Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

TRABAJO DE FIN DE GRADO

Grado en Ingeniería informática

Aplicación Web para diagnóstico de glaucoma basado en Deep Learning a partir de la recopilación de imágenes de retina

Web application for the diagnosis of glaucoma based on Deep Learning from the collection of images of the retina

---

Autor: Carlos Arvelo García

San Cristóbal de La Laguna, 11 de septiembre de 2020

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

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

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

Que la presente memoria titulada: Aplicación Web para el diagnóstico del glaucoma basado en Deep Learning a partir de la recopilación de imágenes de la retina

ha sido realizada bajo su dirección por D. Carlos Arvelo García, con N.I.F. 51.165.279-P.

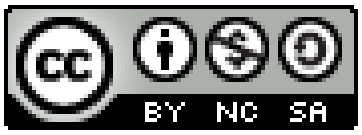
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de septiembre de 2020.

Agradecimientos

En primer lugar, quiero agradecer a los tutores José Francisco Sigut Saavedra y Francisco José Fumero Batista por orientarme y ayudarme a realizar este proyecto de TFG, así como el apoyo y compresión durante su desarrollo.

También agradecer a mi familia por el apoyo constante y los ánimos ofrecidos durante el desarrollo.

Licencia



Resumen

El proyecto surge de la necesidad de mejora de diagnóstico de una enfermedad frecuente en nuestro medio: el glaucoma. Se trata de una de las principales causas de pérdida irreversible evitable de visión a nivel mundial (en España afecta casi a un 3% de la población según la Sociedad Española de Glaucoma). Esta patología lesiona el nervio óptico progresivamente; lo cual puede detectarse a través del estudio del fondo de ojo, en concreto de la afección de la papila, que es el punto de unión de la retina con el nervio óptico, como se puede apreciar en la siguiente figura.

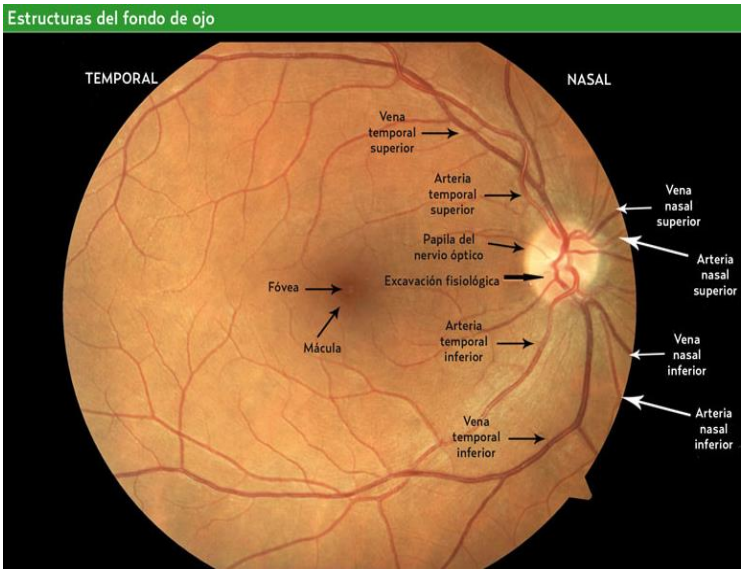


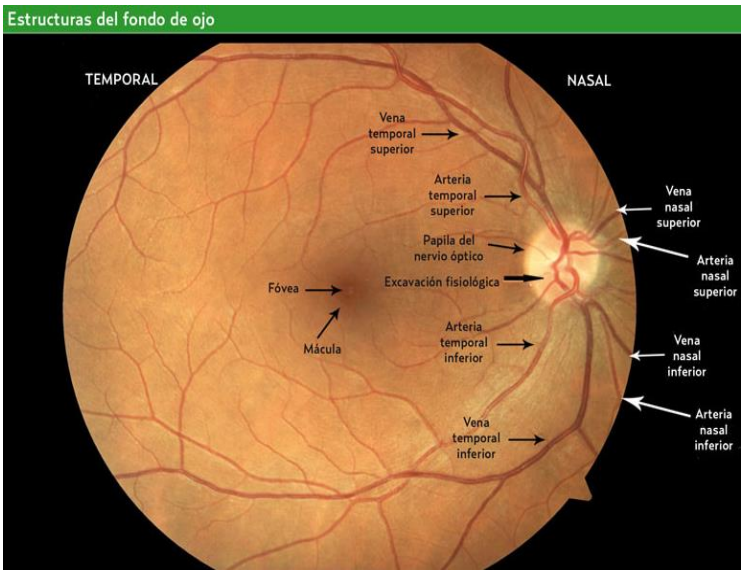
Imagen de fondo de ojo con estructuras anatómicas señaladas (imagen de revista médica de Actualización en Medicina de Familia).

Este proyecto parte de la idea de ofrecer mediante una aplicación web, una red neuronal como servicio de diagnóstico de glaucoma a partir de imágenes de fondo de ojo, generando así un reclamo para obtener retinografías y aumentar la base de datos que servirá como base para entrenar redes neuronales. Para llevar a cabo este objetivo se decidió ofrecer el servicio a través de una aplicación web.

Palabras clave: glaucoma, diagnostico, redes neuronales convolucionales, deep learning, nervio óptico, aplicación web.

Abstract

The project arises from the need to improve the diagnosis of a common disease in our environment: glaucoma. It is one of the main causes of irreversible avoidable loss of vision worldwide (in Spain it affects almost 3% of the population according to Spanish Glaucoma Society). This pathology damages the optic nerve progressively; which can be detected by studying the fundus, specifically the condition of the papilla, the point where the retina attaches the optic nerve, as the image below shows.



Fundus image with anatomical structures indicated (image from the medical journal of Family Medicine Update).

This project starts from the idea of offering a neural network through a web application as a glaucoma diagnostic service, based on fundus images, thus generating a claim to obtain retinographies and increase the database that will serve as a basis for training neural networks. To carry out this objective, it was decided to offer the service through a web application.

Keywords: glaucoma, diagnosis, convolutional neural networks, deep learning, optic nerve, web application.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

1.1 Objetivos

El diagnóstico automatizado de patologías de la retina, como el glaucoma, requiere de algoritmos y redes neuronales con un entrenamiento fiable y completo. Para ello es preciso disponer de una gran base de datos de imágenes etiquetadas por especialistas que sirvan para generar conjuntos de entrenamiento y testing para dichas redes.

El objetivo de este proyecto consiste en desarrollar y desplegar una aplicación web para el diagnóstico del glaucoma y su correspondiente servicio web para la recopilación de imágenes del fondo de ojo, utilizando tecnologías actuales. La aplicación web solicitará al usuario subir una imagen de retina y ofrecerá automáticamente un diagnóstico utilizando un modelo de Deep Learning para informar al usuario si la imagen corresponde a un ojo sano o a un glaucoma. Posteriormente, se solicitará retroalimentación al usuario sobre la exactitud del diagnóstico con el objetivo de ampliar la colección de imágenes utilizada en el entrenamiento de los modelos de Deep Learning.

1.2 Estado actual del tema

Describiremos algunos antecedentes relacionados con este proyecto, tanto desde el punto de vista tecnológico como médico.

1.2.1 Tecnologías web

Desde una perspectiva tecnológica, nos centraremos en las herramientas y librerías disponibles que utilizaremos para el desarrollo del Front-End y Back-End de la aplicación. En este sentido y en relación con las tecnologías utilizadas para desarrollar interfaces de usuario, utilizaremos

React [1], una librería diseñada por Facebook para construir interfaces de usuario en JavaScript y la más extendida y utilizada en el desarrollo de Front-End. Asimismo, para el Back-End podemos mencionar el micro-framework Flask [2] de Python, uno de los más populares, que permite la implementación de endpoints de manera fácil y segura, ya que incorpora módulos de seguridad para cifrar y manejar la información. Adicionalmente, simplificará la integración de los modelos de redes neuronales convolucionales, ya que Python es uno de los entornos más populares para su desarrollo y el que utilizaremos durante este proyecto. Centrándonos ahora en esta última parte, las tecnologías disponibles para Deep Learning, disponemos de herramientas como TensorFlow [3] y Keras [4], que nos permiten implementar y ejecutar modelos de redes neuronales, y hacer uso de las mismas tanto en la nube, como en nuestro equipo local, o incluso en el navegador, apoyándonos por ejemplo en TensorFlow.js. Esta última librería permite ejecutar un modelo de red neuronal directamente en el navegador, reduciendo así la carga computacional en los servidores. Por otra parte, también podríamos utilizarla para integrar redes neuronales en un servidor web desarrollado en Node.js en lugar de Python.

Un ejemplo del uso de estas tecnologías es CaptionBot [5] de Microsoft, que, si bien no está enmarcado en el ámbito médico, tiene cierta similitud con el sistema que se quiere implementar en este TFG. Se trata de una web que permite al usuario subir una imagen a un servidor, donde se utilizará una red neuronal para identificar el contenido de la imagen y describirlo mediante lenguaje natural, mostrando esta descripción al usuario.

1.2.2 Uso de sistemas informáticos en el campo médico

En este campo, uno de los primeros sistemas que se popularizaron para diagnóstico online consistían en ofrecer al paciente la posibilidad de realizar una consulta online a un profesional sanitario, como por ejemplo el sistema idoc24 [6], que permite enviar una imagen dermatológica para su posterior evaluación y diagnóstico online por parte de un especialista. En este caso, se

hace uso de internet para mayor comodidad del usuario, pero no se utiliza ninguna técnica de inteligencia artificial para proporcionar un diagnóstico. Sin embargo, actualmente en el campo de la medicina el uso de Deep Learning se está popularizando ya que se necesita de los últimos avances para proporcionar los mejores servicios a los pacientes. Éste es el caso de RadIO [7], un sistema que fue lanzado por el Departamento TI del Gobierno de Moscú. Se trata de un proyecto open source de detección de cáncer que usa Deep Learning para encontrar signos de cáncer de pulmón en imágenes de tomografía computarizada. Por último, podemos citar RetinaLyze [8], una aplicación similar a la que se propone en este TFG, que suma tanto el diagnóstico online con el uso de Deep Learning, en la que se sube una imagen de la retina a la página web y mediante técnicas de inteligencia artificial identifica las enfermedades oculares que puedas tener en base a una fotografía.

2 Arquitectura del sistema

La arquitectura del proyecto se divide en cuatro partes fundamentales, las cuales se identifican como el Front-end, Back-end, base de datos y la red neuronal que se ofrece como servicio.

- Front-end: muestra la interfaz de la aplicación y hace las llamadas al backend para obtener los resultados.
- Back-end: se encarga de la comunicación entre el cliente, la base de datos y la red neuronal.
- Base de datos: almacena la imagen, la predicción y otros datos relacionados con ellas.
- Red neuronal: Como servicio procesa las imágenes y guarda en la base de datos el resultado de la predicción.

En la figura 1 podemos ver un ejemplo de cómo se desarrolla el flujo de la aplicación para realizar un diagnóstico.

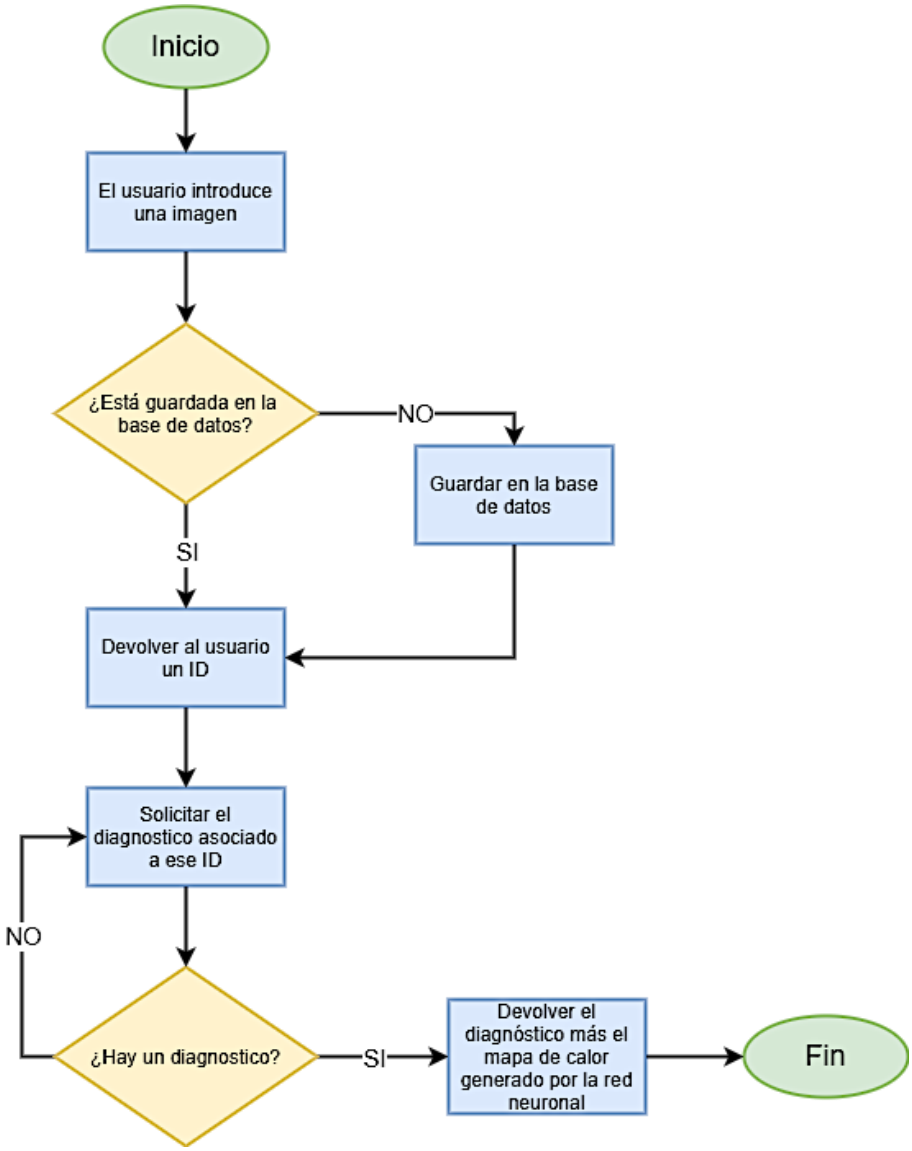


Figura 1: Diagrama de flujo del funcionamiento de la aplicación.

2.1 Estructura del Front-end

El Front-end está compuesto por dos páginas:

- Home: en esta página se desarrolla toda la lógica de la aplicación, aquí se sube la imagen, para posteriormente que el usuario genere las coordenadas de recorte y enviar los datos al Back-end, para finalmente obtener la predicción realizada por la red neuronal (figura 2).
- Guide: esta página solo sirve como documentación para orientar al usuario sobre que área se requiere de la imagen.

El Front-End está desplegado en el propio servidor y ofrecido a través

de Nginx [9], un servidor web/proxy inverso que se encarga de indexar y gestionar las peticiones a la url de la aplicación.

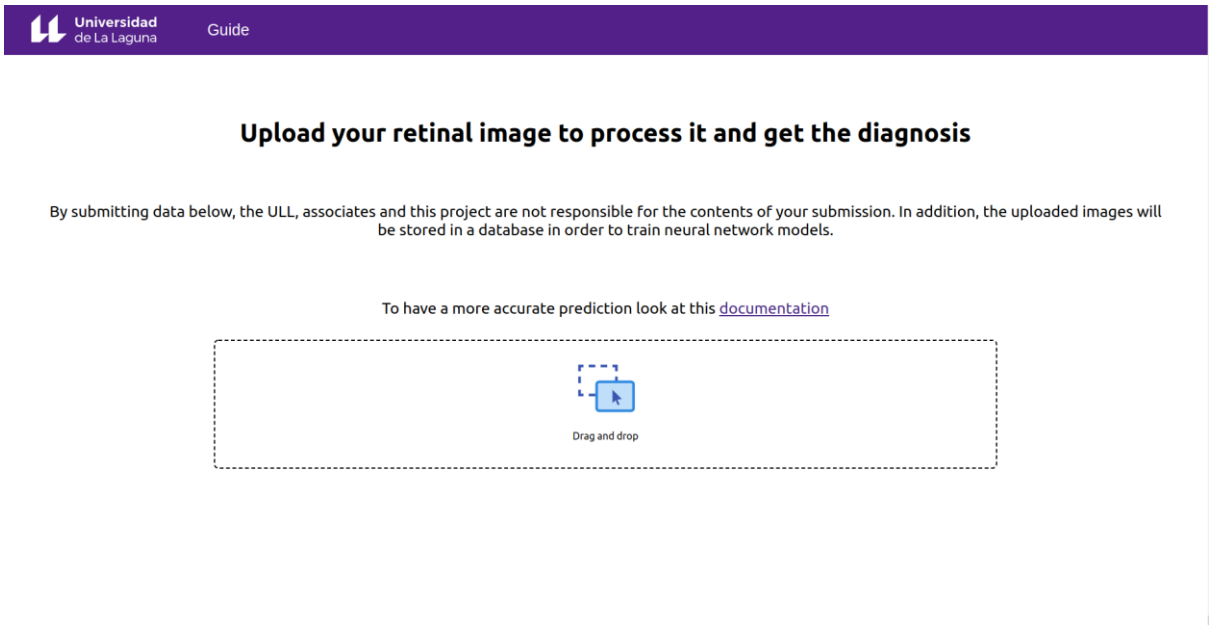


Figura 2: Home de la aplicación.

2.2 Estructura del Back-end y base de datos

El servicio REST [10] se compone principalmente por cuatro endpoints:

- /api/process: este endpoint es el que recibe la imagen y los datos referentes a ella para guardarla en la base de datos y generar un uuid que se almacenará en una cola que contiene las imágenes pendientes de ser procesadas por la red neuronal.
- /api/getDiagnosis: Una vez se almacena el uuid en la cola, el cliente llama de manera recursiva a este endpoint hasta que obtiene la predicción realizada por la red neuronal.
- /api/download: una vez obtenida la predicción se llama a este endpoint para obtener el recorte de la imagen subida por el usuario y mostrarla en la interfaz junto a la predicción.
- /api/metadata: finalmente cuando el usuario recibe el diagnóstico se le piden una serie de datos relacionados con la imagen.

El despliegue del Back-end se realiza en un docker que contiene la

imagen del servicio REST el cual dentro del contenedor permanece a la escucha continuamente debido a gunicorn [11], que es un servidor HTTP uWSGI basado en python para UNIX, que además de mantener el backend en ejecución permite balanceo de carga y gestión de workers de manera sencilla.

Adicionalmente, se han implementado una serie de métodos que ayudan a realizar todas las tareas para el correcto funcionamiento de la aplicación.

2.3 Estructura de la red neuronal como servicio

La red neuronal es el cerebro de la aplicación y es la característica principal que se ofrece como servicio a través de la aplicación. El servicio se compone de varias partes:

- Una cola: esta se encarga de almacenar los uuid's de las imágenes según el orden en el que han sido registradas en la base de datos.
- Un hilo (thread): que se encarga de estar a la escucha para extraer los uuid's de la cola para procesarlos y generar una imagen que pasar a la red neuronal.
- La red neuronal convolucional ("convolutional neural network" o "CNN"): esta se encarga de procesar la imagen y generar una predicción que será almacenada en la base de datos.
- Los pesos: es uno de los parámetros que tiene que aprender el modelo durante el proceso de entrenamiento para poder usarlo luego para realizar las inferencias.

Para que la red neuronal pueda hacer una predicción lo más acertada posible se requiere que la imagen de la retina se trate primero, cumpliendo como requisito, que se seleccione el área en la que se centra la red neuronal para inferir el resultado: la papila o cabeza del nervio óptico (figura 3).

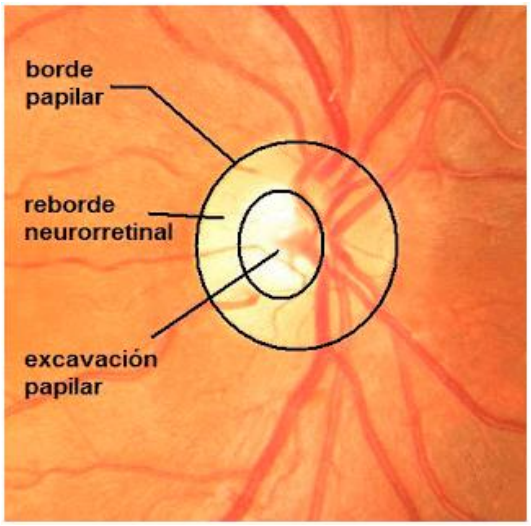


Figura 3: Áreas de la papila.

3 Diseño e implementación del cliente

Para el desarrollo del cliente se utilizó React, una librería diseñada por Facebook para construir interfaces de usuario en JavaScript y la más popular en el desarrollo de Front-End, junto al metalenguaje TypeScript [12] sustituyendo a JavaScript.

3.1 Diseño de la interfaz de usuario

La interfaz está compuesta por un área dropzone (figura 4) en el cual se subirá la imagen seleccionada, seguidamente se mostrará la imagen ofreciendo la opción de recortarla (figura 4) por el área de la del nervio óptico para mejorar la precisión de la predicción, y finalmente aparecerá la opción para subir la imagen al servidor. Una vez obtenido el resultado se le pedirá al usuario que indique si el resultado es correcto, en caso de que lo sepa con anterioridad y una serie de datos médicos referentes a la imagen.

Para la implementación y diseño de la interfaz de usuario se utilizaron varios módulos como react-image-crop [13] permitiendo al usuario tratar la imagen conforme el requerimiento de la red neuronal: recortar la imagen por el área de la papila y con una longitud igual en los lados, formando un cuadrado, para poder ser analizada por la red neuronal de la manera más óptima (figura 4).

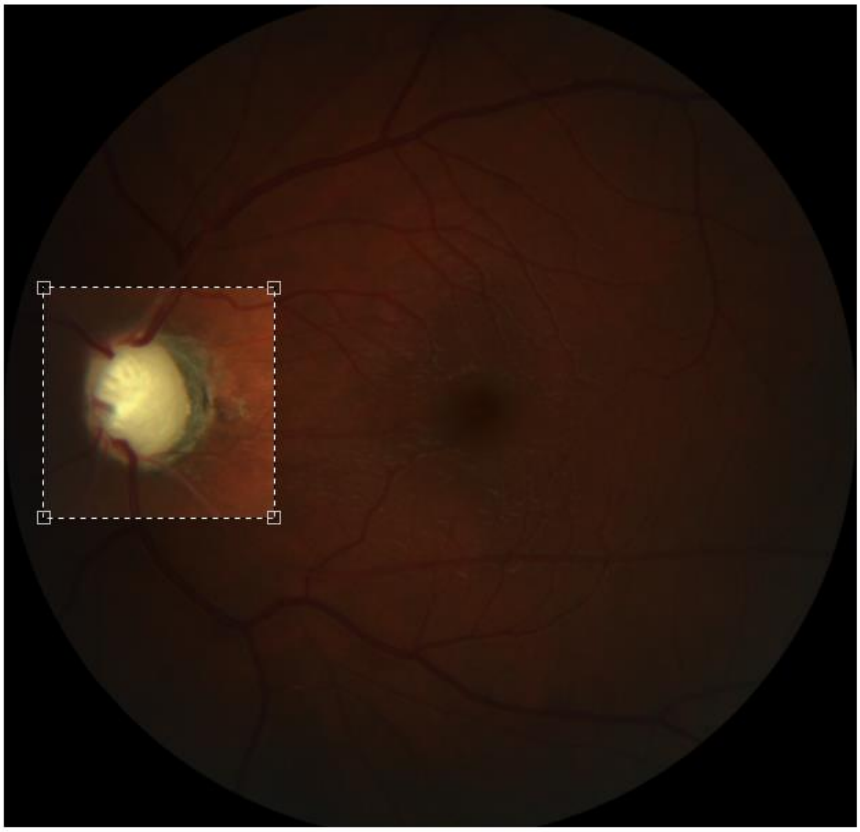


Figura 4: Recorte de la imagen subida.

En cuanto a la forma de subir la imagen se optó por crear un dropzone en el cual el usuario puede tanto subir la imagen arrastrándola, como clicando y seleccionándola en el sistema de archivos (Figura 5 y 6).

Figura 5: Área dropzone.

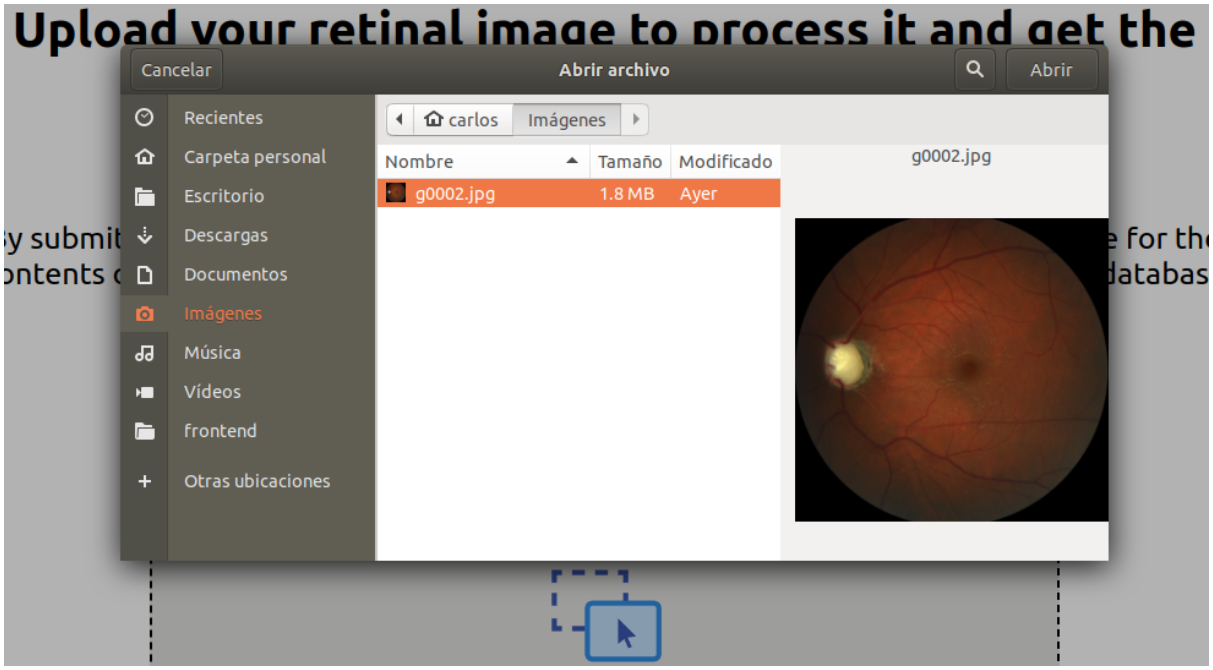


Figura 6: Área dropzone con la opción de subir una imagen desde el sistema de archivos.

Para implementar este método primero se planteó usar el módulo react-dropzone [14], pero debido a que se requería algo más sencillo que lo que ofrecía este módulo, se terminó optando por crear un componente propio que realiza las mismas funciones que este. Esto dio como resultado en una primera instancia un componente que gestionaba el procesamiento de la imagen y la realización de las llamadas al Back-end, pero finalmente se cambió de manera que el componente retornaba mediante una callback un JSON (Figura 7) con los datos de la imagen al padre del componente. Los campos mostrados en la figura 7 contienen la siguiente información:

- cropData: contiene las coordenadas de la imagen que se quieren recortar.
- imageName: contiene el nombre de la imagen.
- b64Image: contiene la imagen codificada en base64.
- imageWidth e imageHeight: contienen el tamaño del *div* donde se muestra la imagen en la interfaz.

```
export interface ImageData {
  cropData: object,
  imageName: string,
  imageWidth: number,
  imageHeight: number,
  b64Image: string
}
```

Figura 7: Datos de la imagen enviados al backend

Debido a que el tamaño de la pantalla del dispositivo varía según la resolución y tamaño de la ventana del navegador, el *div* en el cual se muestra la imagen cambia su tamaño en base a los factores nombrados anteriormente, por lo que la imagen también cambia su tamaño, haciendo que las coordenadas obtenidas para realizar el recorte de un área no se correspondan con esa misma área en la imagen de tamaño original (figura 8).

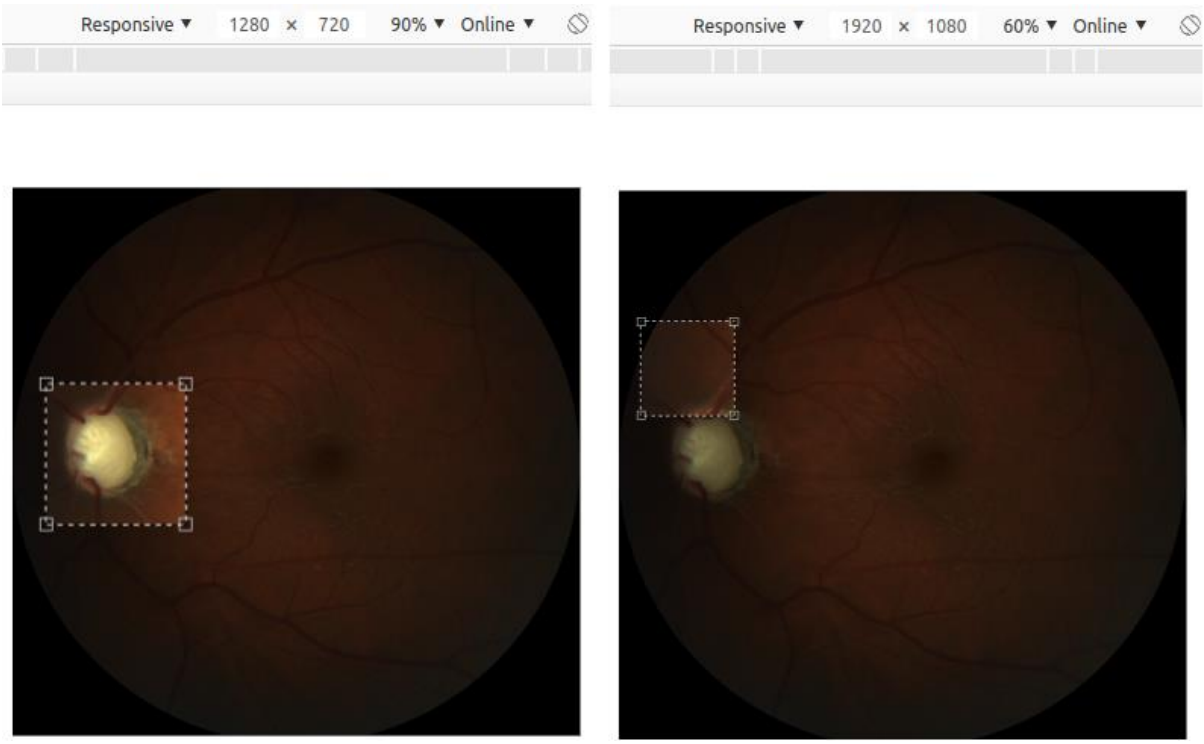


Figura 8: Diferencia entre las áreas seleccionadas para la misma imagen con distintas resoluciones y mismas coordenadas.

Por este motivo se necesita devolver el tamaño del div, para que el Back-end reescale la imagen y haga que las coordenadas se correspondan con el área que se quiere recortar.

Para realizar las peticiones HTTP entre la interfaz y el Back-end se usó el módulo axios [15] (figura 9).

```
return await axios({
  method: 'get',
  url: `http://medimag.iaas.ull.es:8080/api/download/${imageId}`,
  headers: { }
})
  .then(res => {
    return res.data
  })
  .catch(err => {
    return err
  })
```

Figura 9: Petición axios hacia la API para obtener la imagen recortada

En cuanto al diseño de la página se usó emotion [16], un módulo que fusiona el css en el propio React, el cual permite crear componentes de React que actúan como las etiquetas HTML propias con el diseño css ya incorporado, permitiendo su uso en el JSX/TSX. En la siguiente imagen (figura 10) se puede ver cómo se crean los componentes Container e Img.

```
import styled from "@emotion/styled";
import Button from "../../components/Buttons/Button";
...

const Container = styled.div`
  max-height: 80px;
  min-height: 50px;
  background: #5C068C;
  display: flex;
  align-items: center;
  width: 100%;
`

const Img = styled.img`
  width: 15%;
  min-width: 150px;
`

export const Header = () => {

   ...

  return <Container>
    <Img src={logo} alt="Ull logo" onClick={toHome}/>
    <Button value="Guide" onClick={toGuide} type='square'/>
    <Button value="About" onClick={toAbout} type='square'/>
    {currentLocation.pathname !== '/'?
      <Button value="Upload" onClick={toHome} type='square'/>
      :
      <div/>
    }
  </Container>
}
```

Figura 10: Uso de emotion para generar el css de la UI e import del componente de botones personalizado.

Durante el desarrollo de la interfaz se tomó la decisión de crear un componente para establecer un diseño genérico para los botones, permitiendo introducir botones de manera sencilla en distintas partes de la aplicación sin necesidad de estar importando o implementando css cada vez en el que se especifica el contenido del botón, forma y la función con su comportamiento.

Para informar al usuario de que la página tiene propósitos de investigación y que las imágenes subidas van a ser almacenadas se creó un modal y se estableció un texto informativo donde se especifica el carácter de la aplicación (figura 11).

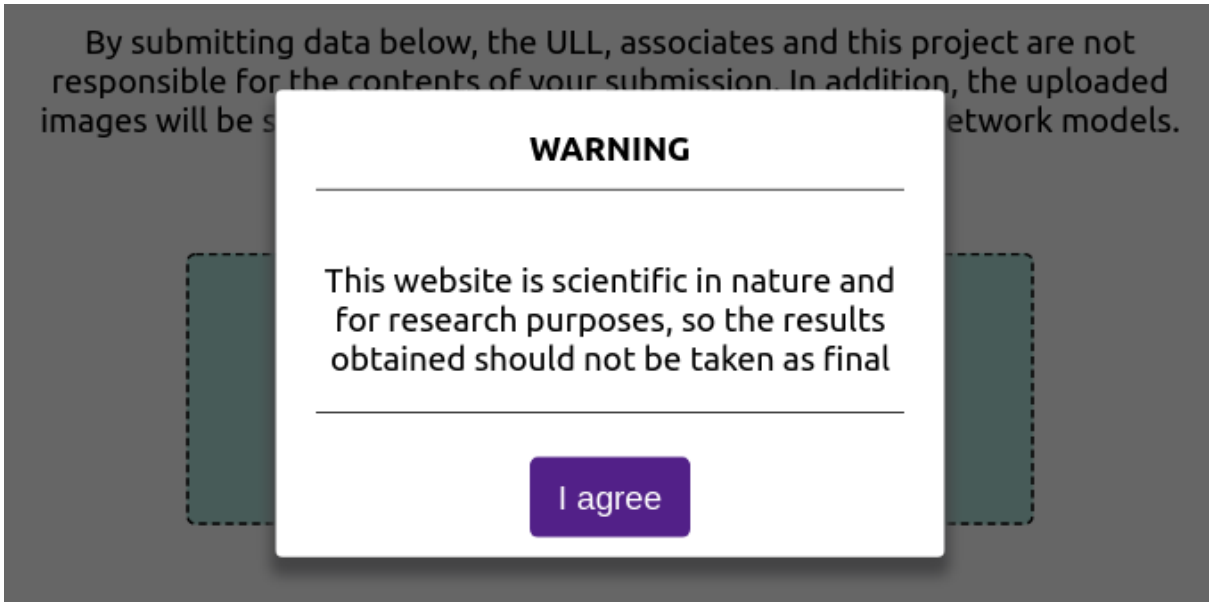


Figura 11: Modal y texto informativo.

Se optó por crear una SPA en la que todo son componentes que van renderizando según el flujo de la aplicación. Finalmente se añadió una página más, con el objetivo de proporcionar una serie de pasos de cómo usar la aplicación y por dónde se debía recortar la imagen para que pudiera ser procesada por la red neuronal.

Para el enrutamiento de las páginas utilizamos react-router-dom [17] y react-router [18], debido a que no se hacen redirecciones a estas páginas ni se le pasan parámetros, es decir son estáticas, para direccionar a ellas utilizamos los hooks useHistory y useLocation (figura 12) de react-router-dom. UseHistory almacena la url a la que se quiere redirigir y comprueba si existe alguna coincidencia con los valores de los argumento path de las etiquetas Route (figura 13). El hook useLocation devuelve entre otras cosas la ruta actual que muestra el navegador, por lo que se usó para limitar el número de veces que se puede introducir la misma url en el hook useHistory,

permitiendo así que, si el usuario ha pulsado varias veces seguidas en el botón que direcciona a otra ruta, no tenga que pulsar el mismo número de veces para volver a atrás en el navegador.

```
import React from 'react'
import styled from "@emotion/styled";
import { useHistory, useLocation } from 'react-router-dom'
import Button from "../../components/Buttons/Button";
import logo from "../../assets/ULL_logo.svg"

...
export const Header = () => {
  const currentLocation = useLocation()
  const history = useHistory();

  const toGuide = () => {
    if (currentLocation.pathname !== '/guide')
      history.push('/guide')
  }

  ...

  return <Container>
    <Img src={logo} alt="Ull logo" onClick={toHome}/>
    <Button value="Guide" onClick={toGuide} type='square'/>
    <Button value="About" onClick={toAbout} type='square'/>
    {currentLocation.pathname !== '/'?
      <Button value="Upload" onClick={toHome} type='square'/>
      :
      <div/>
    }
  </Container>
}
```

Figura 12: Uso de los hooks useHistory y useLocation

```
<Router>
  <div>
    <Header/>
    <Route exact path='/' component={Home}/>
    <Route path='/about' component={About}/>
    <Route path='/guide' component={Guide}/>
    <Footer/>
  </div>
</Router>
```

Figura 13: Pseudo-rutas que ofrecen el contenido estático de la app.

4 Diseño e implementación del Back-end

Para que la red neuronal que ofrecemos como servicio pueda recibir imágenes y devolver un resultado, se ha desarrollado una API [19] (Application Programming Interface) que ofrece una serie de endpoints que permiten la comunicación del usuario con la red neuronal y la base de datos. En este caso es una API REST [20] (Representational State Transfer). De aquí en adelante nos referiremos a ella simplemente como API.

La API del Back-end está desarrollada en el lenguaje Python para favorecer la integración con el modelo de la red neuronal.

4.1 Librerías y framework utilizados

Para el desarrollo de la API se utilizó el microframework Flask, que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código de forma sencilla y rápida sin necesidad de una compleja configuración previa. Junto a este microframework se utilizaron varias librerías: flask-restful [21] que es una extensión para flask que permite crear API`s REST de forma rápida y flask-mongoengine [22] que también es una extensión que ofrece soporte para MongoEngine [23] para poder manejar las conexiones con la base de datos. MongoEngine es un DOM (Document-Object Mapper) que permite a aplicaciones basadas en Python trabajar con mongoDB, que es el sistema de base de datos que utilizaremos en el proyecto.

4.2 Servicios REST de la aplicación

4.2.1 Subir una imagen y procesarla

Para subir una imagen y que la red neuronal pueda procesarla se debe llamar al servicio de la API que permite mediante una petición POST HTTP guardar una imagen a la base de datos y añadirla en la cola para posteriormente realizar la predicción (figura 14).

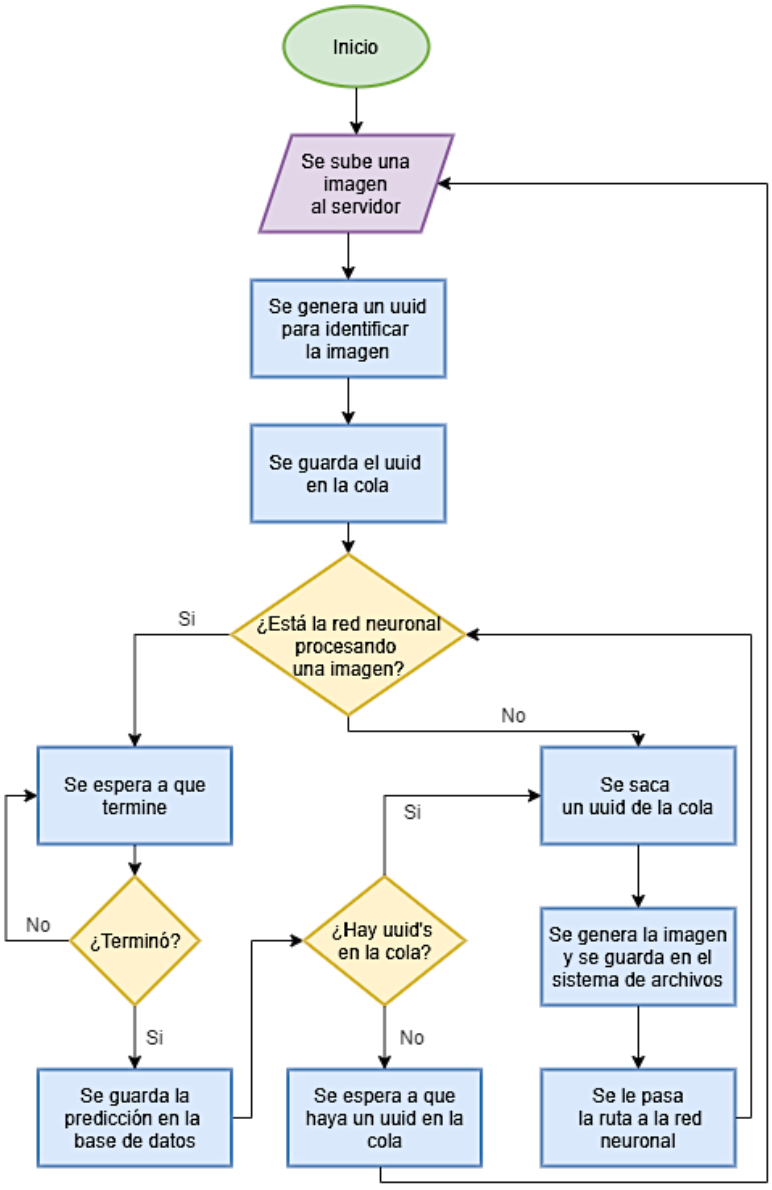


Figura 14: Diagrama de funcionamiento de la cola y la red neuronal.

La lógica que sigue el endpoint se basa en recibir un JSON que contiene, entre otros datos, la imagen, esta se almacena en la base de datos mediante un método auxiliar que se encarga de ello, que genera y devuelve un uuid (universally unique identifier) que identifica a la imagen, el cual es introducido en la cola a la espera de ser procesada por la red neuronal. Este endpoint recibe un objeto JSON con los siguientes campos:

- Las coordenadas de recorte de la imagen.
- el nombre de la imagen.
- el ancho y alto de la imagen según el tamaño de la pantalla del dispositivo.
- la imagen en base64.

Como respuesta, el servidor devuelve un código de estado de la petición junto a un objeto JSON que puede contener el uuid de la imagen o un mensaje de error que informa del tipo de fallo que ha ocurrido (figura 15).

```python
class process(Resource):
  def post(self):
    methods=['POST']

    if not request.method == 'POST':
      return {'message':'Invalid method'}, 405

    if not request.is_json:
      return jsonify({"message": "Missing JSON in request"}), 400

    data = request.get_json()
    imageName = data['imageName']

    if not allowedFileExtension(imageName):
      return {'message':'Invalid extension file'}, 406

    image_uuid = saveImage(data)
    insertInQueue(image_uuid)

    return Response(response={image_uuid}, status=200)
```

Figura 15: Servicio de la API que gestiona la imagen

4.2.2 Obtener el resultado de la red neuronal

Este servicio es llamado mediante peticiones GET HTTP realizadas de manera recursiva por el cliente hasta que es capaz de devolver el resultado almacenado en la base de datos por la CNN (convolutional neural network). Esto se explica con más detalle en el apartado 6.3.

El endpoint recibe como parámetro mediante la url el uuid que identifica a la imagen, y busca en la base de datos si existe una predicción asociada a ella. Si no existe, devuelve un JSON (figura 16) con el estado de la petición y el valor None, el cual representa el null en python. En caso de existir una predicción, la devuelve en un JSON.

```
class getDiagnosis(Resource):
  def get(self, uuid):
    methods=['GET']

    if request.method == 'GET':
      image = Image.objects.get(uuid=uuid).to_json()

      imageDict = json.loads(image)

      if ("diagnosisResult" in imageDict):
        response = jsonify(
          diagnosisResult = imageDict['diagnosisResult']
        )
        return response
      else:
        return Response(None, status = 200)

    return {'message':'Invalid method'}, 405
```

Figura 16: Servicio de la API que devuelve el resultado de la CNN

4.2.3 Obtener el recorte realizado en la imagen por el usuario

Este servicio sirve para obtener el recorte de la imagen que fue procesado por la CNN.

Para invocar al servicio se usa una petición GET HTTP que recibe mediante la url el uuid de la imagen, coge los datos de la base de datos y los

pasa a un método que se encarga de recortar la imagen y finalmente responde devolviendo un JSON con la imagen recortada en un string y el código de estado HTTP correspondiente (figura 17).

```
class downloadImage(Resource):
  def get(self, uuid):
    methods=['GET']

    if request.method == 'GET':
      image = Image.objects.get(uuid=uuid).to_json()
      imageDict = json.loads(image)

      croppedImage = cropImage(imageDict)
      image = croppedImage.decode("utf-8")

      return Response(response={image}, status=200)

    return {'message':'Invalid method'}, 405
```

Figura 17: Servicio de la API que devuelve la imagen recortada

4.3 Métodos para el procesamiento y gestión de las imágenes

Para desarrollar el funcionamiento completo de la aplicación, se han implementado una serie de utilidades extras para el correcto desempeño de la misma. Las más importantes se nombran a continuación.

4.3.1 Guardar una imagen en la base de datos.

Al recibir la imagen en el endpoint a través de un JSON la imagen ya no era de tipo base64 sino que pasaba a string, además también se eliminan las cabeceras añadidas en el Front-end, por lo que no era posible almacenarla en la base de datos ya que en su modelo se había especificado que la imagen se guardaría en un campo binario, para que resultara más fácil leer la imagen y no tener la necesidad de estar transformándola cada vez que se leyera de la base de datos. Por lo que se creó un método que recibe el JSON,

transforma la imagen de string a base64 y la guarda en la base de datos junto al resto de datos, retornando finalmente el uuid de la imagen (figura 18).

```
def saveImage(data):
  b64OriginalImage = stringToB64(data['b64Image'])
  image_uuid = uuid()

  Image(
    uuid = image_uuid,
    b64Image = b64OriginalImage,
    cropData = data['cropData'],
    resizeWidth = data['imageWidth'],
    resizeHeight = data['imageHeight']
  ).save()

  return image_uuid
```

Figura 18: Método que guarda la imagen en la base de datos.

4.3.2 Recortar la imagen

Este método (figura 19) tiene como función recortar una imagen en base a unas coordenadas generadas en la interfaz de la aplicación, las cuales delimitan el área que se desea de la imagen original. Para ello como parámetro recibe un diccionario que contiene:

- Las coordenadas x e y que representan el vértice superior izquierdo del área de selección (figura 4).
- La longitud de los lados del área.
- El ancho y largo que tenía la imagen en el navegador.
- La imagen en base64.

Para realizar el recorte de la imagen se llevan a cabo los siguientes pasos:

- Reescalamos la imagen al tamaño que tenía en el navegador ya que su tamaño varía según la resolución del dispositivo, y las coordenadas de recorte son relativas al tamaño del dispositivo y no de la imagen.

- Transformamos la imagen reescalada en un array de bytes para obtener la tupla que contiene el área de imagen que deseamos.
- Una vez obtenido el recorte se transforma a ndarray para codificarlo en base64 y retornarlo.

```
def cropImage(data):

  x = round(data['cropData']['x'])
  y = round(data['cropData']['y'])
  crop_width = int(data['cropData']['width'])
  crop_height = int(data['cropData']['height'])
  width = data['resizeWidth']
  height = data['resizeHeight']

  base64Image = base64.b64decode(data['b64Image']['$binary'])

  dim = (width, height)

  im_arr = np.frombuffer(base64Image, dtype=np.uint8)
  img = cv2.imdecode(im_arr, flags=cv2.IMREAD_COLOR)

  image_resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)
  image_cropped = image_resized[y:(crop_height+y), x:(crop_width+x)]

  retval, buffer = cv2.imencode('.png', image_cropped)
  b64Image = base64.b64encode(buffer)

  return b64Image
```

Figura 19: Método que obtiene el recorte de la imagen.

4.3.3 Guardar el recorte

La red neuronal necesita la imagen en un fichero y ciertos requisitos más para poder procesarla, por lo que este método hace exactamente eso.

Esta función recibe la imagen recortada en base64, luego crea el path donde se guardará temporalmente mientras la usa la red neuronal, para ello genera un nombre único para la imagen y la cual guarda en el sistema de archivos, retornando la ruta donde está almacenada (figura 20).

```
def saveCrop(b64croppedImage):
  path = os.path.join(Config.TEMP_FOLDER, generateRandomString())
  image_path = path + '/image/'
  if not os.path.exists(image_path):
      try:
          os.makedirs(image_path)
      except OSError as e:
          if e.errno != errno.EEXIST:
              raise

  b64cImage = b64croppedImage
  newImageName = uuid('png')

  with open(os.path.join(image_path, newImageName), "wb") as new_file:
      new_file.write(base64.decodebytes(b64cImage))

  return path
```

Figura 20: Método para generar la imagen en el sistema de archivos.

5 Diseño e implementación de la base de datos

Para la base de datos se ha utilizado mongoDB ya que la aplicación solo cuenta con una tabla o en este caso una colección que no se relaciona con ninguna otra, por lo que se ha optado por este tipo de base de datos noSQL, además es open source y permite guardar estructuras de datos con un esquema dinámico parecidas a los JSON denominadas BSON [24], haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

En la base de datos se almacena el uuid, la imagen en base64, las coordenadas de la zona que se quiere recortar de la imagen, el tamaño de reescalado, el resultado del diagnóstico y una serie de metadatos sobre la imagen, además de la fecha de subida (figura 21).

En la actualidad el hecho de guardar la imagen en la base de datos o en una carpeta en el sistema de archivos y referenciarla mediante el path y un id es algo que tiene fragmenta a la comunidad [25]. En nuestro caso se decidió guardar la imagen en la base de datos ya que la imagen solo va a ser leída de ella cuando se muestra el resultado y cuando el administrador del servidor quiera descargarlas, por lo que no se ralentizarán las acciones de leer y escribir en la base de datos. Además, al estar las imágenes dentro de la base de datos, no pueden quedar huérfanas de ella si se corrompe, y las copias de seguridad incluyen automáticamente los archivos binarios, por lo que es más seguro que guardar en un sistema de archivos.

```
{
   "_id" : ObjectId("5f50ef11eb146559ee70de8d"),
   "uuid" : "kPQTLsKo5o4ldj8FtttJcCswzfhare",
   "b64Image" : { "$binary" : "/9j/4AAQSkZJRgABAgAAAQABAAD/2wBDAAEBAQEB...", "$type" : "00" },
   "metadata" : [],
   "date" : ISODate("2020-09-03T13:26:41.637Z"),
   "cropData" : {
      "x" : 25.0625,
      "y" : 207.203125,
      "width" : 160,
      "height" : 160,
      "unit" : "px",
      "aspect" : 1
   },
   "resizeWidth" : 619,
   "resizeHeight" : 602,
   "diagnosisResult" : 0.999932408332825
}
```


Figura 21: Modelo de la base de datos.

6 Implementación del servicio de Deep Learning y detalles de cómo se comunica con el resto del sistema

6.1 Librerías utilizadas

Para el desarrollo e implementación de la red neuronal y del servicio que la ofrece se usaron una serie de librerías para poder ejecutar el modelo y tratar las imágenes, de entre ellas cabe nombrar algunas.

OpenCV [26] es una librería para desarrollo para visión por computador y machine learning, en el proyecto la función que tiene es la de redimensionar y transformar la imagen de base64 a string y viceversa para poder ser recortada.

Tensorflow [3] es un ecosistema que contiene librerías y recursos que permiten desarrollar modelos de Deep Learning de manera sencilla y centralizada. En este caso vamos a usar la librería Keras [4], que de forma transparente utiliza tensorflow, para importar el tipo de red neuronal que vamos a utilizar para el proyecto además de algunas herramientas que necesitaremos para adaptarla para el caso de uso.

6.2 Información sobre la red neuronal

Para poder determinar la probabilidad de padecer glaucoma que tiene un sujeto a partir de su retinografía, se entrenó una red neuronal convolucional basada en la conocida red VGG-19 (figura 22), la cual cuenta con una profundidad de 19 capas con un tamaño de entrada para imágenes de 224x224 pixels. Posteriormente este modelo fue ajustado para poder

analizar imágenes de fondo de ojo siendo adaptada para clasificar las imágenes en base a dos clases de salida: normal y glaucoma (figura 23).

```
def get_base_model(network, input_size):
    input_tensor = Input(shape=(input_size, input_size, 3))

    if network == 'vgg19':
        base_model = VGG19(input_tensor=input_tensor, weights='imagenet', include_top=False)
    else:
        print('Network unknown')
        return ''

    return base_model
```

Figura 22: Inicialización del modelo base VGG19

```
def build_model(network, input_size):
    base_model = get_base_model(network, input_size)
    if base_model == '':
        print('Network unknown')
        return

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    predictions = Dense(2, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=predictions)
    return model
```

Figura 23: Definición del clasificador.

Para iniciar el modelo de la red junto al Back-end y usarla se crearon dos métodos generales, uno que inicializa el modelo y otro que recibe la ruta de la imagen y devuelve el resultado de la predicción (figura 24).

La primera función se encarga de establecer todos los parámetros y de cargar los pesos devolviendo el modelo compilado de la red neuronal (figura 25).

```
        def evaluate(model, path):
            results_path = WEIGHT_PATH + '/' + network
            input_size = INPUT_SIZE_BY_NETWORK[network]

            predictions = evaluate_network_results(model, results_path,
                                                   input_size, path, BATCH_SIZE)
            rmtree(path)

            return predictions
```

Figura 24: Función que devuelve la predicción

```
def init_model():

    results_path = WEIGHT_PATH + '/' + network
    # Check if weights file exists
    if not os.path.exists(results_path + "/" + 'fold_final_w_net.h5'):
        print('Weights file for network ' + network + ' does not exist')
        return 'error'

    input_size = INPUT_SIZE_BY_NETWORK[network]

    model = build_model(network, input_size)
    model.compile(RMSprop(lr=1e-5), loss='categorical_crossentropy', metrics=['accuracy'])
    model.load_weights(results_path + "/" + 'fold_final_w_net.h5')

    return model
```

Figura 25: Función de inicialización del modelo.

6.3 Comunicación de la red neuronal con el Back-end

Debido a que la red neuronal tarda en analizar una imagen, si se hiciera la predicción en el propio endpoint esto lo paralizaría impidiendo que otro usuario pueda subir su imagen hasta que la anterior se haya procesado. Para solucionar este problema se creó una función (figura 26) que se llama al levantar la aplicación, la cual inicializa el modelo de la red neuronal y crea un hilo que ejecuta un método que se encarga de la lógica implicada para generar la predicción. Esta función se encarga de sacar el id de una cola, coger la imagen de la base de datos y pasársela al método que realiza el crop

para después guardarla en el sistema de archivos de manera temporal, y finalmente pasarle la ruta a la red neuronal para realizar la predicción y guardar el resultado en la base de datos (figura 27).

```
def load_model():
    global model
    model = init_model()
    Thread(target=processImage,).start()
```

Figura 26: Función que inicia el modelo y crea el hilo.

```
def processImage():
    global queue
    with Lock():
        while True:
            id = queue.get()
            imageData = Image.objects.get(uuid=id).to_json()
            imageDict = json.loads(imageData)

            croppedImage = cropImage(imageDict)
            path = saveCrop(croppedImage)

            result = evaluate(model, path)
            Image.objects.get(uuid=id).update(diagnosisResult = result[0])
```

Figura 27: Función ejecutada por el hilo.

7 Despliegue en el servidor

Para el despliegue de la aplicación se crearon una serie de script para facilitar el proceso, además de utilizar tecnologías como docker [27] y docker-compose [28] para levantar el servicio API REST y la base de datos.

7.1 Script de despliegue

Para simplificar el despliegue en el servidor una vez estuvieran los archivos subidos a él, se creó un script que realizara esta tarea (Figura 28).

```
#!/bin/bash
if [ "${BASH_SOURCE[0]}" = "$0" ]; then
  echo "You should source this script, not execute it!"
  echo "  > usage: source run-docker-compose.sh" 1>&2
  exit 1
fi

if [ $# -eq 0 -o $# -eq 1 ]; then
  envpath=.env

  if [ -f $envpath ]; then
    export $(cat $envpath | grep -v '#' | xargs -d '\n')

    docker build -t web-backend . #change path to backend Dockerfile
    docker build -f Dockerfile-mongo -t web-mongo .  #change path to mongodb Dockerfile
    docker-compose up &
  else
    echo "Failed to create docker container, no .env file found" 1>&2
    return 1
  fi
else
  echo "You must introduce one name to database directory or none." 1>&2
  return 1
fi
```

Figura 28: Script para el despliegue

Para poder ejecutar el script correctamente se requiere que exista un fichero .env que contiene una serie de variables de entorno, relacionadas con la base de datos, en las cuales se especifican:

- El nombre de usuario de la cuenta de administración de la base de datos.
- La contraseña de la cuenta de administración.
- El nombre de usuario de la base de datos que será utilizado por la aplicación.
- La contraseña del usuario.
- El nombre de la base de datos.
- El puerto de conexión a la base de datos.
- El mecanismo de autenticación para acceder a la base de datos.

Seguidamente a la creación de las variables de entorno se ejecutan los comandos para generar las imágenes de Docker de los contenedores del Back-end y de la base de datos y finalmente se ejecuta docker-compose (figura 29), una herramienta que permite, de manera simple, conectar y crear los contenedores, habilitar puertos, volúmenes y demás configuración que requieran los contenedores.

7.2 Despliegue del Back-end

Para realizar esta parte del despliegue se creó una imagen de docker que contiene la versión de Python requerida para la ejecución de la aplicación, además se le indica que copie el archivo requirements.txt en el interior del contenedor e instale las dependencias y finalmente copie la aplicación (figura 30).

```
FROM python:3.6.12

WORKDIR /usr/src/app

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

RUN pip install --upgrade pip
COPY ./requirements.txt .
RUN pip install -r requirements.txt

COPY ./app /usr/src/app
```

Figura 30: Dockerfile del Back-end.

Luego de que la imagen sea creada por el script de despliegue, se le específica a docker-compose que una vez montado el contenedor que contiene el Back-end ejecute el comando: gunicorn --timeout 120 --bind 0.0.0.0:5000 app:app para ejecutar el servidor uWSGI, especificando que espere 120 segundos, antes de matar y reiniciar el worker que ejecuta el servicio API REST, por estar dormido en el tiempo que la red neuronal se inicia, y permitir así que la aplicación se ejecute.

7.3 Despliegue de la base de datos

Para el despliegue de esta parte del proyecto se siguió la misma mecánica que con el Back-end. Para crear y configurar la base de datos se creó un fichero Dockerfile (figura 31) en el que se especifica la imagen de mongoDB y se copia un script (figura 32) que se ejecuta en el instante en el que se crea el contenedor.

```
FROM mongo:4.0.19-xenial

COPY mongo-create-user.sh /docker-entrypoint-initdb.d/
```

Figura 31: Dockerfile de la base de datos.

```
mongo --authenticationDatabase admin -u ${MONGO_INITDB_ROOT_USERNAME} -p ${MONGO_INITDB_ROOT_PASSWORD} <<EOF

use ${MONGO_DATABASE_NAME};

db.createUser({
  user: "${MONGO_USER}",
  pwd: "${MONGO_USER_PASSWORD}",
  roles: [
    { role: "readWrite", db: "${MONGO_DATABASE_NAME}" }
  ],
});
EOF
```

Figura 32: Script para la creación de un usuario de mongoDB.

Este script accede como administrador en mongoDB y crea la base de datos con el nombre especificado en el archivo .env y crea un usuario con permisos de lectura y escritura para esa base de datos.

Las variables de entorno que se requieren para configurar la base de datos se envían al contenedor mediante el archivo de docker-compose especificando en el apartado environments las variables que se requieren. También en este archivo se especifican el volumen del host que está conectado con el directorio del contenedor en el cual se almacenan los ficheros de la base de datos (figura 29).

```yaml
version: '3.3'
services:
  db:
    image: web-mongo:latest
    volumes:
      - /gdw/database/mongodb:/var/lib/mongodb
    environment:
      - MONGODB=mongodb://${MONGO_USER}:${MONGO_USER_PASSWORD}@db:...
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_ROOT_USERNAME}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_ROOT_PASSWORD}
      - MONGO_INITDB_DATABASE=${MONGO_DATABASE_NAME}
      - MONGO_USER=${MONGO_USER}
      - MONGO_USER_PASSWORD=${MONGO_USER_PASSWORD}
      - MONGO_DATABASE_NAME=${MONGO_DATABASE_NAME}
    restart: always
    ports:
      - "27017:27017"
  web:
    image: web-backend:latest
    command: gunicorn --timeout 120 --bind 0.0.0.0:5000 app:app
    ports:
      - 8080:5000
    environment:
      - MONGODB=mongodb://${MONGO_USER}:${MONGO_USER_PASSWORD}@db:...
    volumes:
      - /gdw/tmp/:/temp
    depends_on:
      - db
```

Figura 29: Archivo de configuración de docker-compose.

8 Conclusiones y líneas futuras

8.1 Conclusión

El diagnóstico precoz del glaucoma es un problema de gran dificultad debido a la falta de evidencias claras que apoyen dicho diagnóstico y la necesidad de contar con especialistas con años de experiencia en el campo. Es por ello, que el uso de modelos de Deep Learning puede contribuir a facilitar esta labor si disponemos del número suficiente de datos etiquetados con los que entrenar estos sistemas.

Ahí es donde entra en juego la idea de este TFG que consiste, como se comentó en la introducción, en desarrollar una aplicación web que permita subir una imagen de una retinografía y de este modo, ofrecer una estimación diagnóstica a través de una red neuronal con el objetivo final de almacenar esta imagen y de clasificarla y generar, así, una base de datos con suficientes muestras como para entrenar futuras redes neuronales de forma más precisa y robusta.

Para ello he creado una aplicación totalmente funcional cumpliendo los puntos establecidos al comienzo del proyecto, de manera que sea fácil e intuitiva de usar. Desde un principio fue un reto ya que no poseía ningún conocimiento previo sobre Deep Learning ni programación en Python, pero durante el desarrollo fui aprendiendo y aumentando mi rango de conocimiento e interés sobre Deep Learning e inteligencia artificial. Por otro lado, me decanté por este TFG ya que el desarrollo web es uno de los campos que más llaman mi atención en informática, junto con Deep Learning e inteligencia artificial.

8.2 Líneas futuras

Durante el desarrollo del proyecto surgieron nuevas ideas que fueron

añadiendo más funcionalidades y profundidad a la aplicación.

Dejando para el futuro, mejorar ciertos aspectos, como controlar en mayor medida el acceso y uso de la aplicación, así como de las imágenes que se suben al servidor. Se establecerá un sistema de autenticación y registro, además de un panel de administración que permita de manera general ver las imágenes subidas a la base de datos y poder efectuar operaciones de eliminación o descarga entre otras.

9 Summary and Conclusions

9.1 Conclusion

The early diagnosis of glaucoma is a challenge of great difficulty due to the lack of clear evidence to support its diagnosis and the need for specialists with years of experience in the field. For this reason, the use of Deep Learning models can help facilitate this work if there is enough labeled data that can train these systems.

Here is where the idea of this TFG comes into play, since developing a web application that allows uploading an image of a retinography, makes possible a diagnostic estimate through a neural network. In fact, these images will be used with the ultimate goal of storing it and classifying it in order to generate a database with enough samples to train more precise and robust future neural networks.

To aim this, I have created a fully functional application complying with the points established at the beginning of the project, hence it is easy and intuitive to use. From the beginning it was a challenge since I hadn't had any previous knowledge about Deep Learning or Python programming yet, but during the development I learnt and increased gradually my range of knowledge and interest in this topic as well as in artificial intelligence. On the other hand, I chose this TFG since web development is one of the fields that attracts my attention the most in computing, along with Deep Learning and artificial intelligence.

9.2 Summary

During the development of the project, new ideas emerged that added more functionality and depth to the application.

Nevertheless, there are certain aspects that may be improved in the

future, such as controlling the access and use of the application, as well as the images that are uploaded to the server. An authentication and registration system should be established, as well as an administration panel that allows a general view of the images uploaded to the database and the ability to carry out deletion or download operations, among others.

10 Presupuesto

Debido a la naturaleza de este proyecto, solo se contarán las horas de documentación y desarrollo de la aplicación en cada apartado correspondiente. El tiempo y coste del entrenamiento de la red neuronal no se tendrán en cuenta ya que fueron aportados por los directores del proyecto.

Concepto	Coste por horas	Horas	Coste total
Documentación	20€	80	1600€
Desarrollo de la interfaz de usuario	20€	110	2200€
Desarrollo del servicio API REST	20€	90	1800€
Configuración de la base de datos	20€	30	600€
Desarrollo de los scripts y Dockerfiles para el despliegue	20€	5	100€
	Total	315	6300€

Tabla 1: Desglose del presupuesto.

Bibliografía

1. React – Una biblioteca de JavaScript para construir interfaces de usuario [Internet]. Es.reactjs.org. 2020 [cited 10 September 2020]. Available from: https://es.reactjs.org/
2. Welcome to Flask — Flask Documentation (1.1.x) [Internet]. Flask.palletsprojects.com. 2020 [cited 10 September 2020]. Available from: https://flask.palletsprojects.com/en/1.1.x/
3. TensorFlow [Internet]. TensorFlow. 2020 [cited 10 September 2020]. Available from: https://www.tensorflow.org/
4. Team K. Keras: the Python deep learning API [Internet]. Keras.io. 2020 [cited 10 September 2020]. Available from: https://keras.io/
5. CaptionBot [Internet]. captionbot.ai. 2020 [cited 10 September 2020]. Available from: https://www.captionbot.ai/
6. Fråga en hudläkare [Internet]. iDoc24 - Online Dermatologist. 2020 [cited 10 September 2020]. Available from: https://www.idoc24.com/
7. Automatic lung-cancer detection on scans of Computed Tomography with RadIO [Internet]. Medium. 2020 [cited 10 September 2020]. Available from: https://medium.com/data-analysis-center/automatic-lung-cancer-detecion-on-scans-of-computed-tomography-with-radio-945d781aa022
8. Efficient, Safe and Fast eye screenings for eye specialists [Internet]. Retinalyze.com. 2020 [cited 10 September 2020]. Available from: https://www.retinalyze.com/
9. Nginx [Internet]. Es.wikipedia.org. 2020 [cited 10 September 2020]. Available from: https://es.wikipedia.org/wiki/Nginx
10. Transferencia de Estado Representacional [Internet]. Es.wikipedia.org. 2020 [cited 10 September 2020]. Available from: https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional

11. Green Unicorn (Gunicorn) [Internet]. Fullstackpython.com. 2020 [cited 10 September 2020]. Available from: https://www.fullstackpython.com/green-unicorn-gunicorn.html
12. TypeScript [Internet]. Typescriptlang.org. 2020 [cited 10 September 2020]. Available from: https://www.typescriptlang.org/
13. react-image-crop [Internet]. npm. 2020 [cited 10 September 2020]. Available from: https://www.npmjs.com/package/react-image-crop
14. react-dropzone [Internet]. npm. 2020 [cited 10 September 2020]. Available from: https://www.npmjs.com/package/react-dropzone
15. axios [Internet]. npm. 2020 [cited 10 September 2020]. Available from: https://www.npmjs.com/package/axios
16. Emotion [Internet]. 2020 [cited 10 September 2020]. Available from: https://www.npmjs.com/package/@emotion/react
17. react-router-dom [Internet]. npm. 2020 [cited 10 September 2020]. Available from: https://www.npmjs.com/package/react-router-dom
18. react-router [Internet]. npm. 2020 [cited 10 September 2020]. Available from: https://www.npmjs.com/package/react-router
19. Interfaz de programación de aplicaciones [Internet]. Es.wikipedia.org. 2020 [cited 10 September 2020]. Available from: https://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones
20. Lawley R. mongoengine [Internet]. Mongoengine.org. 2020 [cited 10 September 2020]. Available from: http://mongoengine.org/#home
21. Flask-RESTful — Flask-RESTful 0.3.8 documentation [Internet]. Flask-restful.readthedocs.io. 2020 [cited 10 September 2020]. Available from: https://flask-restful.readthedocs.io/en/latest/index.html
22. Flask-MongoEngine — Flask-MongoEngine 0.9.5 documentation [Internet]. Docs.mongoengine.org. 2020 [cited 10 September 2020]. Available from: http://docs.mongoengine.org/projects/flask-mongoengine/en/latest/

23. Lawley R. mongoengine [Internet]. Mongoengine.org. 2020 [cited 10 September 2020]. Available from: http://mongoengine.org/#home
24. BSON Types — MongoDB Manual [Internet]. Docs.mongodb.com. 2020 [cited 10 September 2020]. Available from: https://docs.mongodb.com/manual/reference/bson-types/
25. Which is Better ? Saving Files in Database or in File System | Habile [Internet]. Habile. 2020 [cited 10 September 2020]. Available from: https://habiletechnologies.com/blog/better-saving-files-database-file-system/
26. opencv-python [Internet]. PyPI. 2020 [cited 10 September 2020]. Available from: https://pypi.org/project/opencv-python/
27. Why Docker? | Docker [Internet]. Docker. 2020 [cited 10 September 2020]. Available from: https://www.docker.com/why-docker
28. 3. Overview of Docker Compose [Internet]. Docker Documentation. 2020 [cited 10 September 2020]. Available from: https://docs.docker.com/compose/