaciones, brillos, volteos, etcétera. Estas transformaciones son asignadas a una variable que se aplica a continuación, para generar dos nuevas entidades que representan el conjunto de entrenamiento y validación. En este punto definimos el tamaño de entrada de las fotografías, las cuales decido que serán de 75x100 píxeles. De esta manera, el modelo neuronal tendrá en su primera capa una entrada de 75x100x3 debido a que una imagen tiene tres canales de color, conocidos como RGB.

Tras superar este obstáculo puedo centrarme en elaborar mi modelo secuencial, el cual, en un primer momento, contará con varios bloques idénticos en cuanto a las capas contenidas en ellos, consistiendo en una capa convolucional 2D y una MaxPooling, además de una Dropout, repitiendo tres veces esta estructura. Además, para el bloque final añado una capa Flatten, seguida de dos capas Dense con un nuevo Dropout entremedias. Con las capas convolucionales buscamos que la red filtre las imágenes en función de determinadas características de esta que producen que ciertos píxeles tengan valores más altos que otros permitiendo, por ejemplo, reconocer los bordes de las manchas, pues es capaz de resaltar los saltos de gradiente que reflejan los límites de cada lesión. Las capas MaxPooling son útiles para reducir progresivamente el tamaño de las imágenes porque van analizándola mediante una ventana de píxeles escogiendo aquel que tenga el valor más alto, de manera que actúa como representante del conjunto de píxeles contenido en la ventana de análisis. Uno de los problemas más comunes en el desarrollo de modelos neuronales es el sobreajuste de la red, creando un software que se adapta de manera excesiva a un tipo de imágenes de forma que cuando

es sometido a una imagen ligeramente diferente es incapaz de reconocerla. Para paliar este problema aparecen las capas Dropout que permiten desactivar un porcentaje de las neuronas de forma que la red tenga que hacer un mayor esfuerzo durante el entrenamiento y conseguir así, reducir este sobreajuste.

Al trabajar con imágenes manejamos matrices de dos dimensiones, de manera que para la red neuronal es fundamental convertir esta matriz en un array unidimensional sobre el que poder emitir la predicción. Esta transformación es realizada automáticamente por la capa Flatten, para pasar a las capas Dense, las cuales se encargan, finalmente, de aplicar la función de activación.

Cuando creamos cada capa convolucional podemos elegir el número de nodos que tendrá y añadir otras funcionalidades adicionales. Por norma general suelen emplearse potencias de dos, de manera que en cada capa nueva se dobla el número de nodos. Además, estas capas junto con las MaxPooling nos permiten establecer el tamaño de la ventana que analizará los píxeles de la imagen, escogiendo para las del primer tipo un tamaño de 3x3 y para las segundas, 2x2. Los porcentajes de Dropout han sido añadidos de manera aleatoria, incrementado ligeramente el valor a medida que profundizamos en el modelo. Por último, la elección de las funciones de activación depende de las necesidades de cada proyecto. En mi caso para la capa Dense oculta tomé una unidad de activación lineal rectificada, más comúnmente conocida como ReLU, eligiendo para la capa final una función Softmax. La primera de estas es ampliamente utilizada como función rectificadora en capas ocultas y su lógica puede resumirse en que para todo valor positivo devuelve dicho valor, pero para cualquier valor negativo devuelve cero (*Ilustración 11*). La función Softmax genera un vector que nos indica, a partir de las siete clases posibles, a cuál pertenece la imagen analizada.

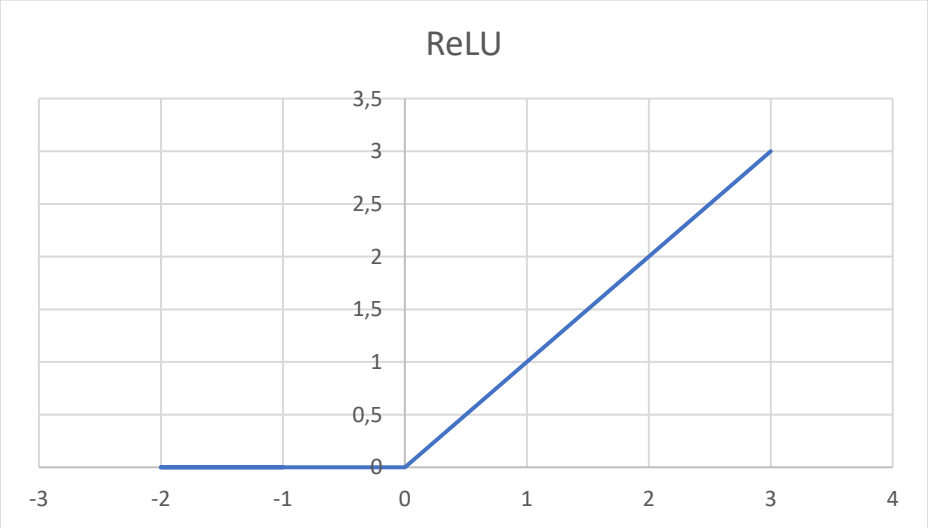


Ilustración 11. Ejemplo función ReLU

Una vez construido mi primer modelo debo crear un optimizador, que será el encargado de comparar los valores estimados con los reales para calcular el error y

ajustar progresivamente los parámetros de la red neuronal. Decido crear un optimizador Adam, estableciendo sus parámetros según valores típicamente usados en este tipo de modelos, destacando la tasa de aprendizaje con un valor de 0.001 y épsilon con 1e-08. Del mismo modo, creo un reductor de la tasa de aprendizaje, con el objetivo de que a medida que se entrena, se reduzca progresivamente dicha tasa para mejorar la obtención del mínimo de la función correspondiente.

Llegado a este punto, puedo comenzar el entrenamiento de la red, para lo cual defino el número de epochs deseados y asigno todos los parámetros creados anteriormente. Por desgracia, tras haber finalizado el proceso, los resultados obtenidos no son nada positivos y debo comenzar a investigar las opciones disponibles para mejorar esta situación. La precisión alcanzada tanto para el dataset de entrenamiento como de validación es demasiado baja, con presencia de overfitting, del cual conoceremos las razones más adelante. Decido duplicar el número de capas convolutivas y volver a realizar el entrenamiento completo, pero obtengo valores muy similares. Tras indagar por internet descubro un nuevo parámetro que puede ser aplicado en las capas convolutivas y que recibe el nombre de regularizador de peso. Este nuevo elemento permite aplicar penalizaciones a los parámetros de manera que se suman en la función de pérdida para lograr una mejor optimización de la red. En mi caso decido aplicarlo al núcleo de las capas con las que cuento actualmente.

Sin embargo, tras varios intentos, los resultados obtenidos eran muy desalentadores. A pesar de que había mejorado ligeramente los valores de precisión, en la partición de entrenamiento eran notablemente superiores a las de la validación, lo cual se traducía en un claro overfitting. Además, a pesar de lograr mejorar los valores de precisión, estos seguían sin ser lo suficientemente altos para estar satisfecho, de manera que me encontraba ante el primer gran problema: solventar el overfitting al mismo tiempo que debía mejorar la precisión. En primer lugar, consideré reducir el número de capas de manera que la red no se ajustara en exceso a cada imagen del entrenamiento, pero esto a su vez provocaría que la precisión se redujera aún más y probablemente la reducción del overfitting fuera prácticamente despreciable.

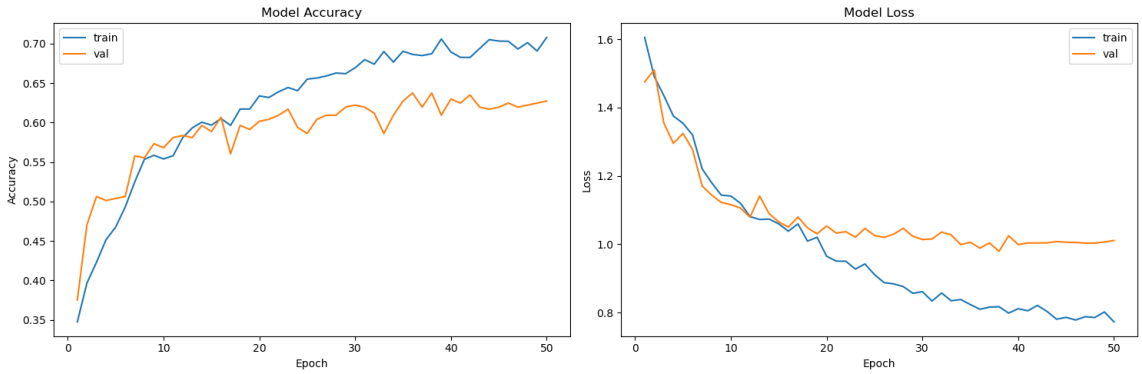


Ilustración 12. Precisión y pérdidas

Como podemos observar en la *Ilustración 12*, los resultados gráficos eran una clara confirmación del overfitting existente y, a pesar de evitar a toda costa este camino, fue necesario revisar el dataset para comprobar si estaba desbalanceado y esto producía el sobreajuste de la red neuronal. Este fenómeno tiene una explicación muy sencilla pues, al existir muchas más muestras de un tipo de enfermedad que de otra, la red se entrena y obtiene unos pesos en sus neuronas que permiten reconocer perfectamente la clase predominante del dataset, pero es incapaz de reconocer de manera correcta el resto de las clases de este. Además, al haber gran cantidad de un tipo y reconocerlas correctamente, la precisión tiene un valor elevado, pero no es realista. Si bien comprender el problema era cuestión de minutos, solventarlo llevaría bastante más trabajo. Tras revisar el número de imágenes que poseía de cada clase me percaté de que el 67% de un total de diez mil imágenes, pertenecían a la clase "nevus melanocítico". Había encontrado el error y lo pude confirmar realizando predicciones con el último modelo entrenado obteniendo que en la gran mayoría de imágenes la red siempre predecía que se trataba de la clase predominante.

Ante este inesperado contratiempo, fue necesario examinar el registro CSV de imágenes, para localizar todas aquellas etiquetadas como "nevus melanocítico" y realizar una limpieza aleatoria de estas para que el dataset estuviera correctamente balanceado. El procedimiento ideal habría sido revisar cada imagen para eliminar aquellas que fueran más problemáticas para la red tanto por la forma de la mancha, color o resolución; pero debido a la gran cantidad de muestras y a mi falta de conocimientos dermatológicos, era un procedimiento completamente inviable y, por ello, opté por realizar el cribado de forma aleatoria. Durante este tiempo revisé el resto de las clases siguiendo el mismo criterio para lograr un repertorio de imágenes donde todas contaran con un número similar de fotografías.

El esfuerzo realizado para corregir el dataset permitió lograr resultados esperanzadores gracias a la reducción del overfitting, aunque seguía sin lograr valores adecuados de precisión y pérdidas. En este punto, decidí que, debido a la naturaleza de las manchas cutáneas, las cuales son de formas muy diversas y con un gran parecido entre todas, empleando modelos como ResNet o DenseNet, con un número de capas infinitamente más grande, podría lograr una mejor precisión. Si bien, los nuevos entrenamientos para estos modelos se demoraban una o varias horas, provocando que realizar modificaciones en los hiperparámetros y en la elección del modelo, se convirtiera en una ardua tarea. Si quería usar modelos muchos más potentes requería también de tiempo, el cual era limitado a causa de la compaginación de este trabajo con las asignaturas.

Llegado a este punto, comienza una nueva etapa de investigación, donde el objetivo principal es encontrar herramientas que me permitan entrenar grandes modelos en menor tiempo para así poder realizar mayor cantidad de pruebas y llegar a un modelo final que cumpla las expectativas. El primer cambio surgió con el empleo de Google Colaboratory en vez de PyCharm como entorno de programación para la creación y entrenamiento del modelo. Tras realizar varias comparaciones observé que

la opción de Google era mucho más eficiente para este tipo de tareas que mi actual ordenador. En segundo lugar, tras dedicar horas a la búsqueda de utilidades que facilitaran dichos entrenamientos, encontré una herramienta denominada Fastai, que permitía, no solo emplear los modelos disponibles en la red y mencionados anteriormente sino, además, realizar un entrenamiento progresivo y donde algunos hiperparámetros son ajustados por la propia librería.

Sin duda, la funcionalidad más útil para mi propósito era la capacidad de poder entrenar el modelo en fases, de manera que cuando procedemos a entrenarlo no necesitamos esperar a que termine de realizar todas las iteraciones para detectar si los cambios aplicados son beneficiosos o perjudiciales, sino que podemos indicarle que queremos realizar diez iteraciones, observar los resultados, quizás modificar la tasa de aprendizaje y, seguidamente, realizar otras veinte para volver a verificar los valores devueltos, (*véase Anexo III, punto 5*). Dicha tasa de aprendizaje determina el tamaño del paso en cada iteración mientras se mueve hacia un mínimo de la función de pérdida y Fastai nos ofrece un sencillo método que a partir del modelo y dataset escogidos, calcula y muestra gráficamente los mínimos relativos y absolutos de la función para poder escoger el valor óptimo. Podemos ver un ejemplo en la siguiente *Ilustración.*

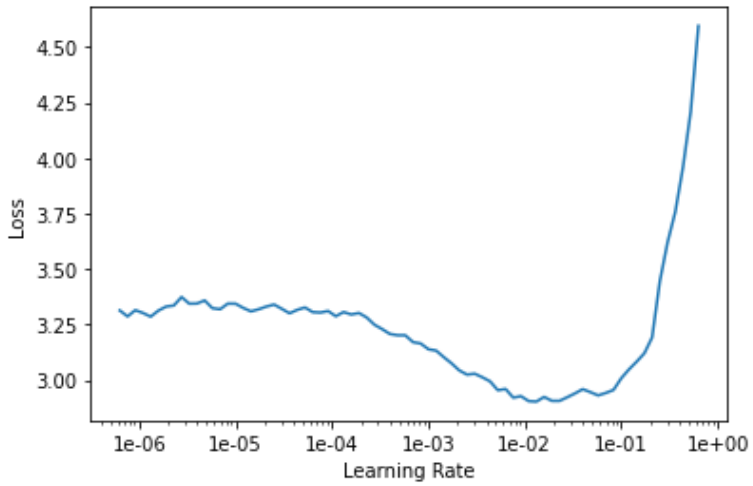


Ilustración 13. Tasa de aprendizaje

Una vez explicadas las razones por las que elegí Fastai como mejor opción para el modelo neuronal, procedo a describir el proceso de entrenamiento que me ha permitido lograr una red lo suficientemente precisa para mi proyecto. Ha sido un proceso completamente iterativo, a base de pruebas y errores.

Para acceder al dataset realicé la subida de este a la nube Google Drive, ya que gracias a dos simples líneas de código puedo acceder a todos los archivos almacenados en mi cuenta. Una vez realizada la vinculación accedo al archivo CSV y clasifico las imágenes en función de su categoría. Además, Fastai también ofrece la posibilidad de generar nuevas imágenes aplicando diferentes modificaciones a las imágenes almacenadas, aprovechando esta funcionalidad para aplicar volteos horizontales y

verticales con la idea de que la red, en el futuro, se adaptase a las imágenes independientemente de la orientación del dispositivo. Con todo esto, podemos generar una variable contenedora de todas las muestras, en la cual especificamos la localización de los archivos, el tamaño de reescalado para las imágenes y otros detalles como el porcentaje del total destinado a la partición para validación, (*véase Anexo III, punto 4*).

A continuación, elijo el modelo que deseo entrenar, de manera que Fastai lo descarga de forma autónoma. Durante varias semanas pruebo prácticamente la totalidad de modelos disponibles y, finalmente, concluyo que los modelos DenseNet son los que mejor funcionan con mi dataset, concretamente el modelo DenseNet121 fue el elegido para el modelo de producción. Estos modelos reciben el nombre de DenseNet debido a que son redes neuronales densas, ejemplo en la *Ilustración 14*, donde cada capa conecta con las sucesivas en vez de conectar únicamente con la siguiente, como ocurre en los modelos tradicionales. Esta característica aporta una serie de ventajas como la propagación de características de cada imagen y, por otro lado, su estructura es muy eficiente, requiriendo menos tiempo para los entrenamientos.

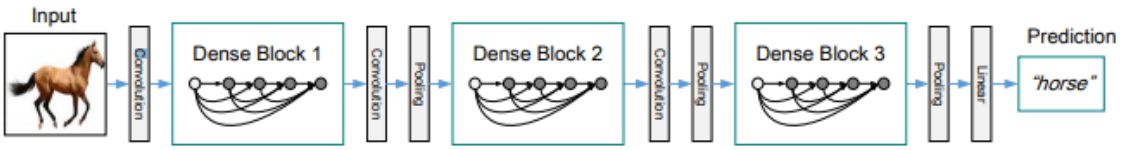


Ilustración 14. Red convolucional densa

Una vez tenemos cargado el modelo deseado podemos crear nuestro "aprendiz", un parámetro con el cual realizaremos todo el proceso de entrenamiento, así como las predicciones y exportación del modelo final. Su creación es muy sencilla y solo requiere especificar los datos de entrenamiento, es decir, el dataset; el modelo que deseamos emplear, en nuestro caso DenseNet121; las métricas que queremos que nos devuelva para analizar si los resultados son positivos o no; y, por último, podemos indicar que nos muestre un gráfico para visualizar rápidamente la evolución de la precisión en el conjunto de entrenamiento y de validación, (*véase Anexo III, punto 6*). El siguiente paso es definir la tasa de aprendizaje a aplicar en las iteraciones, dato el cual normalmente es escogido de forma arbitraria en función de valores tradicionales, pero, como vimos anteriormente, a partir de los datos y el modelo indicados, el aprendiz nos genera un gráfico que nos permite escoger una buena tasa de aprendizaje.

Una vez he escogido el mejor valor para el aprendizaje, indico al aprendiz el número de epochs que quiero que realice, junto con dicho valor. El resultado será una tabla donde podremos ver las pérdidas de entrenamiento y validación, además de la ratio de error y la precisión alcanzada. Junto a esta tabla veremos un gráfico donde comparar las tasas de error visualmente. El modelo final fue entrenado en varias etapas, comenzando con once iteraciones y una tasa de aprendizaje de 1e-02. Tras este primer ciclo de trabajo el modelo había logrado alcanzar una precisión de 0.7175, un valor nada

desdeñable. El siguiente paso fue calcular nuevamente el valor ideal para dicha tasa y realizar un nuevo ciclo de iteraciones, pero las pérdidas de entrenamiento y validación apenas disminuían y la precisión se mantenía estática, lo cual me indicaba que el modelo tenía dificultades para mejorar por muchas más iteraciones que realizara. En la *Ilustración 15* podemos ver la disminución del error y cómo a medida que avanzamos su reducción es cada vez menor.

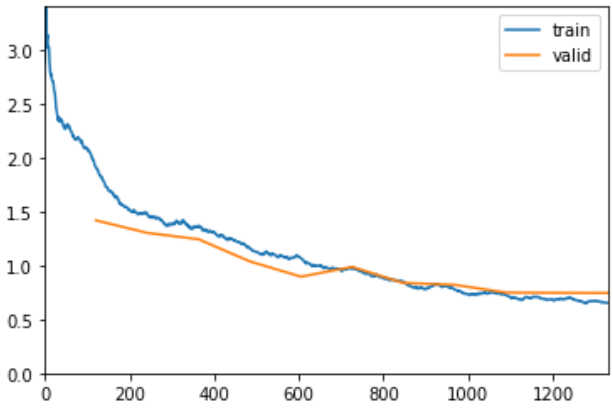


Ilustración 15. Gráfico con las tasas de error

Además de las dificultades para entrenar la red, un día me encontré con que la librería Fastai había dejado de funcionar en Google Colaboratory. Este fenómeno supuso un retraso adicional en la planificación pues el error no había sido causado por mi desarrollo y, por ello, encontrar la causa iba a ser una tarea compleja. Junto a la frustración de no poder seguir avanzando con el proyecto, se sumaba la escasa información que había en Internet acerca de dicho error. Finalmente, tras varios días sin poder avanzar en el modelo, conseguí encontrar el origen del problema, el cual vino dado por una actualización de versiones de librerías requeridas por Fastai para funcionar, concretamente de Torch y Torchvision. Esta actualización provocaba errores de compatibilidad con nuestra librería de inteligencia artificial y, dado que había sido una actualización reciente, Fastai no había lanzado hasta ese momento ningún parche para solventar el inconveniente. La solución consistió en desinstalar de Google Colaboratory las versiones problemáticas e instalar versiones más antiguas.

Tras haber solventado un bache desmoralizante, decidí que en vez de conformarme con los resultados obtenidos hasta el momento y dar por concluido el entrenamiento, buscaría información de cómo poder lograr que el modelo consiguiera volver a aprender. En la propia web oficial de Fastai, donde disponen de un curso completo acerca del uso de su herramienta, se recomendaba que ante estas situaciones se probara a retomar el dataset completo, pero duplicando el tamaño de entrada de las imágenes, de manera que, si inicialmente eran de 112x112 píxeles, en esta nueva etapa fueran de 224x224 píxeles, manteniendo el mismo batch size siempre que el ordenador fuera capaz de soportarlo. Una vez realizado este paso repetí el proceso, buscando la

tasa de aprendizaje más conveniente y realizando un entrenamiento progresivo. Realicé una nueva etapa de diez iteraciones con un learning rate de 1e-05, obteniendo un 74.53% de precisión aproximadamente. En vista de los resultados y que los valores de pérdida continuaban disminuyendo, llevé a cabo tres nuevos periodos de entrenamiento, cada uno de ellos con su correspondiente búsqueda de la mejor tasa de aprendizaje.

Finalmente, he logrado obtener un modelo con una precisión del 81.34% aproximadamente. Dado que las muestras dermatológicas son imágenes con patrones muy aleatorios, incluso dentro de la misma enfermedad, y que un profesional humano tiene una probabilidad de acierto mediante la simple observación muy inferior, considero que he alcanzado un resultado suficientemente positivo para el modelo y su implementación en la aplicación *SkinScanner*.

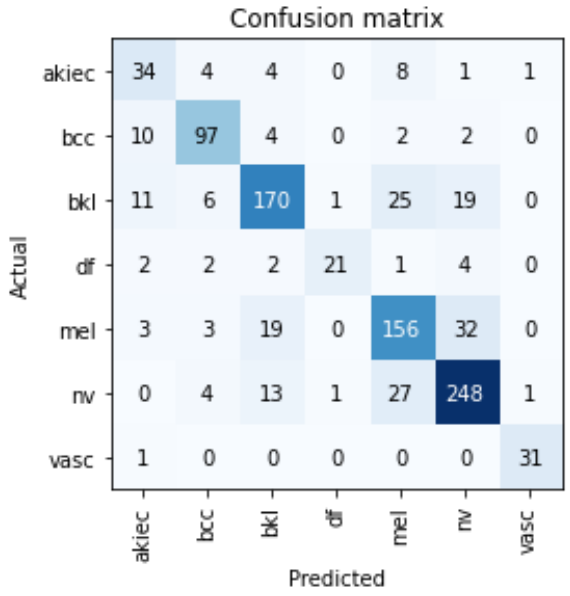


Ilustración 16. Matriz de confusión

En la *Ilustración 16*, podemos observar una matriz que nos muestra el comportamiento de nuestra red neuronal, donde en la diagonal que recorre desde la esquina superior izquierda hasta la esquina inferior derecha observamos el número de predicciones correctas que ha sido capaz de realizar el sistema y, fuera de esta diagonal, todas las predicciones erróneas, pudiendo comprender cuáles son las enfermedades que generan mayores dificultades para ser clasificadas, dada la gran semejanza visual de las manchas cutáneas asociadas. Podemos ver que el modelo tiene dificultades para distinguir entre un melanoma, un nevus melanocítico y una queratosis seborreica. Este hecho es causado directamente por la extrema similitud que existe entre estos tres tipos de patologías, donde sus diferencias visuales son casi inapreciables.

Máquina virtual

He logrado desarrollar una aplicación móvil y una red neuronal, ambas totalmente preparadas para ser integradas en una única infraestructura, para lo cual será necesario crear una máquina virtual que funcione como servidor donde cargar el modelo y procesar las peticiones de análisis que reciba desde la aplicación. Este constituye el último eslabón del desarrollo y, a pesar de inicialmente haber planeado la configuración ideal para montar todo el sistema, ocurrieron una serie de infortunios que propiciaron nuevos retrasos en el despliegue del software.

La propuesta principal descrita en el documento TFT-01 mencionaba el empleo de los servicios de Amazon denominados Amazon Web Services (AWS), para la creación de una máquina virtual Linux donde establecer nuestro servidor desarrollado en Python y almacenar el modelo neuronal creado. Este planteamiento era perfectamente válido en los inicios pues el modelo a desarrollar se apoyaría en Keras y Tensorflow, ambas herramientas fácilmente instalables en entornos Linux de bajas prestaciones. Sin embargo, al usar la librería Fastai para la creación del modelo final, todo el plan se complicó de manera inesperada, debido a que dicha librería requería de mayor capacidad de almacenamiento, la cual no era posible debido a las limitaciones de la versión gratuita para estudiantes del servidor empleado.

Decidí probar suerte con la plataforma Google Cloud, en la cual podía crear la máquina virtual con mayores libertades y, además, controlar el estado de esta desde una aplicación móvil, pudiendo encenderla y apagarla cuando necesitase de manera que la cuota de prueba no se consumiera rápidamente. Únicamente debí crear un proyecto en la plataforma y a partir de este activar aquellas funciones que necesitase, siendo en este caso la herramienta Compute Engine y Storage. A través de la primera pude crear una instancia de una máquina virtual Linux, a la que puedo acceder desde la misma página de Google, mediante una terminal flotante. La segunda herramienta es empleada para la transferencia de archivos desde nuestro ordenador a la plataforma y de esta a la máquina virtual.

A pesar de estas bondades, seguía contando con espacio insuficiente para la instalación de Fastai y surgían excepciones que no había visto hasta el momento. Tras investigar durante varios días, logré encontrar una pequeña esperanza en una herramienta denominada Anaconda, la cual según mencionaban en algunos foros, permitía realizar una instalación completa de Python y Fastai en las máquinas virtuales de Google. Tras seguir la documentación oficial de Anaconda logré instalar todas las librerías y herramientas necesarias para poder ejecutar mi servidor en la máquina. Creé un servicio (archivo con extensión ".service") encargado de ejecutar el servidor cada vez que se arrancase el sistema, de forma que siempre estuviera disponible. Además, habilité la recepción de peticiones HTTP/S y asigné una dirección IP estática a la máquina, para que no hubiera errores de direccionamiento en las futuras peticiones. Una vez configurada la máquina correctamente, me dispuse a completar el código software de la aplicación Android, para llamar a este nuevo servidor y obtener de él un código de siete posibles, donde cada uno representa una enfermedad dermatológica.

Dicho código es usado como identificador en la aplicación para la muestra de información acerca de la patología.

Con un servidor totalmente operativo y una aplicación lista para su uso, finalizo el desarrollo de mi proyecto. Sin duda, un periodo mucho más complejo de lo que inicialmente había imaginado, aunque al mismo tiempo, muy gratificante por haber conseguido superar cada uno de los obstáculos que se presentaban y finalmente tener un producto software completo y que satisface los requisitos de usuario establecidos al comienzo de esta aventura.

Temporalización

En el momento de presentar la documentación necesaria para la aprobación mi propuesta de TFT, dividí el proyecto en cuatro etapas. La primera de ellas abarcaría todo el proceso de estudio e investigación acerca de los diferentes recursos y herramientas que necesitaría en un futuro para alcanzar los objetivos planteados. La estimación realizada fue de ochenta horas y, a pesar de tener que realizar investigaciones y pruebas más extensas de lo que imaginaba, avancé a buen ritmo pudiendo completar esta fase dentro del periodo previsto.

En segundo lugar, tocaba comenzar el desarrollo del producto software, incluyendo tanto la aplicación móvil como la red neuronal y el servidor e infraestructura necesarios para su integración completa. Tal y como he explicado durante el apartado anterior, esta fase requirió enfrentarme a multitud de imprevistos y dificultades que se tradujeron en constantes retrasos y golpes desmoralizantes. Inicialmente había previsto una duración de doscientas horas, considerando que era un tiempo más que suficiente para realizar todo el proyecto, pero, tras concluir este periodo, estoy prácticamente seguro de haber empleado la totalidad del tiempo estimado e, incluso, algunas horas adicionales.

Por último, nos encontramos con dos etapas más simples como son la evaluación del producto y la elaboración de la documentación, etapa en la que me encuentro en estos momentos. Cada una estaba estimada en diez horas respectivamente y, bajo mi punto de vista, han sido unas previsiones acertadas pues las pruebas efectuadas al software no han supuesto inconvenientes y la documentación, aún en desarrollo, avanza al ritmo esperado.

Análisis	
80 horas	Redes neuronales y su implementación en servicios web
	Google Places API, envío de peticiones y recepción de respuestas
	Herramientas para la creación de un software fotográfico para la aplicación
	Creación de máquina virtual remota
	Diseño Material Design, patrones a seguir
Diseño y desarrollo	
200 horas	Creación, entrenamiento y refinamiento de red neuronal
	Creación de servidor remoto mediante máquina virtual
	Creación y despliegue de servicio Linux
	Configuración y empleo de Google Places API
	Desarrollo “Backend” y “Frontend” de app nativa
Evaluación y pruebas	
10 horas	Comprobar resultados de la red neuronal
	Verificar requisitos y correcto funcionamiento de la aplicación en diferentes situaciones
Documentación	
10 horas	Elaboración de memoria
	Revisión con tutor

Prueba y mantenimiento

Desde el comienzo tenía claro que el producto que creara debía satisfacer una serie de requisitos entre los cuales se encontraba la ausencia de errores de funcionamiento. Por esta razón, dediqué bastantes horas a la búsqueda de excepciones que pudieran generarse en la aplicación, para manejar todas y cada una de ellas de manera que nunca se produjera un cierre forzoso de la misma o un comportamiento brusco que empeorase la experiencia de usuario. Además, al ser una aplicación integrada con otros servicios como Places API y mi servidor externo, debí analizar las situaciones que podían darse durante el uso cotidiano.

Al ser una aplicación móvil con la que realizar predicciones sobre imágenes, podemos encontrarnos ante dos escenarios distintos. Estos pueden darse si el dispositivo en el momento de analizar la imagen no tiene conexión a internet y, por tanto, no puede conectar con el servidor responsable del proceso, o bien, el servidor en sí no se encuentra operativo en ese momento. Ante estas situaciones decidí crear una clase encargada de revisar el estado de las conexiones necesarias y, además, establecí un periodo de tiempo máximo en el que la aplicación esperaría por la respuesta del servidor, de manera que, en caso de no recibir respuesta alguna, se muestra una alerta informando del error producido, (*véase Anexo II, punto 7*).

Del mismo modo que para el servidor, antes de realizar una actualización de los dermatólogos próximos a la ubicación del usuario, realizo la misma comprobación de red para asegurar que el dispositivo pueda conectar con la API de Google. Así mismo, como expliqué durante la etapa de desarrollo del proyecto, el uso de esta API requirió realizar bastantes pruebas pues los resultados obtenidos inicialmente no eran del todo correctos, debido a la clasificación errónea de los establecimientos.

Para el uso de las funcionalidades de internet y GPS es necesario contar con el permiso del usuario, de manera que otras de las pruebas efectuadas consistían en intentar esquivar la concesión de dichas autorizaciones. Esta tarea resultó más compleja de lo que en un primer momento pensaba, pues la codificación del proceso de verificación era bastante tediosa, aunque, gracias a la modularización del código, pude desarrollar unos métodos sencillos que antes de permitir acceder a la cámara, galería o GPS, comprobasen si habían sido concedidos los permisos necesarios y, en caso contrario, volver a solicitarlos.

A partir de este punto procederé a describir el proceso de evaluación del servidor y la red neuronal cargada en este. El primero fue probado en dos etapas, comenzando con pruebas ejecutadas a nivel local en el propio ordenador donde estaba siendo desarrollado, realizando las llamadas a este a través de la consola. Esta metodología fue aplicada debido a las enormes dificultades para ejecutar en una máquina virtual remota el servicio, como ya vimos. Más adelante, cuando ya había logrado disponer de la máquina virtual en Google Cloud, realicé nuevas pruebas con llamadas idénticas, pero destinadas a la IP de esta. En este punto no valoraba las predicciones devueltas sino la eficiencia del sistema y los tiempos de espera. Todas las pruebas realizadas fueron un éxito, tanto a nivel local como en el host remoto.

El proceso de prueba de la inteligencia artificial que estaba desarrollando fue un proceso iterativo, marcado por las sucesivas versiones de los modelos que iba creando, donde para concluir si estaba realizando avances o no, empleaba no solo los resultados numéricos y gráficos, sino también las predicciones efectuadas sobre imágenes obtenidas de la red, provenientes de fuentes fiables en el campo de la medicina, (*véase Anexo III, punto 7*). Hubiera sido ideal contar con un repertorio fotográfico aportado por un dermatólogo, pero, como ya comenté, la situación excepcional en la que nos encontramos hizo imposible esta opción.

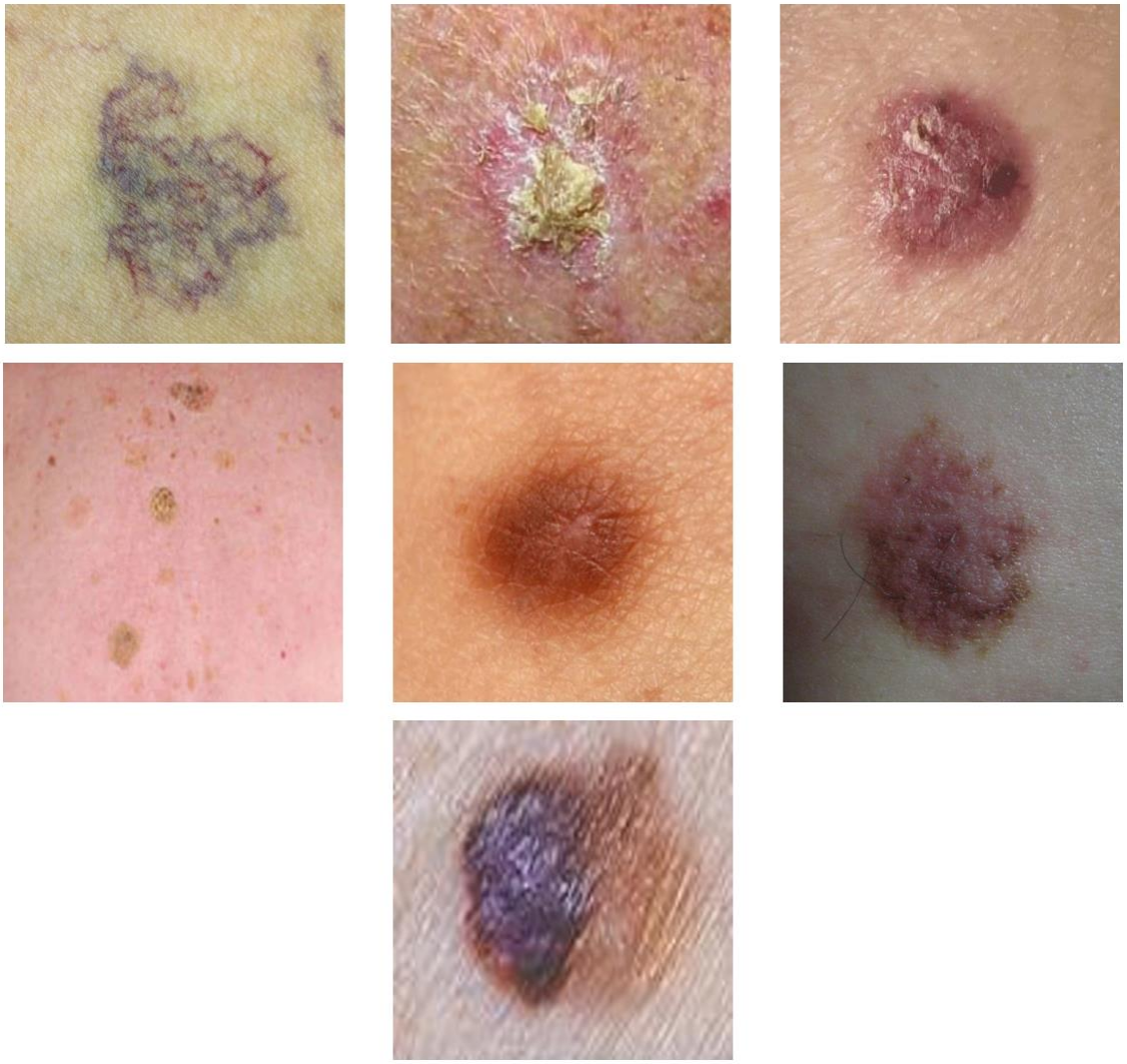


Ilustración 17. Imágenes de pruebas.

En la *Ilustración 17*, podemos visualizar algunas de las imágenes empleadas para validar el funcionamiento del modelo en desarrollo. Cada imagen se corresponde con una de las siete categorías que es capaz de clasificar la red y se puede apreciar la similitud entre algunas de ellas, así como las dificultades generadas por las diferentes calidades de imagen, enfoques e iluminación. Todas estas características perjudican el

rendimiento del modelo, el cual a pesar de ser entrenado correctamente nunca será capaz de lograr precisiones demasiado altas. A su vez, aunque para la muestra en este documento todas las imágenes han sido adaptadas al mismo tamaño, cada una tiene unas dimensiones propias, con lo cual, al realizar el escalado para su análisis y procesamiento, sufren pequeñas deformaciones que complican aún más estas tareas.

Tras toda la experiencia adquirida en estos meses de trabajo, considero que el desbalanceo inicial del dataset ha supuesto un claro problema pues, al realizar el equilibrado de cada categoría, el total de imágenes se redujo notablemente de manera que la red tiene mayores dificultades para aprender las características de cada tipo de enfermedad dermatológica. Además, la calidad de muchas muestras es bastante mediocre e incluso aparecen marcas en la piel que no ayudan en absoluto al proceso de aprendizaje. Indudablemente, un dataset mejor elaborado sería el aliciente perfecto para poder mejorar la red neuronal y alcanzar precisiones aún mejores.

Análisis de costos

SkinScanner es un proyecto caracterizado por la integración con otros servicios como son Places API de Google y el servidor remoto alojado en una máquina virtual de Google Cloud Platform. En el primer caso, para un proyecto de TFT es más que suficiente emplear el servicio de forma gratuita con las limitaciones que ello implica, pero si deseáramos poner en producción nuestro producto para que estuviera disponible en las tiendas de aplicaciones como Google Play y Apple Store, debemos tener en cuenta el costo que puede suponer disponer de esta API en nuestra aplicación. Google ofrece una determinada cantidad de crédito gratuito, pero una vez este es consumido se empieza a facturar en función de la cantidad de miles de solicitudes que reciba el servicio. Por ejemplo, en el caso de mi localizador de centros dermatológicos cercanos, cada mil peticiones se genera un cobro de diecisiete dólares, que en caso de lograr tener éxito en el mercado podría suponer un gasto mensual nada despreciable.

Por otro lado, actualmente el servidor se encuentra alojado en una instancia Linux proporcionada por Google de forma completamente gratuita, pero esto conlleva la limitación de recursos del sistema, haciendo inviable mantener esta infraestructura para una aplicación disponible para todas las personas del planeta. En este caso, los costos indicados por la compañía dependen de varios factores como el tiempo de uso, la cantidad de memoria requerida y los recursos para procesamiento gráfico que se necesite. Por ejemplo, una máquina con treinta y dos CPUs virtuales y ciento cuatro gigabytes de almacenamiento, suponen un gasto mensual aproximado de 968 dólares. Como vemos, un precio bastante elevado y con características que no terminan de ser idóneas para el tipo de infraestructura que se necesita, ya que requerimos mayor espacio.

Sin duda alguna, la mejor opción sería montar un servidor propio, con grandes capacidades de almacenamiento que nos permitan almacenar el repositorio de imágenes, así como añadir las nuevas que vayan tomando los usuarios, siempre con su previo consentimiento, para aumentar el tamaño y la calidad del dataset y, con ello, lograr mejorar la red neuronal. Pero claro, incorporar nuevas muestras al repertorio supone que debe existir una persona capaz de analizar cada imagen recibida y determinar a qué enfermedad se corresponde. La contratación de un experto en este campo sería otro costo adicional que habría que tener en cuenta en caso de decidir mejorar la red y el producto final. Además, contar con servidores propios supone una inversión considerable pues es necesario disponer de los recursos hardware y del espacio físico donde albergarlos, así como una buena conexión a internet para la comunicación entre los dispositivos móviles y dicho servidor.

Por último, no podemos olvidar el salario de toda aquella persona involucrada en el desarrollo, ya que como mínimo, debe existir un programador capaz de dar soporte a la aplicación para lanzar nuevas versiones donde se corrijan los posibles fallos que se encuentren. En un primer momento, podemos suponer que esta persona sería la propia creadora del producto de manera que su salario dependería completamente del éxito de la aplicación y de su modelo de negocio, el cual explicaré en el siguiente apartado.

Modelo de negocio

Hemos visto que ofrecer un producto software de estas características lleva asociado una serie de gastos importantes por lo que, para conseguir obtener beneficios, es necesario crear un modelo de negocio que sea efectivo a la vez que atractivo de cara a los usuarios finales. La primera opción que se viene a la mente es fijar un precio de descarga para aquel usuario que desee utilizar el software, pero dado que los costos descritos se generan mensualmente, es un sistema totalmente inviable debido a la imposibilidad de obtener beneficio, salvo estableciendo un precio de adquisición muy elevado que ahuyentaría a todo potencial cliente. Tras esta reflexión concluí que la opción más segura era lograr un sistema de ingreso basados en cuotas como ocurre con plataformas de series y películas como Netflix, pero este modelo tenía serias carencias pues las manchas cutáneas no constituyen un aspecto que preocupe a diario a la sociedad, de manera que un usuario de a pie no vería interesante realizar un desembolso económico todos los meses por una aplicación cuyo uso sería más bien ocasional.

Por tanto, este producto debe ser planteado de una manera muy diferente y basándose en un principio establecido desde el comienzo de este documento, donde se insiste en que *SkinScanner* no es un sustituto de un especialista en dermatología sino un complemento a este. De esta manera, al igual que cuando contratamos un seguro privado nos ofrecen una serie de ventajas para diferenciarse de la competencia, ¿por qué no considerar *SkinScanner* un producto adicional que incluir a su oferta para ofrecer a todos sus clientes? De esta manera todas las partes pueden obtener beneficios, ya que el usuario final no tiene que realizar ningún desembolso extra si ya pertenece a una compañía aseguradora que incluye el producto y, además, dichas compañías serían más atractivas al incluir en sus productos una solución completamente innovadora, a cambio de pagar unas cuotas en función de la cantidad de clientes con acceso a nuestro software. Por último, como desarrollador del proyecto podría tener acceso a convenios con las aseguradoras, firmados por varios años, que permitan una fuente de ingresos estable y suficiente para consolidar el producto y ganar adeptos, al mismo tiempo que se continúan añadiendo funcionalidades al sistema para enriquecerlo y aportar mayor valor de negocio con nuevos casos de uso, tal y como veremos más adelante.

Por tanto, el producto se entendería como un añadido a la oferta de seguros médicos de determinadas compañías, favoreciendo el atractivo de estas, en vez de como un producto indispensable en el día a día de las personas por el cual merecería la pena pagar una suscripción mensual. Además de conseguir un convenio con una entidad, estamos dirigiéndonos a un público mucho más amplio que al que llegaríamos mediante la publicación de la aplicación en las tiendas disponibles, si bien es cierto que será un proceso mucho más costoso atraer a dichas empresas.

Por último, si necesitáramos obtener mayor cantidad de ingresos podríamos ofrecer a las distintas comunidades científicas dedicadas a la detección y prevención de este tipo de patologías, acceso a todas aquellas imágenes que sean tomadas mediante nuestra aplicación, siempre y cuando los usuarios que sacaron dichas imágenes hayan

dado su consentimiento. Este valioso aporte de muestras nos permitiría tener unos ingresos adicionales al mismo tiempo que colaboramos con la causa, lo cual considero que se trata de un aliciente muy positivo, pues el principal objetivo de este proyecto desde sus inicios, era poder colaborar con los problemas causados en nuestras islas con motivo de la fuerte radiación solar a la que estamos expuestos la mayor parte del año. Así mismo, al colaborar con comunidades científicas estas podrían mencionarnos en sus investigaciones como proveedores de muestras para sus estudios, con lo que ganaríamos fama y, sobre todo, reputación que nos haría que las aseguradoras y usuarios finales depositaran su confianza en nuestro producto con mayor facilidad.

Conclusiones

Llega el momento de echar la vista atrás y valorar todo camino recorrido hasta llegar al objetivo, con un producto software completo que me causa gran orgullo al saber que ha sido resultado de mucho esfuerzo y dedicación. Sin duda ha sido una experiencia muy diferente a lo imaginado en un primer momento pues, inocentemente, consideraba que el Trabajo de Fin de Título sería similar a un proyecto de una asignatura, pero de mayor envergadura. Al poco tiempo de comenzar, se evidencian dificultades debido a la falta de documentación para realizar determinadas tareas. Además, debes ser totalmente autónomo, tomar decisiones clave para el desarrollo del software y enfrentarte a todos los problemas que van apareciendo únicamente con la ayuda del tutor o tutora y de las fuentes de información disponibles en internet.

Si bien gracias a las ganas aplicadas en el proceso, consigues que estos aspectos que al principio suponen obstáculos de importantes dimensiones, acaben transformándose en pequeñas lecciones que te ayudan a ser mejor ingeniero, pues mejoras en autonomía, te vuelves más creativo y, sobre todo, asumes responsabilidades en cuanto a plazos y requisitos que debes cumplir. Y, por supuesto, una vez has terminado el desarrollo te sientes enormemente orgulloso de haber sido capaz de elaborar un proyecto completo, desde la búsqueda de ideas hasta la puesta en funcionamiento del programa o aplicación. Esto último supone una gran motivación de cara a la entrada en el mercado laboral porque ganas confianza en ti mismo y compruebas en primera persona que eres capaz de afrontar retos importantes, aunque inicialmente te encuentres perdido.

Estoy enormemente satisfecho del desempeño en cuanto al desarrollo de la aplicación Android. Siempre he disfrutado creando software para este sistema operativo, aunque nunca había tenido la oportunidad ni el tiempo de crear una aplicación verdaderamente funcional y con utilidad de cara a los usuarios finales. Sin embargo, gracias a este proyecto y bajo mi punto de vista, he logrado ofrecer un producto con un salto de calidad notorio frente a mis anteriores intentos, siguiendo en todo momento los patrones y recomendaciones de la propia Google y cuidando cualquier detalle por mínimo que fuera. Además, siempre tuve curiosidad por el manejo de APIs y en *SkinScanner* encontré la oportunidad perfecta para enfrentarme a lo desconocido, aprender su funcionamiento y realizar una implementación totalmente funcional y eficiente.

Del mismo modo, he adquirido muchas nuevas facultades en cuanto al manejo de máquinas virtuales remotas y su empleo como servidores, lo cual siempre había sido una incógnita para mí. Linux es un sistema operativo que me agrada mucho por sus libertades de manejo, así como por su sencillez y poder abarcar un poco de este mundo en mi proyecto. Considero que ha contribuido a reflejar en el TFT todos los campos de la informática que más me han marcado durante mis cuatro años como estudiante del Grado. Evidentemente, en este punto de las conclusiones debo mencionar la inteligencia artificial ya que el poder entender el funcionamiento de determinados modelos para mí ha sido uno de los logros más satisfactorios como estudiante. Y ya no solo conocer su

operativa sino, además, ser capaz de desarrollar mis propios modelos para la consecución de diversos objetivos. Sin duda, este campo es increíblemente amplio y dudo llegar a comprenderlo en su totalidad; pero estoy seguro de que nunca dejaré de interesarme en él y buscar nuevas fuentes donde poder seguir formándome y enriqueciendo mis conocimientos sobre IA.

En cuanto al desarrollo concreto para este proyecto debo admitir que nunca imaginé que tendría tantas dificultades para poder obtener un modelo relativamente preciso para mi aplicación. La falta de conocimientos en el campo médico me hizo creer que, con un poco de esfuerzo, sería capaz de lograr una red neuronal capaz de reconocer cualquier imagen que se pusiese por delante, pero desgraciadamente, la realidad es muy distinta. Durante el desarrollo pude charlar con varios amigos que se encuentran estudiando carreras de Ciencias de la Salud y entre todos, pudimos concluir que es totalmente imposible que la IA logre predecir con total certeza si la mancha dermatológica que posee una persona es “x” o “y” enfermedad. Esto se debe a la inmensa variabilidad que existe en este campo de la salud, donde cada mancha es un caso totalmente diferente al anterior y que, a pesar de poder tener la misma patología, su materialización en la piel puede ser radicalmente distinta. Por otro lado, bajo mi experiencia en las sucesivas pruebas realizadas puedo afirmar sin temor a equivocarme, que aspectos tan simples como el color de piel, vello encima de la mancha, brillo de la fotografía o, incluso, el nivel de nitidez, son factores que complican exponencialmente el entrenamiento de un modelo para realizar predicciones sobre dichas muestras.

A pesar de no haber logrado un modelo completamente fiable, considero que he realizado un trabajo bastante aceptable teniendo en cuenta las dificultades expuestas, así como, el empleo de un dataset completamente desbalanceado para realizar los entrenamientos de manera correcta y que debí equilibrar manualmente, con la consecuente pérdida de muestras para la tarea. Así mismo, el haber encontrado tantos problemas ha enriquecido la experiencia, pues he dedicado muchas horas a la investigación y realización de pruebas para encontrar las causas de todos los problemas acaecidos.

En definitiva, estos meses han supuesto un completo desafío y, aunque en muchas ocasiones deseé haber escogido un tema para mi TFT mucho más sencillo, estoy gratamente satisfecho con el resultado logrado y, en especial, con el sinfín de nuevos conocimientos y habilidades desarrolladas gracias a este.

Trabajos futuros

SkinScanner se puede presentar como el complemento perfecto para tu supervisión dermatológica, sin llegar a sustituir en ningún caso al profesional en la materia. De hecho, este producto tiene un gran potencial de mejora mediante la implementación de nuevas funcionalidades que permitan no solo recalcar la importancia del dermatólogo, sino que este pase a ser un elemento clave dentro del software. Las consultas de estos profesionales suelen estar caracterizadas por largas esperas para obtener una cita que, en multitud de ocasiones, tiene como objetivo realizar una consulta sencilla al especialista que podría ser resuelta en cuestión de minutos y sin necesidad de acudir al centro. Pues bien, aquí nos encontramos con el punto de inflexión que haría que este proyecto se convirtiera en una potentísima herramienta tanto para los usuarios pacientes como para los especialistas dermatólogos.

La aplicación actuaría como vía de comunicación entre una persona interesada en analizar la muestra de su piel fotografiada y un especialista que podría visualizar dicha imagen y realizar una primera valoración y, en caso de requerirlo, indicar al usuario la importancia de acudir rápidamente a un centro especializado para un análisis más preciso y seguro. Del mismo modo, una persona podría directamente comunicarse con un profesional para realizar una consulta rápida, consiguiendo evitar la espera por obtener una cita y las molestias que ocasiona el desplazamiento más el consiguiente gasto económico que supone la visita.

Por supuesto, sería necesario determinar qué dermatólogos serían ofertados en la aplicación para establecer una conversación vía chat y esta responsabilidad sería otorgada a aquellas compañías aseguradoras que hayan decidido contratar nuestro servicio, de manera que ambas partes se ven beneficiadas. Por un lado, la aseguradora sabe con certeza que sus clientes obtendrán asesoramiento online por parte de los profesionales en los que ya ha depositado su confianza y, además, estos ganarían popularidad pues la aplicación al fin y al cabo se convertiría en un expositor donde promocionar su consulta. Para nosotros como negocio supone una liberación de trabajo y negociaciones pues no necesitamos buscar profesionales interesados. Por otro lado, cada dermatólogo podría obtener una pequeña comisión por cada consulta online realizada de manera que se convierta en un incentivo adicional para que ofrezcan el mejor servicio posible.

Agradecimientos

Tras todos estos meses de trabajo e investigación, considero que la mejor manera de concluir esta memoria es realizando un especial agradecimiento a mi tutor. Tras estos meses de trabajo e investigación, puedo afirmar sin temor a equivocarme que D. Francisco José Santana Pérez es un grandísimo docente, con una vocación envidiable hacia su profesión y siempre dispuesto a ayudar en lo que necesites independientemente del día y hora que sea. Antes de comenzar el último curso tenía muchísimas dudas e incertidumbres acerca de cómo sería la relación con un tutor de TFT y, al menos en mi caso, no puedo estar más contento de haber podido contar con su supervisión y ayuda en todo momento.

Bibliografía

1. https://www.kaggle.com/kmader/skin-cancer-mnist-ham10000
2. https://coursera.org/verify/specialization/JEPPXDS2NGMT
3. https://blog.keras.io/building-a-simple-keras-deep-learning-rest-api.html
4. https://voragine.net/linux/como-crear-y-gestionar-demonios-en-linux-con-systemd
5. https://developer.android.com/training/camera/photobasics
6. https://developer.android.com/jetpack/docs/guide
7. https://developer.android.com/training/data-storage/room
8. https://codelabs.developers.google.com/codelabs/android-room-with-a-view-kotlin/#0
9. https://github.com/bumptech/glide
10. https://course.fast.ai/
11. https://docs.fast.ai/basic_train.html
12. https://anaconda.org/fastai/fastai
13. https://www.mayoclinic.org/
14. https://arxiv.org/abs/1608.06993
15. https://github.com/NSR98/TFT-SkinScanner (repositorio)

Anexo I. Manual de usuario

Manual de usuario SkinScanner

No olvide que está usando una aplicación orientativa. Asegúrese de acudir a un centro especializado.

1.

Tome o suba una **fotografía** de la mancha cutánea que desea analizar.

2.

Una vez la imagen ha sido procesada, acceda al **historial** y verá el resultado correspondiente.

3.

Puede acceder a los **detalles** de la muestra.

4.

Obtenga **información** verificada acerca de la enfermedad predicha por la aplicación.

5.

Puede añadir **anotaciones** para recordar, por ejemplo, en qué parte del cuerpo se encuentra la mancha.

6.

En la sección "Dermatólogos" podrá encontrar **centros** especializados cercanos a su ubicación actual.

7.

Recuerde acudir a un **especialista** para conocer a ciencia cierta si padece una enfermedad y poder tratarla en ese caso.

TFT Nahuel Sánchez Rodríguez

Anexo II. Clases Kotlin destacadas

1. Base de datos

```
@Database(entities = [SkinRecord::class, DermatologistRecord::class],
version = 2, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun skinDAO():SkinDAO
    abstract fun dermatologistDAO(): DermatologistDAO

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getAppDatabase(context: Context): AppDatabase {
            val tempInstance =
                INSTANCE
            if (tempInstance != null) {
                return tempInstance
            }
            synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "AppDatabase"
                ).build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

2. Entidad de muestra dermatológica

```
@Entity(tableName = "skin")
data class SkinRecord(
    @PrimaryKey(autoGenerate = true) val id: Int,
    @ColumnInfo(name = "imagePath") val imagePath: String,
    @ColumnInfo(name = "result") var result: String,
    @ColumnInfo(name = "date") val date: String,
    @ColumnInfo(name = "annotations") var annotations: String
)
```

3. Entidad de centro dermatológico

```
@Entity(tableName = "dermatologist")
data class DermatologistRecord (
    @PrimaryKey (autoGenerate = true) val id: Int,
    @ColumnInfo(name = "name") val name: String,
    @ColumnInfo(name = "location") val location: String,
    @ColumnInfo(name = "rating") val rating: Double,
    @ColumnInfo(name = "lat") val lat: Double,
    @ColumnInfo(name = "lng") val lng: Double
)
```

4. Clases obtenidas a partir de respuesta JSON

```
data class Response (
    @SerializedName("html_attributions") val htmlAttributions:
List<Object>,
    @SerializedName("results") val results: List<Result>,
    @SerializedName("status") val status: String
)
```

```
data class Result (
    @SerializedName("geometry") val geometry: Geometry,
    @SerializedName("icon") val icon: String,
    @SerializedName("id") val id: String,
    @SerializedName("name") val name: String,
    @SerializedName("opening_hours") val openingHours: OpeningHours,
    @SerializedName("photos") val photos: List<Photo>,
    @SerializedName("place_id") val placeId: String,
    @SerializedName("plus_code") val plusCode: PlusCode,
    @SerializedName("rating") val rating: Double,
    @SerializedName("reference") val reference: String,
    @SerializedName("scope") val scope: String,
    @SerializedName("types") val types: List<String>,
    @SerializedName("user_ratings_total") val userRatingsTotal: Int,
    @SerializedName("vicinity") val vicinity: String
)
```

```
data class Geometry (
    @SerializedName("location") val location: Location,
    @SerializedName("viewport") val viewport: Viewport
)
```

```
data class Location (
    @SerializedName("lat") val lat: Double,
    @SerializedName("lng") val lng: Double
)
```

5. GPS

```
class Gps(private val context: Context) {
    private var locationManager : LocationManager? = null
    private var locationListener: LocationListener

    init {
        locationManager = context.getSystemService(LOCATION_SERVICE)
as LocationManager?
        locationListener = object : LocationListener {
            override fun onLocationChanged(location: Location) {}
            override fun onStatusChanged(provider: String?, status:
Int, extras: Bundle?) {}
            override fun onProviderEnabled(provider: String?) {}
            override fun onProviderDisabled(provider: String?) {}
        }
    }

    @SuppressLint("MissingPermission")
```

```
    fun requestLocation(): Location {
        runBlocking {
            launch {

locationManager?.requestLocationUpdates(LocationManager.GPS_PROVIDER,
0L, 0f, locationListener)
            }
        }
        return
locationManager!!.getLastKnownLocation(LocationManager.PASSIVE_PROVIDE
R)
    }
}
```

6. Captura de imágenes

```
class Camera(val context: Context) {

    fun createIntent(file: File): Intent {
        val builder: StrictMode.VmPolicy.Builder =
StrictMode.VmPolicy.Builder()
        StrictMode.setVmPolicy(builder.build())

        var takePictureIntent =
Intent(MediaStore.ACTION_IMAGE_CAPTURE)

        if (takePictureIntent.resolveActivity(context.packageManager)
!= null) run {
            val picturePath = file.absolutePath
            val pictureURI = FileProvider.getUriForFile(context,
                "com.example.skinscanner.provider", file)

            takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
pictureURI)
            takePictureIntent.putExtra("path", picturePath)
        }
        return takePictureIntent
    }

    fun createPictureFile(): File {
        val picturesFolder = "/storage/emulated/0/Pictures"
        val skinScannerDirectory: File =
File("$picturesFolder/SkinScanner")

        if (!skinScannerDirectory.exists())
skinScannerDirectory.mkdir()

        return File("$skinScannerDirectory/" +
                    System.currentTimeMillis() + "_SkinScanner.jpg"
        )
    }
}
```

7. Verificación de conexión

```
class ConnectionDetector(private val context: Context) {

    fun verifyNetwork(): Boolean {
        val connectivityManager =
context.getSystemService(Context.CONNECTIVITY_SERVICE) as
ConnectivityManager
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            val nw      = connectivityManager.activeNetwork ?: return
false
            val actNw = connectivityManager.getNetworkCapabilities(nw)
?: return false
            return when {
                actNw.hasTransport(NetworkCapabilities.TRANSPORT_WIFI)
-> true

actNw.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) -> true
                else -> false
            }
        } else {
            val nwInfo = connectivityManager.activeNetworkInfo ?:
return false
            return nwInfo.isConnected
        }
    }
}
```

Anexo III. Código destacado

1. File Provider

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="com.example.skinscanner.provider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths" />
</provider>
```

2. Adaptador RecyclerView

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    var skinRecord = skinRecords[position]
    holder.name.text =
context.resources.getString(SkinLesions.names[skinRecord.result] ?:
error("N/A"))
    holder.date.text = skinRecord.date

    Glide
        .with(context)
        .load(File(skinRecord.imagePath))
        .centerCrop()
        .into(holder.image)

    holder.detailsButton.setOnClickListener(View.OnClickListener {
        val seeDetailsIntent = Intent(context,
SkinRecordDetails::class.java).apply {
            MySharedPreferences(context).put("id",
skinRecord.id.toString())
        }
        context.startActivity(seeDetailsIntent)
    })

    holder.deleteButton.setOnClickListener {
        val builder = AlertDialog.Builder(context)

builder.setTitle(context.getString(R.string.title_delete_data))

builder.setMessage(context.getString(R.string.text_confirm_delete_data
))
        builder.setIcon(R.drawable.ic_delete)

builder.setPositiveButton(context.getString(R.string.text_confirm)) {
_, _ ->
            runBlocking {
                launch {

AppDatabase.getAppDatabase(context).skinDAO().removeSkinRecord(skinRec
ord)
                }
            }
        }
```

```
        builder.setNegativeButton(R.string.cancel_button) {_, _ -> }
        builder.show()
    }
}
```

3. Petición HTTP y obtención de respuesta JSON

```
try {
    url = URL(

"https://maps.googleapis.com/maps/api/place/nearbysearch/json?"
                +
"location=${location.latitude},${location.longitude}"
                + "&rankby=$rankBy"
                + "&type=$type"
                + "&keyword=$keyword"
                + "&key=$apiKey"
    )

    val urlConnection: HttpURLConnection = url.openConnection() as
HttpURLConnection
    urlConnection.requestMethod = "POST"
    urlConnection.setRequestProperty("Content-Type",
"application/json; utf-8")
    urlConnection.setRequestProperty("Accept", "application/json")
    urlConnection.doOutput = true
    urlConnection.connectTimeout = 4000

    val stream: InputStream = urlConnection.content as InputStream
    val bufferedReader: BufferedReader = BufferedReader(
        InputStreamReader(
            stream,
            "utf-8"
        ), 8
    )
    val stringBuilder = StringBuilder()

    BufferedReader(bufferedReader).use { r ->
        r.lineSequence().forEach {
            stringBuilder.append(it + "\n")
        }
    }
    stream.close()
    bufferedReader.close()

    val jsonObject: JSONObject = JSONObject(stringBuilder.toString())
    val response: Response =
        Gson().fromJson(jsonObject.toString(), Response::class.java
```

4. Datos de entrenamiento

```
data = ImageDataBunch.from_df(path='/content/drive/My
Drive/Desarrollo/KAGGLE/DATASET', df=labels, ds_tfms=tfms, size=112,
bs=32, valid_pct=0.2, fn_col='path',
label_col='dx').normalize(imagenet_stats)
```

5. Ciclo de entrenamiento

```
learn.fit_one_cycle(11, slice(1e-02))
```

epoch	train_loss	valid_loss	error_rate	accuracy	time
0	1.908635	1.410214	0.420619	0.579381	36:44
1	1.535007	1.237373	0.412371	0.587629	00:31
2	1.345110	1.261487	0.430928	0.569072	00:26
3	1.198182	1.116789	0.417526	0.582474	00:26
4	1.029480	1.015851	0.383505	0.616495	00:26
5	0.938035	0.951187	0.342268	0.657732	00:26
6	0.851326	0.854178	0.319588	0.680412	00:26
7	0.793549	0.779668	0.300000	0.700000	00:26
8	0.735173	0.777945	0.292783	0.707217	00:26
9	0.661095	0.761256	0.298969	0.701031	00:26
10	0.649279	0.758180	0.289691	0.710309	00:26

Tabla 1. Resultados tras 11 epochs

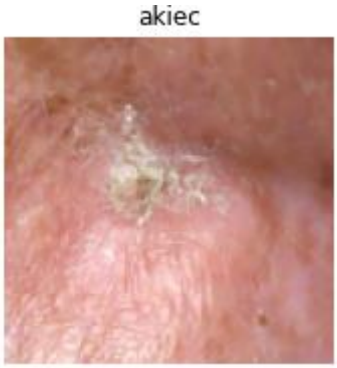
6. Aprendiz/Learner

learn = cnn_learner(data, arch, metrics= [error_rate, accuracy], callback_fns=ShowGraph)

```
Downloading: "https://download.pytorch.org/models/densenet121-a639ec97.pth" to /root/
100%                                 30.8M/30.8M [00:00<00:00, 203MB/s]
```

7. Predicción sobre imagen externa

```
img = open_image('/content/drive/My
Drive/Desarrollo/KAGGLE/TFG_DEMO/akiec_1.jpg')
img.resize(112)
img.show(y=learn.predict(img)[0])
```




Skin
Scanner