

Tecnológico Nacional de México Campus Culiacán

Aplicación móvil para reconocimiento de placas
vehiculares

Tópicos de Inteligencia Artificial

Profesor:

Zuriel Dathan Mora Felix

Integrantes:

Aguilar Recio Jesús Octavio

Echeagaray Aceves Astrid Monserrath



Repositorio GitHub:

<https://github.com/OctavioAR/TOPICOS-IA.git>

Link del Video de pruebas:

<https://goo.su/sr5EA2>

Índice

1. Introducción	3
2. Objetivos	3
2.1. Objetivo general	3
2.2. Objetivos específicos	3
3. Justificación	4
4. Alcance	4
5. Desarrollo	5
5.1. Documentación de instalación de dependencias	5
5.1.1. Estructura de carpetas y archivos	5
5.2. Cómo ejecutar el programa en entorno de desarrollo	6
5.2.1. Paso 1:	6
5.2.2. Paso 2:	6
5.2.3. Paso 3:	7
5.3. Especificaciones técnicas	8
5.3.1. Modelo de Visión Artificial	8
5.3.2. Implementación API REST	11
5.3.3. Interfaz Grafica	13
5.3.4. Resultados de las pruebas	13
5.4. Manual de usuarios	14
5.4.1. Pantalla de Carga	14
5.4.2. Pantalla Inicial	14
5.4.3. Captura de Placa	15
5.4.4. Resultados de Búsqueda	16
5.4.5. Datos cargados	17
5.4.6. Casos de Error	18
6. Agenda	19
7. Conclusión	19

Índice de Imágenes

1.	Código QR para ejecutar la aplicación	6
2.	Aplicación Expo Go	7
3.	Arquitectura de la aplicación móvil	8
4.	Resultado de las epocas de entrenamiento	9
5.	Matriz de confusión	10
6.	Imagenes de validación	10
7.	Vista de la consola de la API	11
8.	Estructura de BD en Firebase	12
9.	Pantalla de carga de la aplicación	14
10.	Pantalla inicial de la aplicación	14
11.	Interfaz de captura de placa vehicular	15
12.	Resultado exitoso de búsqueda	16
13.	Pantalla de procesamiento	17
14.	Mensaje de error cuando no se encuentra la placa	18
15.	Mensaje de error cuando no esta en la BD	18
16.	Cronograma de actividades	19

1. Introducción

En los últimos años, los avances en la visión artificial y el aprendizaje profundo han ayudado a desarrollar soluciones capaces de interpretar imágenes y poder extraer información importante en tiempo real. Con esto, se ha impulsado el diseño de sistemas inteligentes con aplicaciones en la seguridad y control vehicular. Un ejemplo de esto, son los sistemas de detección automática de placas vehiculares, que se han convertido en una herramienta esencial para la identificación rápida y precisa de automóviles, facilitando procesos como la verificación de datos, control de accesos, monitoreo de estacionamiento, etc.

El presente trabajo presenta el desarrollo de un sistema basado en técnicas de visión artificial para la detección y reconocimiento de matrículas vehiculares mediante dispositivos móviles. El proyecto combina un modelo de detección entrenado con YOLO, que es un sistema de reconocimiento óptico de caracteres (OCR) para interpretar los datos alfanuméricos de las placas, y una API como enlace entre el usuario y el servidor para devolver la información almacenada en una base de datos. Todo esto se integra en una aplicación móvil desarrollada con Expo React Native, que permite capturar la imagen de una placa y obtener los datos del vehículo y su propietario.

2. Objetivos

2.1. Objetivo general

Desarrollar un modelo de visión artificial, e implementarlo en una aplicación móvil, capaz de detectar de forma automática placas vehiculares en tiempo real, recibiendo información del dueño de dicho vehículo.

2.2. Objetivos específicos

- Implementar un modelo de entrenamiento específicamente para la detección de placas.
- Incorporar un OCR para el reconocimiento de caracteres alfanuméricos de la placa.
- Crear una API para la comunicación cliente-servidor.
- Integrar todo en una aplicación móvil para detectar placas.

3. Justificación

La automatización de procesos con IA se ha convertido en una necesidad para muchas industrias, entre ellas, las relacionadas con el control vehicular. La detección y reconocimiento de placas vehiculares es una de las tareas más relevantes, en estos contextos, ayuda a identificar automóviles de manera rápida y confiable sin intervención humana. De acuerdo con el artículo A survey of license plate recognition algorithms, los sistemas de reconocimiento de placas forman parte fundamental de los sistemas inteligentes de transporte, ya que “permiten identificar un vehículo de manera automática, eficiente y con mínima interacción humana”. Esto evidencia la importancia de implementar soluciones tecnológicas que hagan más accesible y práctico el reconocimiento vehicular. Por otra parte, el artículo Automatic License Plate Recognition (ALPR): A State-of-the-Art Review (2013) destacan que “los sistemas ALPR dependen críticamente de la calidad de la imagen y de la robustez del algoritmo de detección”, lo cual justifica el uso de modelos modernos como YOLO, que han demostrado ser muy eficientes en escenarios complejos. Por lo que, el uso de un modelo especializado para detección de objetos se vuelve esencial para garantizar precisión en condiciones reales. En este contexto, el desarrollo de una herramienta capaz de detectar placas vehiculares de manera automática no solo responde a necesidades actuales del sector, sino que también aprovecha los avances recientes en visión artificial para ofrecer una solución más rápida, precisa y adaptable.

4. Alcance

El alcance del presente proyecto se centra en desarrollar un modelo de visión artificial capaz de detectar el número de la placa vehicular de algún auto y hacer una consulta a una base de datos para ver la información del propietario de dicho automóvil (modelo, marca, color, año del vehículo, propietario del vehículo). Por lo tanto, el alcance incluye el desarrollo de dicho modelo de visión artificial, el desarrollo de una base de datos que contenga por lo menos una tabla propietarios y vehículos, también el desarrollo de una API para llevar a cabo la comunicación entre el usuario y el servidor, mandando una foto de alguna placa y el servidor la recibe para procesarla con el modelo y OCR para el reconocimiento de caracteres alfanuméricos, para después hacer una consulta a la base de datos y devolver la información al usuario de la aplicación. Además, el proyecto se limita a mostrar aquellas placas que están ingresadas en la BD, además que no cuenta con una interfaz para agregar placas desde la aplicación, ya que se enfocó en aprender a entrenar un modelo de visión artificial y poder implementarlo en un sistema.

5. Desarrollo

5.1. Documentación de instalación de dependencias

5.1.1. Estructura de carpetas y archivos

Para poder realizar el proceso de instalación del proyecto primero hay que entender la estructura de carpetas del proyecto. El proyecto de aplicación de detección de placas de vehículos consta de tres carpetas principales y en cada una de ella se emplean lenguajes y tareas diferentes.

API / Backend con Python:

- app
 - Modelos
 - modeloYolo.py
 - Servicios
 - servicioFirebase.py
 - servicioOcr.py

■ models

- best.pt

■ firebase-credentials.json

■ run.py

DetectorPlacas/ Frontend móvil con Expo React Native:

- app
 - tabs
 - camara.tsx
 - index.tsx

ModeloCNN/ Entrenamiento del modelo:

- entrenar_modelo.py
- preparar_datos.py
- imágenes placas.zip
- runs.zip (aquí se encuentra el modelo best.pt)

Ahora entendiendo la estructura de carpetas podemos pasar al proceso de instalación de las dependencias necesarias para que el proyecto funcione. Ubicados en la carpeta raíz de API se tienen que instalar las siguientes dependencias: fastapi, firebase-admin, ultralytics, opencv, numpy, pandas, easyocr. Ahora en la carpeta raíz de DetectorPlacas instalas las dependencias de npm y con el comando `npx expo start` inicias la app. Por último, en la carpeta ModeloCNN tener instalado ultralytics la cual cuenta con el modelo de entrenamiento de YOLO.

5.2. Cómo ejecutar el programa en entorno de desarrollo

5.2.1. Paso 1:

Para probar la aplicación en entorno de desarrollo, se tienen que hacer algunas cosas específicas para que puedas ejecutarla la app. Para empezar, tienes que configurar la dirección IPv4 de tu API y aplicación de expo. En el símbolo de sistema (CMD) revisas cuál es tu dirección IPv4 con el comando `ipconfig`. Ya que tengas la IP ingresas al archivo `index.tsx` ubicado en la carpeta de la app `DetectorPlacas`, después localizas la variable “`apiURL`” la cual tiene la dirección del servidor donde se encuentra la API y lo único es que cambias la dirección IPv4 por la tuya.

5.2.2. Paso 2:

Ahora tienes que levantar el servidor y la aplicación de expo. Para levantar el servidor ingresas a la carpeta raíz llamada `API` y ejecutas en la terminal el siguiente comando “`Python run.py`”. Después el servidor empezará a iniciarse y a inicializar el modelo, el ocr y firebase cuando en la consola salga el mensaje “`INFO: Application startup complete`” eso indica que ya puedes hacer peticiones al servidor sin ningún problema.

Para levantar la aplicación de expo, es un proceso parecido, primero ingresas a la carpeta raíz de `DetectorPlacas` y en la terminal ejecutas el comando “`npx expo start`” lo cual iniciará el proceso de carga de la aplicación y mostrará la lista de comandos como se muestra en la figura 1 los cuales puedes hacer uso de ellos durante el proceso de depuración.

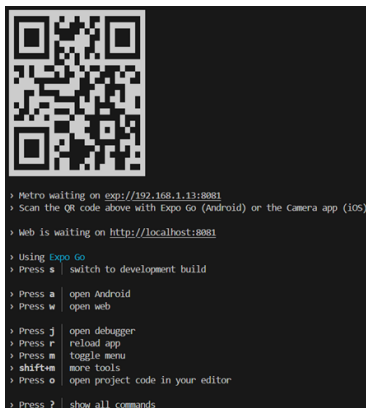


Figura 1: Código QR para ejecutar la aplicación

5.2.3. Paso 3:

Para utilizar la aplicación como entorno de desarrollo, expo te permite dos opciones. La primera sería que si en tu computadora tienes instalado algún emulador de Android por ejemplo el emulador del Android studio solamente con presionar la tecla “a” el emulador abrirá la aplicación. Pero si no tienes instalado ningún emulador en tu computadora, otra muy buena opción sería depurar con tu propio teléfono móvil, lo único que tienes que hacer es descargar la aplicación de “Expo Go” como se muestra en la figura 2 la cual lo puedes encontrar en PlayStore o AppStore. Una vez ya instalado en tu dispositivo abres la aplicación y escaneas el código QR que se muestra en la figura 1 y en automático empezará la depuración en tu celular.

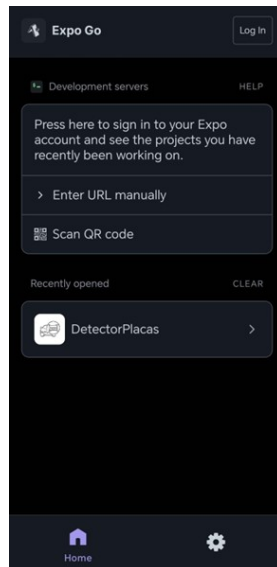


Figura 2: Aplicación Expo Go

5.3. Especificaciones técnicas

El sistema de reconocimiento de placas vehiculares sigue una arquitectura modular compuesta por tres componentes principales como se muestra en la figura 3 : Frontend Móvil, Backend (API) y modelo de entrenamiento. Es por eso que en este apartado se explican los detalles técnicos de cada módulo de la arquitectura.

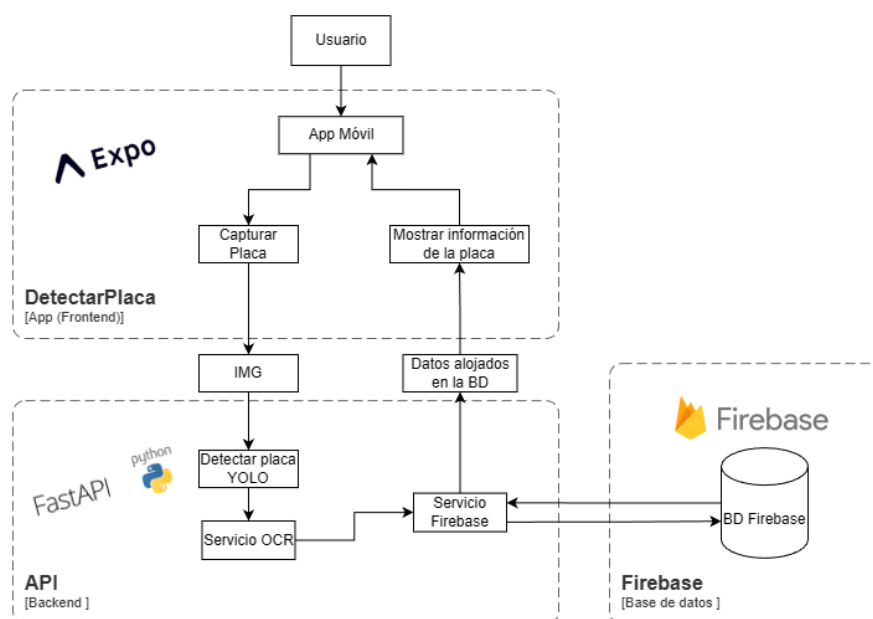


Figura 3: Arquitectura de la aplicación móvil

5.3.1. Modelo de Visión Artificial

Para todo modelo de visión artificial se debe de realizar una selección de imágenes etiquetadas y preprocesadas para poder someterlas después a un entrenamiento de red neuronal convolucional (CNN). Para este proyecto se utilizó un conjunto de imágenes de placas vehiculares de México, obtenidas de la página de RoboFlow. Para la cuestión del preprocesamiento de las imágenes, con el fin de volver más robusto el modelo; la misma página RoboFlow permite agregarles un poco de ruido a las imágenes, lo cual hicimos agregando un poco de aumento en la escala de grises con el fin que pueda detectar en espacios con poca luz y también un poco de rotación para prevenir la captura de las placas desde ángulos un poco complejos.

El modelo que se escogió para nuestro proyecto fue YOLOv8, ya que a diferencia de otros modelos como PyTorch, TensorFlow, etc. Este tiene una ar-

arquitectura CNN optimizada ofreciendo un equilibrio superior entre velocidad y precisión, además de brindar un entorno de desarrollo más amigable. En nuestro proyecto primero preparamos los datos con nuestro archivo “prepararDatos.py” aplicando unas transformaciones de coordenadas al formato YOLO requerido. Esta transformación permitió normalizar las coordenadas de los cuadros delimitadores ayudando así al proceso de aprendizaje del modelo, estos cálculos de coordenadas los efectúa la función “formatoYolo()”. Además, que con la generación automática del dataset pudimos organizar los datos en entrenamiento, validación y pruebas.

Para el apartado de entrenamiento con la lógica del archivo “entrenarModelo.py”, lo primero que hicimos fue utilizar la función “generarArchivoYaml()” para crear el archivo de configuración YAML que utiliza el modelo YOLOv8, este archivo actúa como plano de configuración esencial para definir tanto la arquitectura del modelo como los parámetros del conjunto de datos, por eso mismo la función divide el conjunto de imágenes en train, valid y test en la que cada una tiene sus imágenes y etiquetas. Después de generar este archivo YAML podemos entrenar el modelo, para eso creamos la función “entrenarModelo()” la cual inicializamos el modelo con “modeloYolo = YOLO(yolov8n.pt)” para dar inicio al entrenamiento y con la función “modeloYolo.train()” definimos los parámetros de entrenamiento, cargamos el archivo de configuración YAML, definimos las épocas o ciclos de entrenamiento las cuales fueron 35 ya que no contamos con un equipo de computo poderoso, el tamaño de las imágenes fue de 640x640 y definimos la dirección y nombre del modelo “best.pt”. Durante el proceso de entrenamiento como se puede observar en la figura 4, demostró un aprendizaje estable con mejoras en las métricas a lo largo de las épocas. Además, que la estabilidad de las pérdidas en estas últimas épocas alcanzó un punto óptimo de entrenamiento. Como se puede observar al llegar en la última época la precisión de detección llego a 0.993 lo que indica que el 99.3 % de las detecciones realizadas por el modelo corresponden a placas vehiculares y el recall obtuvo el valor de 1.0 lo que indica que el modelo fue capaz de detectar todas las placas presentes en las imágenes de validación.

Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size	
32/35	0G	0.3717	0.4616	0.8288	10	640: 100%	14/14 3.5s/it 48.4s
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95): 100%	1/1 1.4it/s 0.7s
	all	9	9	0.993	1	0.995 0.877	
Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size	
33/35	0G	0.3723	0.4655	0.8194	11	640: 100%	14/14 3.5s/it 48.8s
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95): 100%	1/1 1.3it/s 0.7s
	all	9	9	0.993	1	0.995 0.849	
Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size	
34/35	0G	0.3524	0.4489	0.8204	10	640: 100%	14/14 3.4s/it 48.2s
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95): 100%	1/1 1.3it/s 0.7s
	all	9	9	0.993	1	0.995 0.863	
Epoch	GPU_mem	box_loss	cls_loss	dfl_loss	Instances	Size	
35/35	0G	0.3438	0.444	0.8092	11	640: 100%	14/14 3.4s/it 47.6s
	Class	Images	Instances	Box(P)	R	mAP50 mAP50-95): 100%	1/1 1.3it/s 0.8s
	all	9	9	0.993	1	0.995 0.865	

Figura 4: Resultado de las épocas de entrenamiento

El propio modelo de YOLO genera una matriz de confusión como se observa en la figura 5 mostrando que el modelo logró una precisión del 100 % en la clase “placas” sin generar ningún falso positivo. Para la clase “background” la precisión fue de 80 % mostrando que tiene capacidad de distinguir entre regiones que tienen placas y aquellas que no.

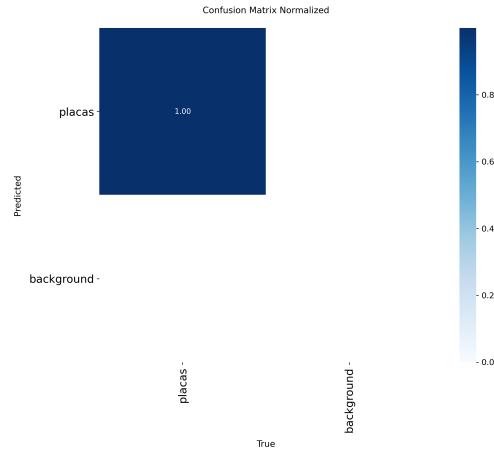


Figura 5: Matriz de confusión

Por ultimo las imágenes de validación 6 generadas muestran que el modelo puede detectar correctamente placas en distintas regiones, direcciones, ángulos, etc.

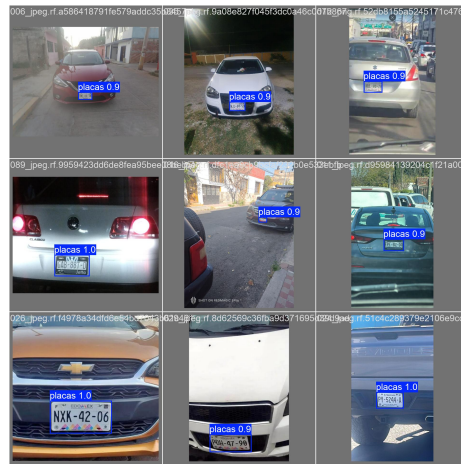


Figura 6: Imagenes de validación

5.3.2. Implementación API REST

Para el Backend de la aplicación móvil para el reconocimiento de placas se pensó primero en cargar simplemente el modelo de entrenamiento directamente en la aplicación móvil, pero no encontramos ningún modelo compatible con Expo, entonces decidimos realizar una lógica de cliente/servidor, para esto primero pensamos en realizarlo con Node.js, pero como queremos utilizar solamente una API para cargar el modelo, servicio de OCR y servicio de BD, se decidió hacerlo mediante una FastAPI con Python ya que este lenguaje cuenta con las librerías necesarias para temas de visión artificial y el uso de modelos.

Primero vamos a hablar sobre el servidor principal alojado en el archivo “main.py” donde se encuentran las direcciones específicas que reciben las solicitudes para acceder a los recursos del servidor. Tenemos el evento “startup” en donde el servidor al arrancar inicializa todos los servicios disponibles dentro del mismo como se muestra en la figura 7, como por ejemplo el modelo de detección de YOLO, el servicio de OCR para el reconocimiento de texto y el servicio de cliente para hacer consultas a la base de datos no relacional alojada en Firebase. La principal ventaja de inicializar los modelos al arrancar el servidor es que los modelos pesados se cargan una sola vez en memoria, a diferencia que se tengan que inicializar cada vez que se fuera hacer una llamada al servidor. La configuración del servidor incluye también middleware CORS la cual actúa como intermediario para que la aplicación por el lado del cliente pueda acceder a los recursos de nuestro servidor. El endpoint principal de nuestro servidor es “/detectar-placa” el cual por medio de la petición POST recibe la imagen tomada por el cliente, la procesamos para detectar la placa vehicular, extraemos su texto con el OCR y por último hacemos la consulta a la BD en Firebase.

```
INFO: will watch for changes in these directories: ['C:\\Users\\octav\\TOPICOS-IA\\UNIDAD 4\\API']
INFO: uvicorn running on http://0.0.0.0:8000 (press CTRL+C to quit)
INFO: Started reload process [9676] using Starline
2025-11-28 16:12:41,961 - app.servicios.serviciofirebase - INFO - Firebase inicializado
INFO: Started server process [15596]
INFO: Waiting for application startup.
2025-11-28 16:12:41,963 - app.main - INFO - Iniciando servidor...
2025-11-28 16:12:41,964 - app.modelos.modeloYOLO - INFO - cargando modelo YOLO desde: C:\\Users\\octav\\TOPICOS-IA\\UNIDAD 4\\API\\models\\best.pt
2025-11-28 16:12:42,066 - app.modelos.modeloYOLO - INFO - Modelo YOLO cargado correctamente
2025-11-28 16:12:42,066 - app.main - INFO - Modelo YOLO cargado correctamente
2025-11-28 16:12:42,067 - easyocr.easyocr - WARNING - Using CPU. Note: This module is much faster with a GPU.
2025-11-28 16:12:43,621 - app.main - INFO - OCR inicializado correctamente
2025-11-28 16:12:43,621 - app.main - INFO - Firebase conectado correctamente
2025-11-28 16:12:43,622 - app.main - INFO - Servidor listo para recibir peticiones
INFO: Application startup complete.
```

Figura 7: Vista de la consola de la API

El archivo “modeloYolo.py” utiliza el framework Ultralytics YOLO para realizar en tiempo real los procesos de las imágenes recibidas. De manera sencilla podemos decir que la clase “DetectorPlacas” engloba toda la lógica de cargar el modelo entrenado al servidor, verificamos la existencia de dicho archivo y lo cargamos solamente una vez en memoria, preparándolo para procesar múltiples solicitudes de manera continua. La lógica es simple, el proceso detección recibe una imagen en formato numpy array y aplicamos a esta el modelo YOLO entrenado para identificar aquellas regiones que contengan la placa vehicular. Para cada detección extraemos sus coordenadas y el nivel de confianza.

Después tenemos el primer archivo de servicio dentro del servidor “servicioOCR.py” el cuál se encarga de extraer texto de las regiones de placas detectadas, utilizando las bibliotecas de EasyOCR, añadiendo procesos de pre-procesamiento de imagen para mejorar la precisión del reconocimiento de los caracteres alfanuméricos. Dentro del procesamiento OCR se implementó una serie de validaciones que evalúan los candidatos detectados según los patrones definidos para placas ejemplo: AAA-898-A, 32A-FR-4D, etc. Para esto utilizamos expresiones regulares para identificar aquellas cadenas que se identifiquen como placas validas, así evitando extraer nombres de estado, números telefónicos o cualquier otro carácter de la placa que no sea útil.

El ultimo servicio que tenemos es “servicioFirebase.py” el cual se encarga de gestionar y consultar a la base de datos alojada en Firestore, como se puede observar en la figura 8 contamos con dos colecciones “Propietarios” y “Vehículos”. Lo primero que hacemos es establecer la conexión durante el proceso de inicialización utilizando las credenciales de servicio almacenadas localmente “firebase-credentials.json”. El servicio no solamente maneja búsquedas por coincidencias exactas si no también es flexible a errores comunes de reconocimiento del OCR utilizando un diccionario de confusiones. Para las consultas en Firebase seguimos una serie de validaciones la primera sería encontrar la placa exacta con el texto extraído del OCR, luego generamos una variante del prefijo de la placa para considerar sustituciones de caracteres comunes y por último realizamos una búsqueda por similitud en todos los documentos de la colección. Realizando este enfoque podemos aumentar las probabilidades de encontrar el vehículo correcto incluso cuando el OCR comente errores de reconocimiento.

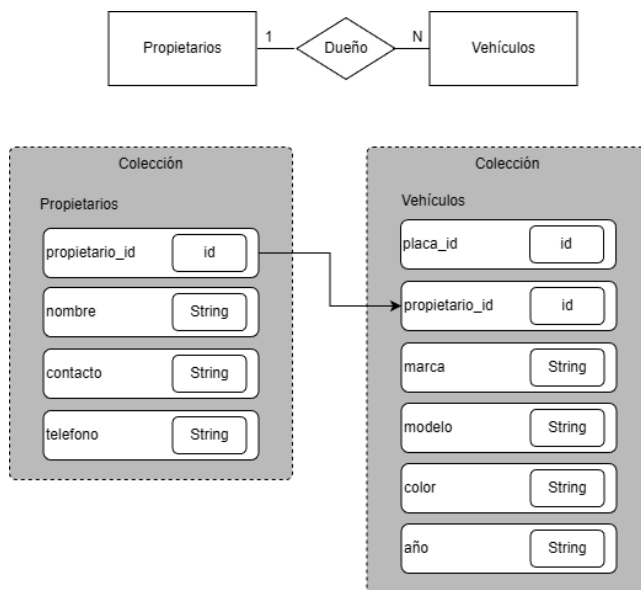


Figura 8: Estructura de BD en Firebase

5.3.3. Interfaz Grafica

Para el desarrollo del Frontend de la aplicación se utilizó React Native con Expo, ya que este framework es muy accesible ya que no pesa tanto como otros entornos por ejemplo Android Studio, además que cuenta con la facilidad de depurar la aplicación con tu mismo teléfono móvil utilizando la aplicación de Expo Go. La comunicación con el Backend se realiza mediante peticiones HTTP a la API REST y utilizando FormData el cuál es un objeto de JavaScript que permite construir datos y mandarlos a un servidor, en este caso las imágenes.

La estructura de navegación de la aplicación es sencilla ya que cuenta con dos estructuras. La primera es la pantalla inicial en el archivo “index.tsx” el cual sería el punto de entrada a la aplicación, con una interfaz clara y unas instrucciones de uso muy sencillas.

El segundo componente sería la cámara en el archivo “camara.tsx”. En dicho componente se efectúan tres tareas: capturar las imágenes de las placas, procesamiento y visualización de resultados. Además, en dicho componente se establecen una serie de validaciones y permisos, como por ejemplo para acceder a la cámara. La aplicación utiliza la función “ImageManipulator” de Expo permitiendo optimizar la imagen antes de enviarla al servidor, como por ejemplo reducir el tamaño del archivo. En las funciones “enviarImgApi” de envió se implementan las respuestas y manejos de errores. Por ultimo el flujo de captura y procesamiento sería: capturar usando la interfaz de la cámara, procesamiento enviando la imagen al servidor, validación en la que se aplican las reglas a la respuesta recibida, resultado se muestra información o mensaje de error y recuperación es la opción para nueva captura en caso de error.

5.3.4. Resultados de las pruebas

El video lo puede encontrar en la portada de este mismo documento y las capturas de resultados las puede observar también en el manual de usuario que se encuentra en la siguiente hoja 12, 13.

Link del Video de pruebas:

<https://goo.su/sr5EA2>

5.4. Manual de usuarios

5.4.1. Pantalla de Carga

Cuando abres la aplicación, lo primero que aparece es una pantalla de carga, en la que después de cierto tiempo aparecerá la pantalla inicial, así como se muestra en la figura 9.

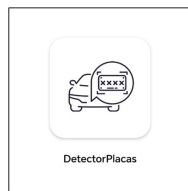


Figura 9: Pantalla de carga de la aplicación

5.4.2. Pantalla Inicial

Posteriormente aparece la pantalla inicial 10 de la aplicación en la que se le da la bienvenida a los usuarios y las instrucciones de uso de la aplicación.

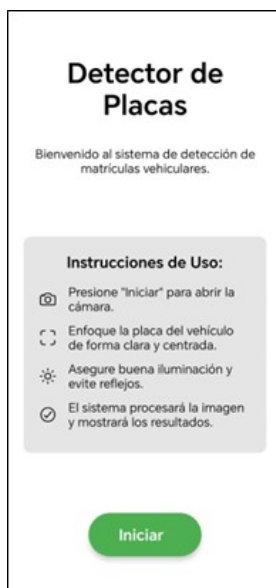


Figura 10: Pantalla inicial de la aplicación

5.4.3. Captura de Placa

Al presionar el botón inicial la aplicación te redirige a la cámara para poder capturar la placa vehicular.

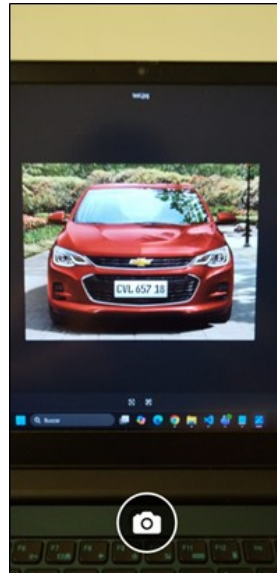


Figura 11: Interfaz de captura de placa vehicular

- **Botón de captura:** Presiona este botón para tomar la fotografía de la placa
- **Área de enfoque:** Mantén la placa dentro del área demarcada para mejor reconocimiento
- **Indicador de estado:** Muestra si la cámara está lista para capturar

5.4.4. Resultados de Búsqueda

Cuando capturas una matrícula y esta se encuentra integrada en la BD se muestra un mensaje 12 emergente que indica que se encontró la matrícula y muestra la información del propietario de dicho vehículo.

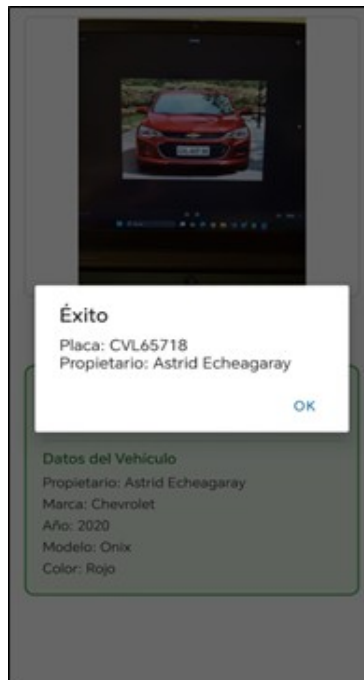


Figura 12: Resultado exitoso de búsqueda

Información mostrada:

- Número de placa detectado
- Nombre del propietario
- Modelo del vehículo
- Marca del vehículo
- Color del vehículo
- Año del vehículo

5.4.5. Datos cargados

Hay que esperar a que capture la información obtenida ya que hay ocasiones en las que tarda un poco más en extraer el contenido de la placa y hacer la consulta a la BD.

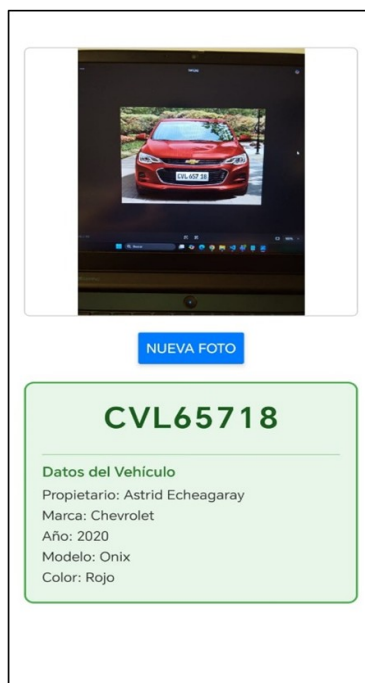


Figura 13: Pantalla de procesamiento

Nota: El tiempo de procesamiento puede variar dependiendo de:

- Calidad de la imagen capturada
- Claridad de los caracteres en la placa
- Condiciones de iluminación
- Velocidad de la conexión a internet

5.4.6. Casos de Error

Si la placa no se encuentra en la base de datos o no puede ser reconocida correctamente, se mostrará un mensaje de error.

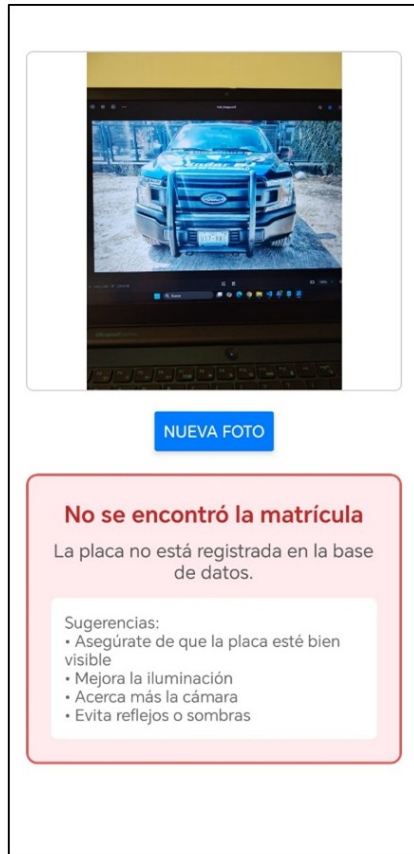


Figura 14: Mensaje de error cuando no se encuentra la placa

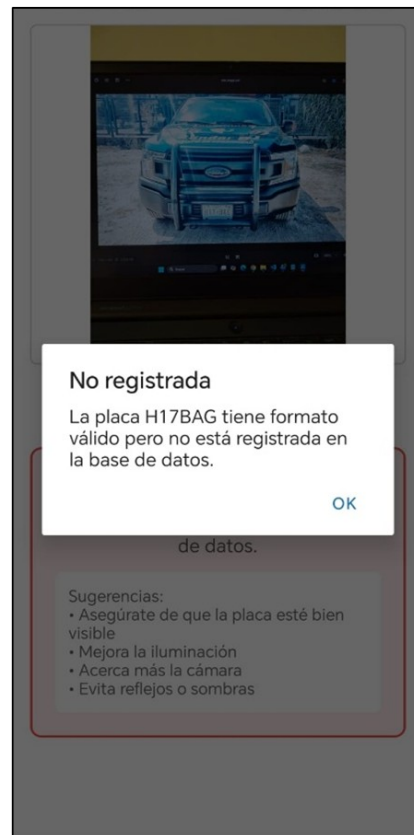


Figura 15: Mensaje de error cuando no esta en la BD

Possibles causas de error:

- Placa no registrada en la base de datos
- Imagen muy borrosa o fuera de foco
- Caracteres de la placa obstruidos o dañados
- Condiciones de iluminación insuficientes
- Problemas de conexión con el servidor

6. Agenda

El proyecto constó de 16 días para su elaboración, como se puede apreciar en la figura 16 la cual representa cómo es que se dividieron los días para poder realizar dicho proyecto, siendo los cuadros de color verde los días esperados para su elaboración y rojos los días en que se realizó.

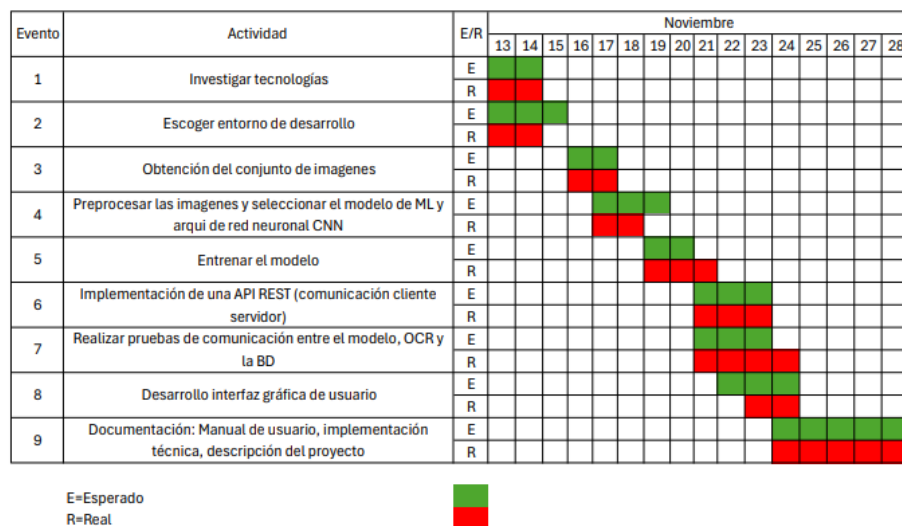


Figura 16: Cronograma de actividades

7. Conclusión

El desarrollo de este proyecto permitió comprender cómo las técnicas de visión artificial y los modelos de inteligencia artificial pueden aplicarse para identificar placas de manera rápida y precisa.

El uso de modelos modernos de detección de objetos facilita el procesamiento de imágenes en tiempo real y ofrece resultados confiables incluso en condiciones variables. Gracias a ello, es posible construir sistemas más eficientes que reduzcan la intervención humana y aumenten la precisión en tareas que antes eran manuales.

En general, este trabajo demuestra que integrar IA en el reconocimiento de placas es una solución práctica y viable, y sienta las bases para futuras mejoras como el reconocimiento de caracteres, la integración con bases de datos o la implementación en entornos más complejos.