Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Localizador de códigos de barras mediante Inteligencia Artificial

Barcode detector using Artifical Inteligence

Claudio Néstor Yanes Mesa

La Laguna, 8 de julio de 2022

D. **José Gil Marichal Hernández**, con N.I.F. 78.677.406-H, profesor contratado doctor adscrito al área Teoría de la Señal y Comunicaciones, del Departamento de Ingeniería Industrial de la Universidad de La Laguna, como tutor

D. **Óscar Gómez Cárdenes**, con N.I.F. 78.859.209-Y, doctor en Informática por la Universidad de La Laguna, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"*Localizador de códigos de barras mediante Inteligencia Artificial*"

ha sido realizada bajo nuestra dirección por D. **Claudio Néstor Yanes Mesa**, con N.I.F. 79.087.556-D.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 8 de julio de 2022.

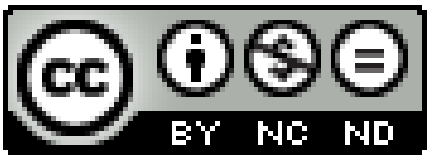
Agradecimientos

A la Universidad de La laguna por haberme provisto de una enseñanza de calidad que me ha permitido llegar hasta este punto.

A José Marichal y Óscar Gómez por haberme permitido participar en esta experiencia única; así como por su tiempo, ayuda y paciencia.

Finalmente a mi familia por apoyarme en el transcurso de mi vida académica.

Licencia



Resumen

Los códigos de barras se diseñaron como una forma de representar información corta alfanumérica en etiquetas que luego eran leídas por dispositivos electrónicos, nacidos a la par que estos: los escáner laser. La tarea de decodificación en sí de los códigos se diseñó para que fuera trivial, mientras que el avance tecnológico estaba en la aplicación de la fotónica para recuperar los anchos de las barras y huecos entre ellas.

No obstante, un rol muy importante en el proceso de lectura era dejado al usuario: la tarea de localización de los códigos de barras y alineación del lector con los mismos. Que a pesar de ser inmediato para un operario humano supone una tarea no despreciable para un sistema de visión por ordenador.

Este TFG se propone como reto que un dispositivo móvil sea capaz de detectar y marcar todos los códigos de barras presentes en el campo de visión de su cámara aprovechando los avances en machine learning*. Se ha partido de un banco de imágenes de códigos de barras. Estas imágenes han tenido que ser etiquetadas, para lo cual se ha desarrollado una herramienta etiquetadora semiautomática.*

Tras el proceso de etiquetado del banco de imágenes, se han entrenado con él una serie de modelos de machine learning*. Estos modelos se basaron en la red de R-CNN para las tarea de segmentación de instancia y para realizar segmentación semántica en DeepLabV3+ con variantes que usaban alternativamente ResNet o MobileNetV2 como backbone de la red. El modelo basado R-CNN obtuvo un AP de 0.7554 y un tiempo de inferencia promedio de 79.16 ms, bastante por encima de nuestro tiempo objetivo de 33 ms. Por otro lado, los modelos basados DeepLabV3+ consiguieron un AP de 0.8369 y 0.7856 (con ResNet y MobileNetV2 respectivamente) y un satisfactorio tiempo de inferencia promedio de 20 ms para todas las variantes.*

Luego estos modelos han sido utilizados en una aplicación de Android con el fin de evaluar su comportamiento en un teléfono móvil. En dicho dispositivo, los modelos basados en DeepLabV3+ lograron un tiempo de inferencia promedio 127.0 ms y 89.83 ms para las variantes utilizando ResNet y MobileNetV2 respectivamente, lo cual si bien no logra llegar al tiempo objetivo (33ms), sí que son resultados prometedores dada la plataforma de ejecución. Estos resultados se obtuvieron tras aplicar sobre los modelos el optimizador de TensorFlow Lite.

Los resultados de este TFG son en resumen: la herramienta etiquetadora semiautomática, varios modelos de machine learning para la detección de códigos de barras y una aplicación Android con el mismo fin.

Palabras clave: código de barras, machine learning, Segmentación semántica, reconocimiento de patrones, TensorFlow, Android.

Abstract

The barcodes were invented to represent and store short pieces of alphanumeric information in tags to be later consumed by machines. With them, the optical barcode reader was born. The barcode decoding logic was designed to be as simple as possible. What made this technology revolutionary was the use of photonics to discern the width of the bands and the gaps between them.

A key role was left to the user: The user was in charge of discovering the barcodes and aligning the reader with them. This task, trivial for a human, hides considerable complexity for a computer vision system.

This TFG proposes making a mobile device to detect and tag all barcodes in the range of vision of its camera using the latest advancements in machine learning. A dataset of images containing barcodes was used as a starting point. These images need to be correctly tagged, so a semiautomatic tagging tool was built to fulfill this task.

Once the dataset was tagged, it was used to train a number of machine learning models. The models intended, for instance sementation were built using R-CNN, while the ones intended for sematic segmentation were built using DeepLabV3+ instead. The DeepLabV3+ has two variations using two different backbones: ResNet and MobileNetV2.

The R-CNN based model obtained an AP of 0.7554 with a mean inference time of 79.16ms. This inference time fails to reach our target time of 33ms or less. Conversely, the models based on DeepLabV3+ reached an AP of 0.8369 and 0.7856 with a mean inference time of 19.52 ms and 20.11 ms for the backbones of ResNet and MobileNetV2 respectively.

These models were then used in an Android app to evaluate their behavior on a mobile phone. In the said mobile phone, the DeepLabV3+ archived a mean inference time of 127 ms for the ResNet variant and 89.83 ms for the MobileNetV2 variant. Even if these results do not reach our target time, we consider them quite promising. All models executed in Android were previously optimized with the TensorFlow Lite optimizer.

This TFG has produced a semiautomatic tagging tool, multiple machine learning models for barcode detection, and an Android app with the same purpose.

Keywords: barcodes, machine learning, sematic segmentation, pattern recognition, TensorFlow, Android.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Códigos de barras

Los primeros códigos de barras fueron patentados en los años 50 del pasado siglo (1), si bien se extendieron a las cadenas de *retail* con su forma actual en la década de 1970 (2). Existen múltiples tipos de código de barras, dividiéndose principalmente en códigos de barras unidimensionales (1D) o lineales y los códigos de barras bidimensionales (2D) o matriciales. En este TFG solo abordaremos los códigos de barras unidimensionales y de ahora en adelante será a lo que nos referiremos cuando hablemos de códigos de barras si no se especifica lo contrario. Dentro de estos grandes tipos de códigos de barras existen subtipos, denominados simbologías (3) siendo las predominantes las simbologías EAN/UPC y CODE-128, si bien no son relevantes para este TFG.

Figura 1.1: A la izquierda un código de barras unidimensional o lineal, y a la derecha un código de barras bidimensional o matricial.

Los primeros códigos de barras surgieron de extender verticalmente los códigos Morse. Aún hoy, los códigos lineales más elaborados siguen codificando la información en base a los anchos relativos de una serie de barras, por lo que es un problema muy acotado. La mayoría de lectores de códigos de barras analizan la imagen, la decodifican y transmiten la información que éste contenía simulando pulsaciones de teclado. Por supuesto, los teléfonos móviles también son capaces de tal tarea, existiendo ya en 2004 artículos que discuten la decodificación de código de barras en teléfonos móviles(4).

Sin embargo, la lectura de códigos de barras resulta tan sencilla para los dispositivos porque relegan la tarea de localizar los códigos de barras y alinear el lector con ellos

al usuario, ya que esta tarea se considera trivial para un humano, pero resulta en un problema de coste elevado para un computador. Si bien ésta ciertamente no es complicada para un humano, sí que puede transformarse en una fuente de frustración y pérdida de productividad cuando se tiene que hacer de forma reiterada. Además, esta dependencia al usuario limita la utilización de los códigos de barras en sistemas autónomos.

1.2. Aprendizaje automático y Segmentación semántica

El aprendizaje automático o *machine learning* es el subcampo de las ciencias de la computación y una rama de la inteligencia artificial, cuyo objetivo es desarrollar técnicas que permitan que las computadoras aprendan. Se dice que un agente aprende cuando su desempeño mejora con la experiencia y mediante el uso de datos; es decir, cuando la habilidad no estaba presente en su genotipo o rasgos de nacimiento (5).

La segmentación (de imágenes) es un problema del campo de la visión artificial. Consiste en categorizar cada píxel que compone una imagen, para poder realizar agrupaciones conocidas como segmentos. La segmentación semántica es uno de los casos más sofisticados de la segmentación, donde las categorías tienen un elevado significado, comúnmente se clasifican los objetos que aparecen en la imagen. En el contexto de este TFG solo se trabaja con dos categorías, “código de barras”, que evidentemente representa a los códigos de barras en la imagen y fondo o *background* que representa a todo lo demás. Existe además una variante de la segmentación semántica denominada segmentación de instancia, donde no solo se clasifican los objetos en la imagen, sino que también se distingue las unidades individuales de dichos objetos.

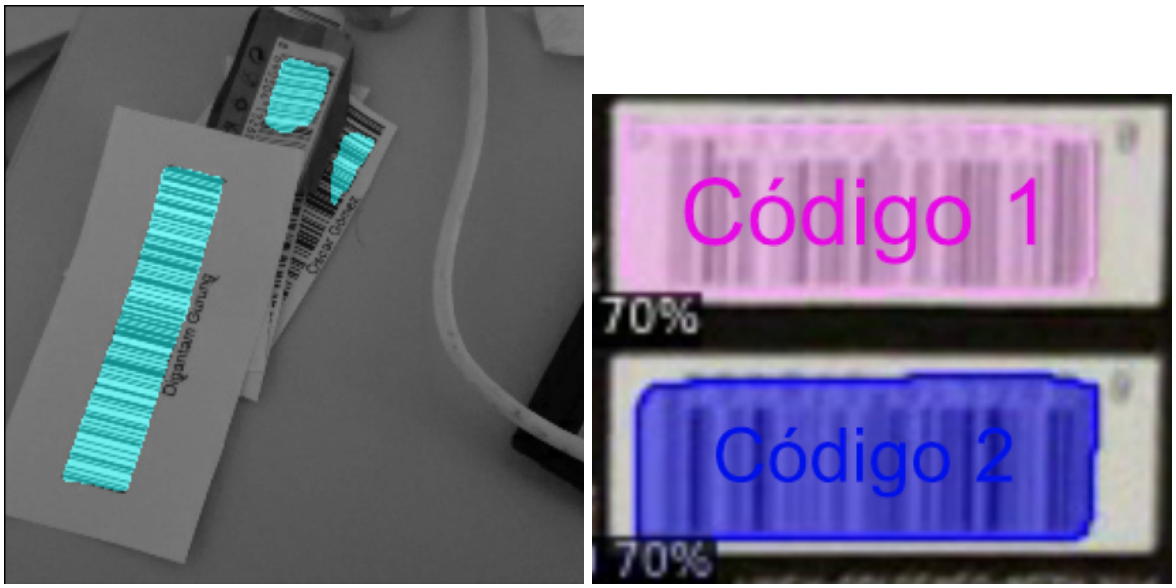


Figura 1.2: A la izquierda una imagen donde se ha aplicado segmentación semántica, los códigos de barras se han resaltado en cian. A la derecha una imagen donde se ha aplicado segmentación de instancia y se han indicado cada instancia individual con colores diferentes.

La aplicación de algoritmos de *machine learning* para la segmentación semántica produce excelentes resultados (6) y se ha transformado en la manera estándar de afrontar

esta tarea. El problema es que el entrenamiento de los modelos de *machine learning* normalmente requiere de un gran banco de ejemplos correctamente clasificado y etiquetados para poder realizar correctamente su entrenamiento. En Internet existen algunas colecciones de imágenes con códigos de barras con etiquetas de localización disponibles públicamente (7; 8), no obstante, dichas colecciones fueron tomadas con dispositivos con cámaras muy inferiores a las disponibles actualmente y no son adecuados para el correcto entrenamiento de nuestra IA (inteligencia artificial). Es por ello por lo que se decidió formar nuestra propia colección de fotos con códigos de barras, la cual fue etiquetada y utilizada en este TFG.

1.2.1. Evaluación de modelos para segmentación semántica

Es necesaria una métrica para poder comparar los resultados de los distintos modelos de segmentación semántica, pues dado el elevado número de imágenes a tratar es poco práctica la revisión manual de la salida del modelo para cada una. En este TFG se utilizara la métrica del AP (*Average Precision*) tal y como se define en *The PASCAL Visual Object Classes* (VOC) Challenge(9), sección 4.2.

Comenzamos por definir la precisión y la sensibilidad (*recall*). La precisión define cuan exactas son las predicciones, y matemáticamente se define como:

$$precisión = \frac{verdaderos\,positivos}{verdaderos\,positivos + falsos\,positivos}$$

Por otro lado, la sensibilidad indica cuan bien se han encontrado todos los positivos, la definición matemática es la siguiente:

$$recall = \frac{verdaderos\,positivos}{verdaderos\,positivos + falsos\,negativos}$$

Ahora podemos definir la curva PR (precisión-recall) es el resultado de dibujar una gráfica colocando la precisión en el eje de las ordenadas y la sensibilidad en el eje de las abscisas. El AP es en esencia el área bajo esta curva, no obstante, es necesario suavizarla para poder calcular correctamente el AP, es por esto que el AP se define como:

$$AP = \frac{1}{11} \sum_{r \in \{0;0,1;...;1\}} Pinterp(r)$$

Donde

$$Pinterp(r) = \max_{\tilde{r}:\tilde{r} \geq r} p(\tilde{r})$$

Y donde $p(\tilde{r})$ equivale a la precisión medida para una sensibilidad $\tilde{r}$.

El hecho de que el AP unifique la sensibilidad y la precisión en un solo número comprendido entre el 0 y el 1, lo hace que sea ideal para con un simple valor entender si nuestro modelo se esta comportando de la forma que deseamos. Además presenta una gran utilidad para comparar distintos modelos en la misma tarea.

1.2.2. Aprendizaje transferido

El aprendizaje transferido o *transfer learning* se basa en utilizar un aprendizaje previo para resolver otro problema relacionado (10). Por ejemplo, el conocimiento que ha obtenido un modelo que reconoce rascacielos, probablemente se pueda reutilizar para reconocer códigos de barras, dado que el modelo internamente ha aprendido a reconocer estructuras elongadas.

El aprendizaje transferido puede permitir un uso mucho más eficiente de las muestras disponibles para el entrenamiento (11).

Capítulo 2

Objetivos, motivaciones y estado del arte

2.1. Objetivos

El objetivo principal de este TFG es aprovechar los avances tecnológicos en el mundo de la segmentación semántica a través del *machine learning* para elaborar una aplicación capaz de ser ejecutada en teléfonos móviles Android que sea capaz de detectar los códigos de barras que se encuentren en el campo de visión de la cámara del teléfono móvil aparentemente sin demora. Esto sucede cuando se detectan a un *framerate* cercano a 30fps, el tiempo habitual de adquisición de vídeo, lo que en unidades temporales equivale a 33 ms.

Como ya se comentó, para poder entrenar dicho modelo se decidió utilizar una colección de imágenes previamente recopilada por los tutores de este TFG, carente de etiquetas. Por lo que el segundo objetivo de este TFG será etiquetar gran parte de la colección, tanta como sea necesario para poder entrenar con garantías el modelo.

El gran volumen de imágenes a etiquetar y la baja productividad que proporcionaban las herramientas etiquetadoras generalistas obligó a reformular ese segundo objetivo al desarrollo de una herramienta etiquetadora semiautomática que permita realizar el etiquetado tan solo requiriendo una interacción mínima por parte del usuario.

2.2. Motivaciones

La motivación final de este trabajo, más allá del puro afán formativo, y conjuntamente con otros aportes que serán posteriores al mismo, es desarrollar aplicaciones de AR (Realidad Aumentada) que posicionen, certeramente y sin retraso, los códigos de barras como puntos de referencia del mundo real.

2.3. Estado del arte

Recientemente Google ha liberado una nueva API de lectura de códigos de barra que realiza por el desarrollador las dos fases del proceso de lectura de códigos de barras: la localización y la decodificación.

Para ello hace uso de redes neuronales, y de hecho está contenida en el denominado *machine learning kit* (12). Su rendimiento es completamente satisfactorio, salvo porque no respeta los tiempos de procesamiento a ritmo de adquisición, por lo que su uso sigue siendo inviable en aplicaciones de AR.

La empresa *Scandit* también presenta un SDK con un desempeño admirable, aunque sufre el mismo problema que el ML Kit de Google, pues tampoco logra satisfacer el ritmo de adquisición, tal y como se puede observar en su propio contenido promocional(13). Al contrario que el SDK de Google, el utilizar este SDK lleva aparejado el pago de una licencia de uso.

Para tener un punto de comparación algo más realista que enfrentarnos al gigante Google, cabe mencionar un artículo reciente del estado del arte (14) en la misma línea. Reporta 19.2 ms en la tarea de localización usando redes neuronales. Ahora bien, sobre un PC con CPU i7 dotado además con una tarjeta gráfica Titan V, encargada de la inferencia de la red. Aún así lo describe como un gran logro ya que los métodos que se revisan en dicho artículo, datados entre 2013 y 2017, y contra los que se compara, tardan entre 130 y 25 ms.

Para terminar de enmarcar el alcance de ese trabajo, publicado en una revista Q2 JCR, se analizan exclusivamente imágenes estáticas, no se lleva a implementación en móvil, y el entrenamiento se realizó a partir de los *datasets* públicos ya anteriomente mencionados (7; 8). Las características de estos *datasets* son:

- Artelab: Contiene 365 imágenes etiquetadas de códigos de barras EAN (*European Article Number*) capturados con diferentes teléfonos móviles previos a 2013, cuando el *flagship* de Samsung de la época era el SIII dotado con una cámara 8Mpx, f2.6 y 1/3” de diagonal.
- Muenster: Contiene 1055 imágenes de códigos de barras EAN y UPC (*Universal Product Code*). La mitad están etiquetadas. El móvil con el que se capturaron, un Nokia N95, data de 2007, y su cámara exhibía 5 MPx con la captura de vídeo limitada a 480p a 30fps.

El tamaño de las imágenes de ambos *datasets* es de 640 × 480.

Capítulo 3

Metodología de trabajo

Se ha utilizado un metodo de trabajo iteritivo similar a *AGILE* (15), donde primero, en una reunión entre el alumno y los profesores tutores, se definen los objetivos a realizar y se añaden a la lista de tareas pendientes en el servicio Trello (16) (donde se tenia un tablero *kanban* compartido). En la duración de la iteración el alumno iba trabajando en tareas que movía a la columna de ”en proceso”. Los cambios se iban almacenando en distintas ramas en un repositorio de Git(17), cuando se consideraba la tarea como correctamente terminada, se unía su rama a la rama principal y se marcaba la tarea como terminada en Trello. Las iteraciones no tenían una duración fija e iban variando según disponibilidad y dificultad de cada tarea.

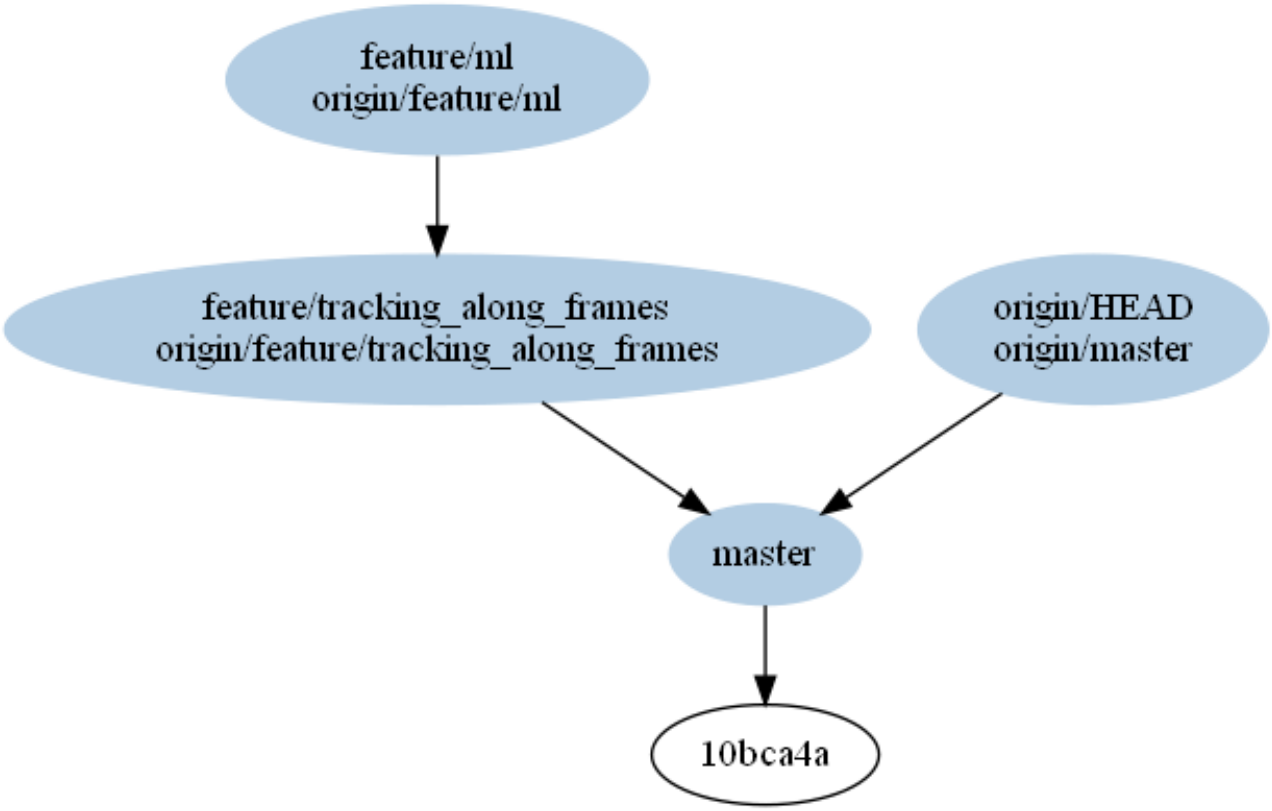


Figura 3.1: Gráfico de las ramas del repositorio git. No contiene todas las ramas pues la mayoría se iban eliminando según se unían con la rama principal

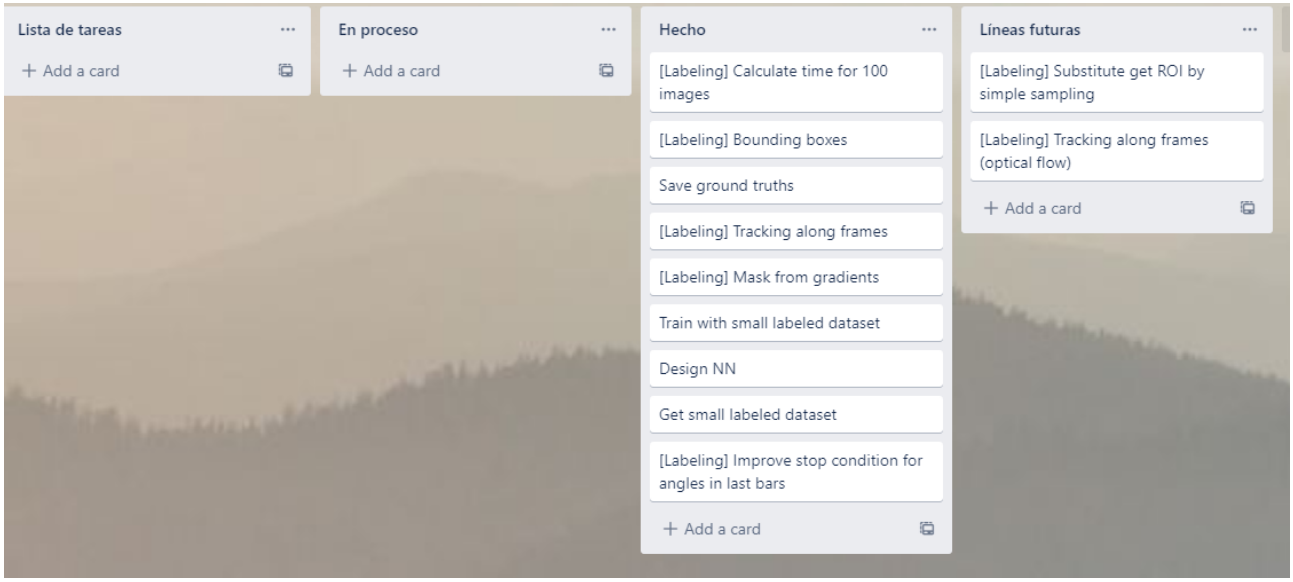
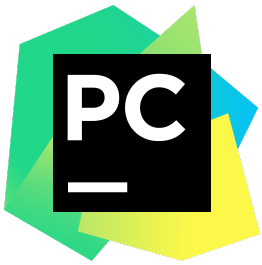


Figura 3.2: Captura del estado actual del tablero compartido en Trello.

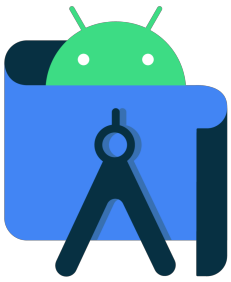
La mayoría del código generado en este TFG ha sido escrito en el lenguaje de programación Python (18), usando el entorno de desarrollo PyCharm (19). Con la excepción del desarrollo de la aplicación Android que se programo utilizando los lenguajes Java (20) y Kotlin (21), usando el entorno de desarrollo Android Studio (22). La memoria se ha redactado en LaTeX (23), usando el entorno Overleaf (24).

(a) Logotipo del lenguaje de programación Python.



(b) Logotipo del entorno de desarrollo PyCharm

(c) Logotipo del lenguaje de programación Kotlin.



(d) Logotipo del entorno de desarrollo Android Studio

(e) Logotipo del lenguaje de programación Java.

(f) Logotipo del sistema de composición de textos LaTeX.

(g) Logotipo del entorno Overleaf.

Figura 3.3: Logotipos de algunas de las herramientas utilizadas.

Capítulo 4

Herramienta etiquetadora

Como se indicaba en los objetivos, 2.1, se busca desarrollar una herramienta etiquetadora semiautomática. En el desarrollo de esta herramienta se probaron dos acercamientos, pues el primero se consideró no satisfactorio. Ambos acercamientos se detallan a continuación.

4.1. Mediante *watershed*

4.1.1. Modo de empleo

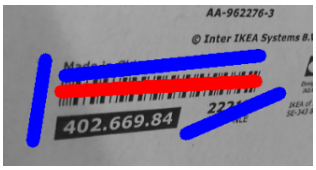
El usuario guía a la herramienta etiquetadora de *watershed*(25), indicando una serie de líneas sobre la imagen. En concreto, se esperaba que pintara líneas azules para marcar el fondo y que luego pintara una línea roja en la longitudinal de cada uno de los códigos de barras. Con estos *inputs* la herramienta era capaz de generar trapezoides que contienen a los códigos de barras.

4.1.2. Algoritmo

Primero debemos explicar como funciona la transformación de watershed por inundación. La forma más sencilla de entender su funcionamiento es con un ejemplo y un símil. Imaginemos la imagen de la figura 4.1a como un mapa topográfico donde el nivel de brillo de cada punto representa su altura. Si ahora pintamos una lineas como en la figura 4.1b, pintando de rojo nuestra zona de interés y de azul zonas que corresponden al fondo, y localizamos los mínimos locales bajo dichas lineas y comenzamos a verter en ellos líquidos del mismo color de la linea que les corresponde hasta inundar completamente el terreno. Según se va inundando el terreno construimos de forma inmediata barreras allí donde los líquidos de distinto color se toquen. Cuando finalice el proceso nos quedaría una imagen similar a la figura 4.1c, que representa la ubicación de nuestras barreras, y son aproximadamente el contorno de nuestra zona de interés.

El etiquetador realiza este proceso para hallar el contorno de los códigos de barras en

(a) Imagen de entrada.



(b) Lineas dibujadas por el usuario sobre la imagen.

(c) Salida de la transformación de *watershed*.

(d) Trapezoide de salida (en cian) sobre el contorno obtenido del *watershed*.

Figura 4.1: Entradas, paso intermedio y salida del etiquetador por *watershed*.

la imagen, no obstante esto no es suficiente, pues es necesario simplificar el contorno para que pueda ser descrito con cuatro puntos. En un inicio se optó por calcular el rectángulo rotado mínimo, pero se determinó que no aportaba la suficiente precisión, por lo que se decide calcular un trapezoide en su lugar. Este paso en concreto llevo un largo periodo en solventarse pues la línea a seguir no era clara, ya que no existe un método matemático que calcule el trapezoide mínimo que recubra un polígono cualquiera. Lo más parecido es la aproximación de un contorno utilizando el algoritmo de Ramer–Douglas–Peucker (26).

El problema de esta aproximación es que no se puede controlar cómo hace la aproximación y puede dejar fuera del trapezoide grandes cantidades del código de barras o incluir demasiado del fondo. Finalmente se solvento el problema implementando un algoritmo que aproxima el trapezoide siguiendo una heurística. El algoritmo funciona de la siguiente forma:

1. Se calcula el rectángulo rotado de área mínima que envuelve al contorno.
2. Se considera este rectángulo como la mejor solución hasta el momento.
3. Se repiten 4000 veces los pasos indicados en la sublista:
   a) Se escoge un punto aleatorio de la mejor solución actual y se acerca una distancia aleatoria al centro del contorno.
   b) Se calcula una puntuación con una función heurística que tiene en cuenta la cantidad de fondo que se ha dejado fuera del trapezoide (aumenta la puntación) y la cantidad de código de barras que se ha dejado fuera del trapezoide (disminuye la puntación).
   c) Si la puntación actual supera a la de la mejor solución, se considera la actual modificación como mejor solución.
   d) Si se lleva 50 iteraciones sin actualizar la mejor solución, se guarda la solución actual como un posible candidato y se decide de forma aleatoria con pesos cualquiera de los dos siguientes pasos:
      - **80 % de probabilidad**: Se escoge un candidato de la lista de candidatos que tenga mejor puntación que la mejor solución actual y se utiliza como mejor solución actual.
      - **20 % de probabilidad**: Se toma como mejor solución actual el rectángulo rotado de área mínima calculado en el primer paso.

e) Si se lleva 50 iteraciones sin actualizar se alejan de centro del contorno los puntos de la mejor solución actual una distancia aleatoria (una distinta por cada punto).

4. Se devuelve el candidato con mayor puntuación.

4.1.3. Motivos de abandono

Si bien el algoritmo anteriormente descrito conseguía aproximar de forma razonable los trapezoides, la precisión del contorno que generaba la transformación de *watershed* deja que desear como se puede apreciar en la figura 4.1c. Además, se planteaban dudas de la ventaja ergonómica frente a simplemente marcar los cuatro puntos de forma manual como se haría en las herramientas etiquetadoras, pues puesto en términos simple, la mejora de dibujar cuatro líneas en vez de cuatro puntos no es mucha. Por último, si bien se cree que es más factible transportar y ajustar las líneas a la siguiente imagen (asumiendo que las imágenes son una secuencia extraída de un vídeo como es nuestro caso) que, si se usaran los puntos directamente, la viabilidad de esta última estrategia es algo que se determinó como poco efectivo.

Por los motivos expuestos se decide abandonar el método de *watershed* y desarrollar un nuevo método.

4.2. Mediante Gradientes

4.2.1. Modo de empleo

Esta nueva herramienta etiquetadora es mucho más sencilla de usar, pues solo requiere realizar un clic en algún punto dentro del código de barras y la herramienta se encarga de calcular de forma automática el trapezoide que contiene al código de barras. Además, si la imagen forma parte de una secuencia extraída de un vídeo, con solo pulsar una tecla la herramienta saltará a la siguiente imagen e intentará volver marcar todos los códigos de barras que todavía estén presentes en ubicaciones cercanas a las que se encontraban en la anterior imagen.

Esta simplificada metodología de uso se acerca muchísimo más a lo que buscábamos cuando planteamos desarrollar una herramienta de etiquetado semiautomática.

4.2.2. Algoritmo

El primer paso del algoritmo es calcular los gradientes de la imagen. En una función multidimensional el gradiente es un vector con las derivadas parciales en cada dimensión. Las imágenes, en escala de grises, en este sentido se pueden considerar funciones discretas bidimensionales, $I(x, y)$, y en ellas el gradiente se calcula con filtros en diferencia, en

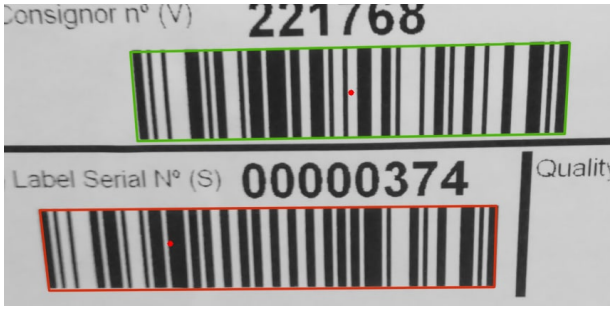


Figura 4.2: Ejemplo de uso de la herramienta etiquetadora mediante gradientes. Los dos puntos rojos representan dónde ha pinchado el usuario. Los trapezoides son calculados automáticamente.

vertical y horizontal.

$$\nabla I\left(x_0, y_0\right) = \begin{bmatrix} \frac{\partial I}{\partial x}\left(x_0, y_0\right) \\ \frac{\partial I}{\partial y}\left(x_0, y_0\right) \end{bmatrix}$$

Este vector, que se calcula por píxel, nos indica la intensidad y dirección del cambio de la intensidad de la imagen en torno a ese elemento. Al ser un vector bidimensional se puede interpretar atendiendo a su módulo y ángulo, que llamaremos *moduli* y *theta*.

Los gradientes son extremadamente útiles para la detección de códigos de barras pues se cumple que en las barras de un código el gradiente tendrá una alta intensidad y también se cumple que el ángulo del gradiente sera bastante similar al ángulo de las barras del código. Esto se puede observar en la figura 4.3.

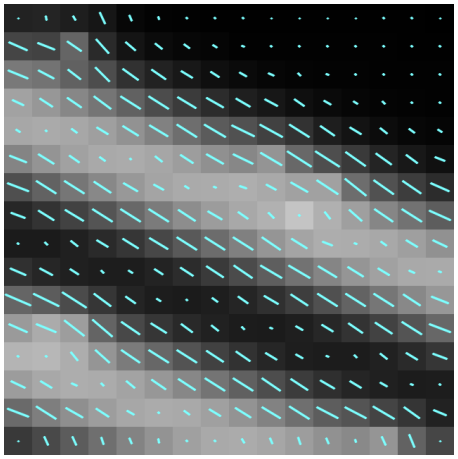


Figura 4.3: Gradientes por píxel en una zona con barras. La longitud de las líneas cian se corresponde a la intensidad del gradiente, el *moduli*. La inclinación corresponde a la dirección de mínimo cambio de gradiente, las *thetas*.

El algoritmo comienza por calcular los gradientes de la imagen, los cuales devuelve en dos salidas las *thetas* y el *moduli*. Las *thetas* nos indican el ángulo de los gradientes sobre cada píxel, mientras que el *moduli* nos indica la certeza de que cada píxel individual pertenezca un grupo que conforma una línea recta.

Usaremos los gradientes para transitar a lo largo del código de barras partiendo de un punto inicial indicado por el usuario.

- Se calcula el ángulo del código de barras mediante una media circular de los valores de *thetas* de los píxeles contenidos en un cuadrado con centro el punto marcado por el usuario.
- Se recalcula el ángulo, pero esta vez considerando un rectángulo rotado con el ángulo anteriormente calculado. Esto se realiza para mejorar el ángulo pues la nueva ventana contendrá menos parte del exterior del código de barras.
- Se avanza perpendicular al ángulo del código de barras para encontrar su extremo inferior y superior.
- Se calcula el punto medio entre ambos y se posiciona el nuevo punto de referencia en este punto medio, así nos aseguramos de estar centrado verticalmente en el código de barras y no en uno de sus extremos inferiores o superiores. Una vez más se reajusta el ángulo de forma similar a lo ya descrito.
- A partir de ese punto de referencia se avanza por el código de barras siguiendo la dirección y sentido indicado por el ángulo calculado y cuando se encuentre el final se apuntará y se repetirá el proceso, pero esta vez caminado siguiendo la misma dirección, pero con sentido contrario al del ángulo, para así finalmente haber recorrido la totalidad del código de barras.

De forma más concreta cuando se recorre el código de barra se va moviendo el punto de referencia y junto a él una ventana rectangular rotada con el mismo ángulo que el código de barras, dicha ventana siempre mantiene el punto de referencia como su punto central. En cada paso que da el punto se calcula la media del *moduli* y la desviación típica circular de las *thetas* que se encuentran debajo de la ventana, estos valores se comparan con unos umbrales obtenidos de forma experimental para discernir si el punto de referencia se encuentra o no en el código de barras. Cada cierto número de pasos se recentra en la vertical el punto de referencia para compensar con pequeñas diferencias entre el ángulo real del código de barras y el calculado, además en este paso se vuelve a recalcular el ángulo pues éste no tiene por qué ser constante durante todo el código de barras ya que ciertas perspectivas pueden hacer que el ángulo varíe en la longitudinal del código de barras.

Cuando finalmente se determina que el punto ha salido del código de barras, el punto desanda el camino esta vez utilizando una ventana de misma altura, pero menor ancho para poder detectar con certeza la última barra negra del código de barras (que sería la primera que encontrarse pues está andando desde fuera hacia dentro) y por tanto el final real del mismo. Una vez localizado se recorre verticalmente la última barra negra del código, pero con la peculiaridad de que, si bien se vuelven a utilizar las dimensiones de la ventana original, ésta ya no está centrada en el punto, sino que éste forma parte de su borde.

De esta forma nos aseguramos de que la totalidad de la ventana está sobre el código de barras, en lugar de estar con la mitad por fuera como ocurriría con la ventana centrada en el punto de referencia cuando éste se encuentra en el límite del código.

Existe otra peculiaridad y es que en este recorrido vertical no usamos la desviación típica de las *thetas* bajo la ventana sino que calculamos un histograma de las *thetas*. Este

histograma se suaviza sustituyendo el valor de cada contenedor con la media de sí mismo junto a su vecino anterior y posterior de forma circular (es decir el vecino anterior de la primera posición es la última posición y por ende el vecino posterior de la última posición es la primera posición), esto se hace para tener en cuenta el comportamiento circular de los ángulos. Este histograma se compara con unos umbrales para determinar si seguimos en la última barra o hemos llegado a la esquina del código. Al finalizar se apuntan las coordenadas de las esquinas de dicha barra.

Una vez terminado el paseo por el código de barras ya contamos con las coordenadas de las cuatro esquinas del código de barra y estas representaran nuestro trapezoide.

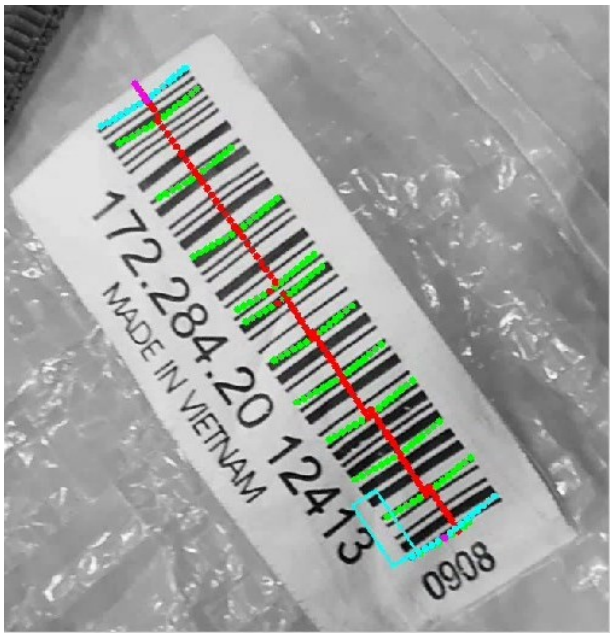


Figura 4.4: Se muestra el camino recorrido en el código de barras. Los puntos rojos representan las posiciones del punto de referencia, los puntos verdes representan el camino en la vertical necesario para recentrar verticalmente el punto de referencia, los puntos rosas representan el caminado hacia atrás que realizado el punto de referencia, los puntos cian representan el caminando para determinar los bordes de la última barra y finalmente el recuadro cian representa la ventana deslizante que en esta imagen se encuentra en la posición donde detectara que el punto de referencia ha llegado al límite de la barra vertical.

Se adjunta extracto de este código como Anexo A.1.

4.2.3. Limitaciones

Este método de herramienta etiquetadora funciona de forma bastante satisfactoria, pero tiene unas ciertas limitaciones.

Primero, requiere que el código de barras no este desenfocado en ninguna de sus partes, o de lo contrario, en la mayoría de los casos, no lo reconocerá de forma correcta. Además, es posible que el etiquetador se confunda y tome como parte del código de

barras elementos cercanos al mismo que no forman parte de él, ejemplo de esto son: barras divisorias que ponen los fabricantes en las etiquetas, caracteres especialmente rectos (como “1”, “i” o “l”) cercanos al código, o un segundo código pegado al que está siendo analizado.

La primera limitación es inherente al uso de los gradientes, pues estos no se pueden calcular de forma correcta en una imagen desenfocada, mientras que la segunda, aunque se ha tratado de minimizar, resulta imposible de eliminar en todos los casos (al menos sin usar un algoritmo órdenes de magnitud más complejo) por el simple parecido que guardan estos caracteres y líneas con los componentes de un código de barras.

Por ultimo, cabe resaltar que la herramienta etiquetadora solo es funcional en imágenes en blanco y negro.

(a) Primera imagen de la secuencia. El punto de referencia se ha señalado manualmente.

(b) Segunda imagen de la secuencia. Punto de referencia se ha extrapolado de la imagen anterior.

(c) Tercer imagen de la secuencia. Punto de referencia se ha extrapolado de la imagen anterior. El código presenta desenfoque y se ha realizado una detección parcial, por lo que esta imagen se descartaría.

Figura 4.5: Etiquetado de una secuencia de imágenes.

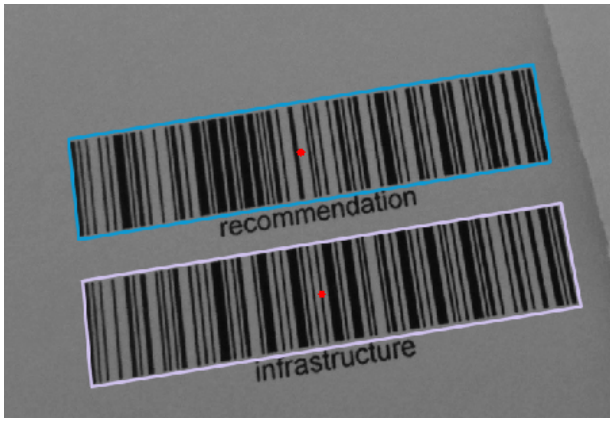


Figura 4.6: Etiquetado de una imagen con múltiples códigos de barras

Capítulo 5

Etiquetado del *dataset*

La etiquetación del *dataset* es una tarea altamente mecánica y fue facilitada en gran medida por la herramienta etiquetadora desarrollada. Del *dataset* se procesaron más de 25.000 imágenes, de las cuales resultaron con etiquetas válidas **4.545** imágenes (aproximadamente el 18 % de las imágenes). Este número puede resultar alarmantemente bajo, pero realmente no lo es tanto si se tiene en cuenta que realmente no todas las imágenes contienen códigos de barras y de las que, si lo contenían, muchos tuvieron que ser descartados por estar parcialmente o completamente desenfocada, una minoría tuvo que ser descartada porque el etiquetador incluía elementos que no pertenecían al código de barras. También hay que resaltar que cualquier problema en la detección se amplificaba en las imágenes con múltiples códigos de barras pues solo era necesario que uno no se pudiera etiquetar correctamente para desechar la imagen completa.

El *dataset* se componía por los frames extraídos de múltiples vídeos, cuyas grabaciones se realizaron en diversas localizaciones. Las imágenes procesadas provienen de 6 localizaciones y 153 vídeos distintos. Es importante destacar esta información pues introduce un sesgo en nuestro *dataset* ya que las imágenes pertenecientes a un mismo vídeo contendrán un alto parecido entre sí y los vídeos grabados en la misma localización mostrarán el mismo formato de código de barras. En las imágenes estaban presentes códigos EAN/UPC, CODE-128 e ITF14.

Si comparamos con los *datasets* disponibles públicamente, estos contienen la décima y la octava parte de imágenes etiquetadas, y se limitaban a códigos EAN/UPC.

5.1. Transformación al formato de *labelme*

La herramienta etiquetadora guarda la información de las etiqueta en archivos con extensión `.pickle` que guarda con el mismo nombre y en la misma ruta de directorio que la imagen a la que corresponde. La forma en la que se guarda la información en los archivos `.pickle` está altamente adaptada al etiquetador, por lo que resultaba conveniente transformarlo en un formato más estándar. Se decidió utilizar el formato de *labelme* (27) por su sencillez y su parecido a la información que guardaba el etiquetador, ya que este formato guarda la información relevante a la imagen en un fichero `JSON` junto

a la imagen que acompaña, la estructura de los datos en el fichero `JSON` es bastante sencilla por lo que un simple *script* de Python bastó para transformar la salida del etiquetador al formato de labelme.

5.2. Transformación al formato de COCO

Si bien la herramienta de *labelme* es popular, su formato no es utilizado por la inmensa mayoría de herramientas de *machine learning*. Pero esto no resulta problemático pues el formato de COCO(28) si es ampliamente soportado y con la herramienta labelme2coco (29) la transformación apenas requiere de un comando y unos escasos minutos.

Capítulo 6

Entrenamiento de los modelos de *Machine Learning*

Cuando dispusimos de un *dataset* adecuado para el entrenamiento de nuestros modelos, pasamos a elegir qué modelos son los más adecuados para nuestros objetivos. En este TFG se han entrenado dos modelos para dos propósitos distintos, con el primero de ellos se buscaba hacer segmentación de instancia, mientras que con el segundo se buscaba realizar segmentación semántica.

6.1. Entorno de pruebas

Todas las pruebas descritas se han ejecutado en un ordenador con las siguientes especificaciones técnicas:

- **CPU**: AMD Ryzen 7 3700X
- **GPU**: NVIDIA GeForce GTX 1080 8GB
- **RAM**: 16GB

Respecto al entorno de software, las pruebas se ejecutaron en contenedores OCI (30) sobre la plataforma Docker Desktop (31) en Windows 11 (32) usando la WSL2 (33) como *backend*. De forma más concreta las pruebas de la sección 6.2 se basaron en la imagen *nvidia/cuda:11.3.1-cudnn8-devel-ubuntu20.04* (34) y las de la sección 6.3 en la imagen *tensorflow/tensorflow:latest-gpu-jupyter* (35).

Las pruebas en teléfono móvil se realizaron sobre un Xiaomi Mi 10, el cual cuenta con un SoC Snapdragon 865 (topo de gama de la generación de 2020) ejecutando la versión del SO más actual disponible en la fecha de elaboración de este TFG, Android 12.1.

6.2. Modelo de segmentación de instancia

Para el modelo encargado de realizar segmentación de instancia se utilizo la librería Detectron2 (36) de Facebook. Se decidió utilizar esta librería porque ofrece algoritmos de estado del arte para la segmentación de imágenes, cuanta con mantenimiento muy activo y goza de una amplia documentación. En cuanto al modelo, se uso el modelo Mask R-CNN (37) con ResNet-50-FPN (38) como *backbone*.

El *dataset* se dividió en dos partes del 80 % y el 20 % respectivamente. La primera con 3636 imágenes se usó para entrenar propiamente al modelo, la segunda compuesta por las 909 imágenes restantes se utilizo para evaluar al modelo. Adicionalmente, se reescalaron las imágenes de su tamaño origina 1024x1024 píxeles a 256x256 píxeles para acelerar el proceso de entrenamiento e inferencia, dada las limitaciones de la máquina en la que entrenábamos.

Para acelerar el proceso de entrenado se aplicó *transfer learning* de los pesos para el desafío de COCO de 2017 para este modelo obtenidos desde el ModelZoo de Detectron (36).

Una vez finalizado el entrenamiento, el modelo reporto un AP de 0.7554 y un tiempo de inferencia medio 79.16 ms para el *dataset* de evaluación. El resultado del AP es satisfactorio pues indica que la mayoría de los casos el modelo se comporta de forma adecuada, pero por otro lado, el tiempo de inferencia resulta algo elevado pues más que dobla el tiempo habitual de adquisición de vídeo, pecado aun más grave si se tiene en cuenta que este tiempo de inferencia es sobre el equipo presentado en el sección 6.1, el cual es significativamente más potente que cualquier teléfono Android ordinario.

El comprometido tiempo de inferencia de este modelo nos hizo decidirnos por probar un modelo de segmentación semántica en su lugar, pues si bien el poder reconocer como únicos los distintos códigos en la imagen puede ser deseable, no es imprescindible para este TFG y el propio artículo que presenta Mask R-CNN reconoce que añade sobrecarga al proceso de segmentación semántica (37).

6.3. Modelo de segmentación semántica

Para el modelo de segmentación semántica se valoraron a múltiples candidatos como ShuffleSeg (39), ENet (40), PSPNet (41), RefineNet (42) y DeepLabV3+ (43) entre otros. De todos ellos decidimos decantarnos por DeepLabV3+ por ser uno de los más recientemente publicados y mejor valorados, por contar con implementaciones que no dependen de versiones de software *legacy*, por ofrecer un buen balance entre tiempo de ejecución y tasa de aciertos y por ser la que la propia Google recomienda a la hora de implementar segmentación semántica en Android (44; 45).

Se utilizó, como base para el modelo, la implementación propuesta por Keras (46) en su documentación oficial (47). Esta implementación tiene la ventaja de funcionar en las ultimas versiones de Tensorflow (48), lo cual facilitará el exportar el modelo a Android y nos permite aprovecharnos de las ultimas mejoras de rendimiento implementadas

en la librería. Adicionalmente, se modificó dicha implementación para poder utilizar opcionalmente MobileNetV2 (49) como *backbone* de la red en lugar de Resnet-50-FPN basándonos en el trabajo publicado por Emil Zakirov (50).

El *dataset* se dividió de la forma presentada en la sección previa, 6.2. Esta vez se realizó el *transfer learning* con pesos entrenados con el dataset ImageNet (51) para el *backbone* de ResNET y otros entrenados en el *dataset* de PascalVOC (9) cuando se usa MobileNetV2.

Una vez entrenadas todas las variantes del modelo, se obtuvieron los siguientes resultados sobre el *dataset* de evaluación:

Modelo	AP	T. inferencia (PC)	T. inferencia (móvil)
DeeplabV3+ (ResNet50)	0.8369	20.11 ms	127.0 ms
DeeplabV3+ (MobileNetV2)	0.7856	19.52 ms	89.83 ms

Cuadro 6.1: Resultados de las ejecuciones de los modelos de segmentación semántica contra el *dataset* de evaluación

Sorprendentemente ambas variantes han obtenido un mayor AP que Mask R-CNN, y tal como esperábamos DeepLabV3+ ha logrado reducir drásticamente los tiempos de inferencia, el cual entra cómodamente en el tiempo habitual de adquisición de vídeo, de ejecutarse en el equipo de sobremesa.

En la ejecución en el teléfono móvil (la cual ha ejecutado una versión optimizada del modelo como se describe en la sección 7.1) nos encontramos que tardamos algo menos del triple de nuestro tiempo objetivo de 33ms para la variante de MobileNetv2.

Si bien los tiempos obtenidos no llegan a cumplir nuestro objetivo de 33ms, consideramos este modelo como un éxito, pues hemos logrado unos tiempos similares a los expuesto en el articulo mencionado en el estudio del estado del arte (14) y lo hemos logrado utilizando una GPU de menor gama y considerable menor potencia. Además, teniendo un enfoque realista, lograr dicho objetivo de tiempo cuando ni siquiera Google en su último intento lo ha logrado, suena poco plausible.

6.4. Discusión de resultados

En general todos los modelos funcionan bien en condiciones lumínicas razonables, pudiendo detectar sin problema múltiples códigos de barras en la misma imagen sin importancia de donde estén colocados (Fig. 6.1b). Incluso se ha observado un rendimiento destacable en situaciones de baja luminosidad como se observa en la figura 6.1a,

(a) Ejemplo de ejecución del modelo basado en DeepLabV3+ (MobileNetV2) en un entorno de baja luminosidad.

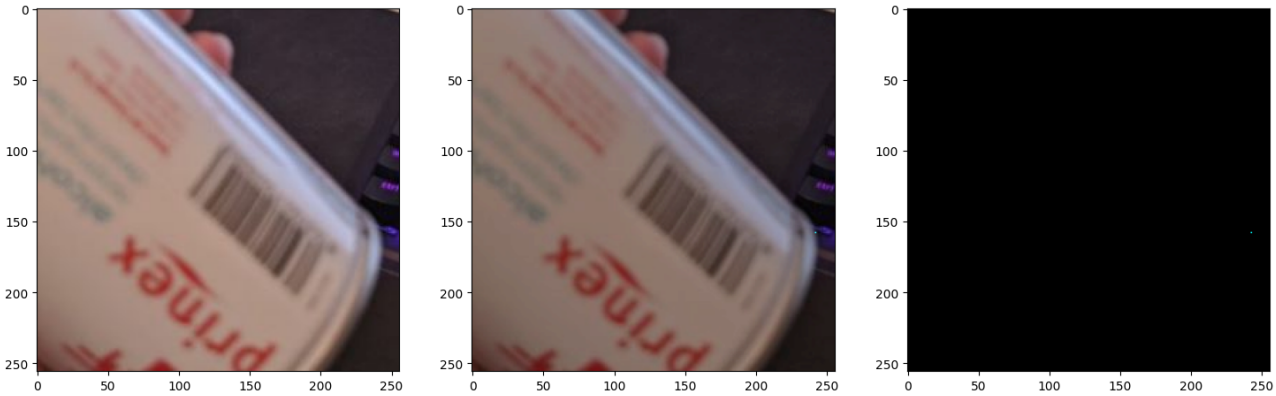
(b) Ejecución del modelo R-CNN sobre una imagen con múltiples código de barras en condiciones normales.

Figura 6.1: Pruebas del correcto funcionamiento de los modelos.

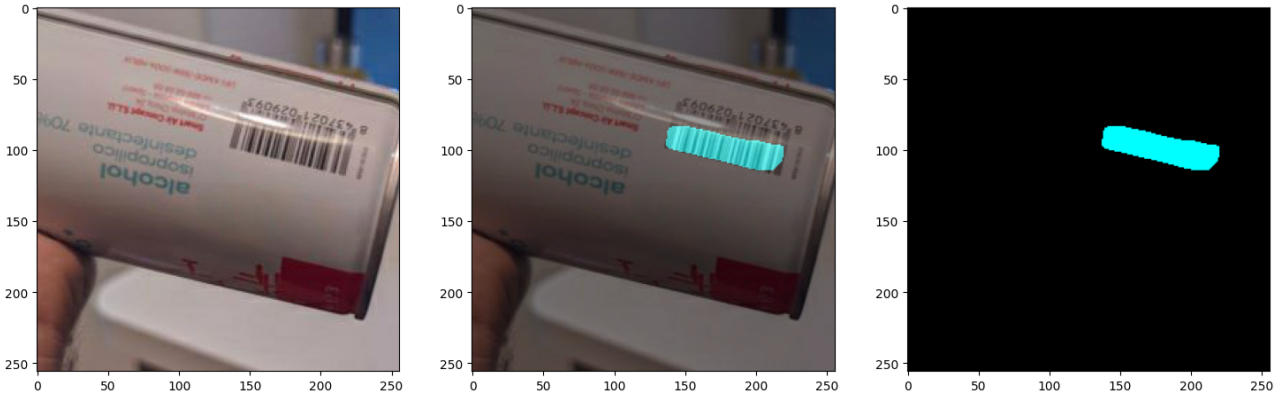
Lamentablemente existen situaciones donde los modelos no funcionan tan bien. Por ejemplo cuando el código de barras sufre desenfoque, ya sea de lente o de movimiento, son constadas las ocasiones en las que los modelos consiguen detectar el código de barras, y de éstas, las situaciones donde consigue señalar con certeza el área completa que ocupa éste son inexistentes. Hay un ejemplo de ello en la figura 6.2a. Otra situación poco favorable, aunque menos común, nos la encontramos cuando un reflejo aparece sobre el código de barras, éste solo se detecta parcialmente, marcando únicamente las partes que no están afectadas por el reflejo, figura 6.2b.

Dado que esta limitación se ha observado en todos los modelos de forma similar, podemos deducir que el problema se encuentra en las imágenes con las que se han entrenado. La baja certeza en condiciones en la que la imagen está borrosa se puede rastrear hasta la herramienta etiquetadora, la cual sufre de la misma limitación, lo cual restringe las imágenes etiquetadas a aquellas imágenes del *dataset* que no estén borrosas. Por otro lado, para los reflejos el problema reside que las imágenes en el *dataset* donde el código presenta un reflejo son limitadas.

Que los problemas provengan del conjunto de entrenamiento es difícilmente sorprendente dado los expuesto en el capitulo 5 y en el aparatado 4.2.3. Pero esto también es esperanzador, pues significa que con tan solo expandir nuestro *dataset* podemos obtener incluso mejores resultados.



(a) Ejecución del modelo DeepLabV3+ (ResNet) sobre una imagen desenfocada.



(b) Ejecución del modelo DeepLabV3+ (ResNet) sobre una imagen que presenta un reflejo sobre el código de barras

Figura 6.2: Demostraciones del funcionamiento del modelo en situaciones poco favorables. A la izquierda la imagen de entrada, en el centro la imagen con la mascara superpuesto, a la derecha la mascara generada.

Capítulo 7

Desarrollo de la aplicación Android

En este capítulo nos centraremos en construir la aplicación Android propuesta en los objetivos usando el modelo basado en DeeplabV3+ con el *backbone* de MobileNetV2 que entrenamos en la sección 6.3.

7.1. Transformar el modelo a TensorFlow Lite

TensorFlow Lite (52) es la librería que nos permitirá ejecutar los modelos en Android, pero para poder utilizar nuestros modelos con esta librería primero debemos transformarlos al formato de ésta. Por suerte, dado que nuestros modelos están en el formato de TensorFlow, este proceso es extremadamente sencillo y está perfectamente detallado en la documentación oficial (53). Al finalizar tendremos un fichero `.tflite` que contendrá nuestro modelo.

Podemos aprovechar este proceso de transformación para además optimizar el modelo para su ejecución en la CPU de los teléfonos móviles, este proceso también es muy sencillo y se encuentra descrito en la documentación oficial (54).

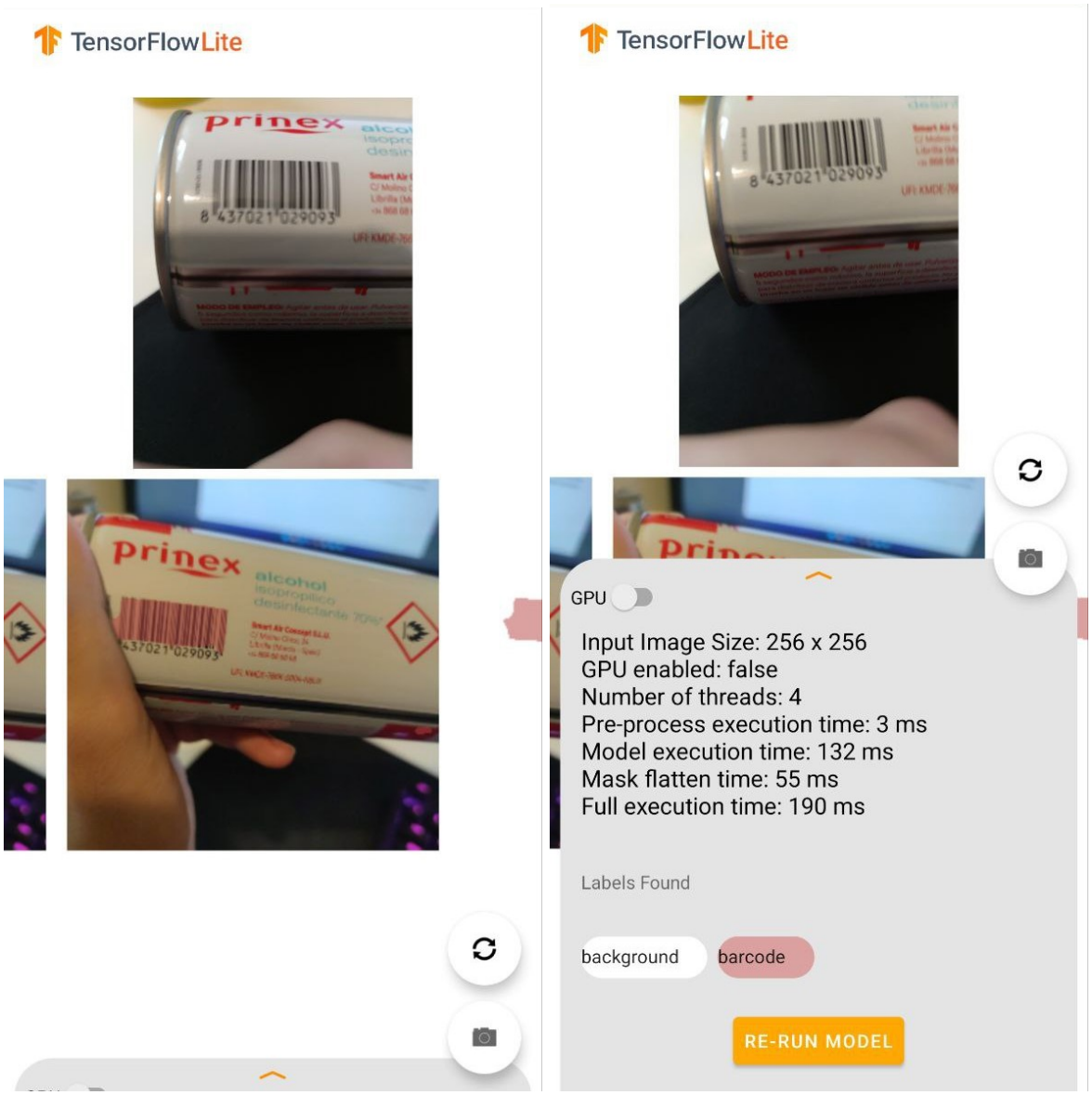
7.2. Modificar una aplicación de ejemplo

Antes de embarcarnos a desarrollar una aplicación que funcione en tiempo real (o tan cerca como nuestro modelo lo permita), primero modificaremos la aplicación de ejemplo que proporciona Google (55).

Una vez hemos clonado la aplicación solo necesitamos abrirla en Android Studio y modificar el archivo `ImageSegmentationModelExecutor.kt` que se encuentra en la solución *libinterpreter*, ahí modificaremos los valores del número de clases, la lista de clases, el nombre del fichero del modelo y la resolución de la imagen de entrada. A continuación colocamos nuestro fichero `.tflite` a la carpeta `assets` dentro de `app`.

A continuación designamos *interpreterRelease* como variante activa en el panel lateral

de *Build variants* de Android Studio y ya podemos lanzar la aplicación.



(a) Aplicación de ejemplo, resultado de ejecución del modelo

(b) Aplicación de ejemplo, panel de estadísticas

Figura 7.1: Capturas de la aplicación Android de ejemplo

La aplicación se mostrara como en la figura 7.1a. La aplicación nos permite capturar una instantánea pulsando el botón de la cámara. Dichas instantáneas aparecerán en la parte inferior de la aplicación con la mascara resultante de aplicar el modelo. Adicionalmente podemos desplegar el menú de la parte de abajo para ver detalles sobre la ejecución del modelo, tal y como se puede ver en la figura 7.1b.

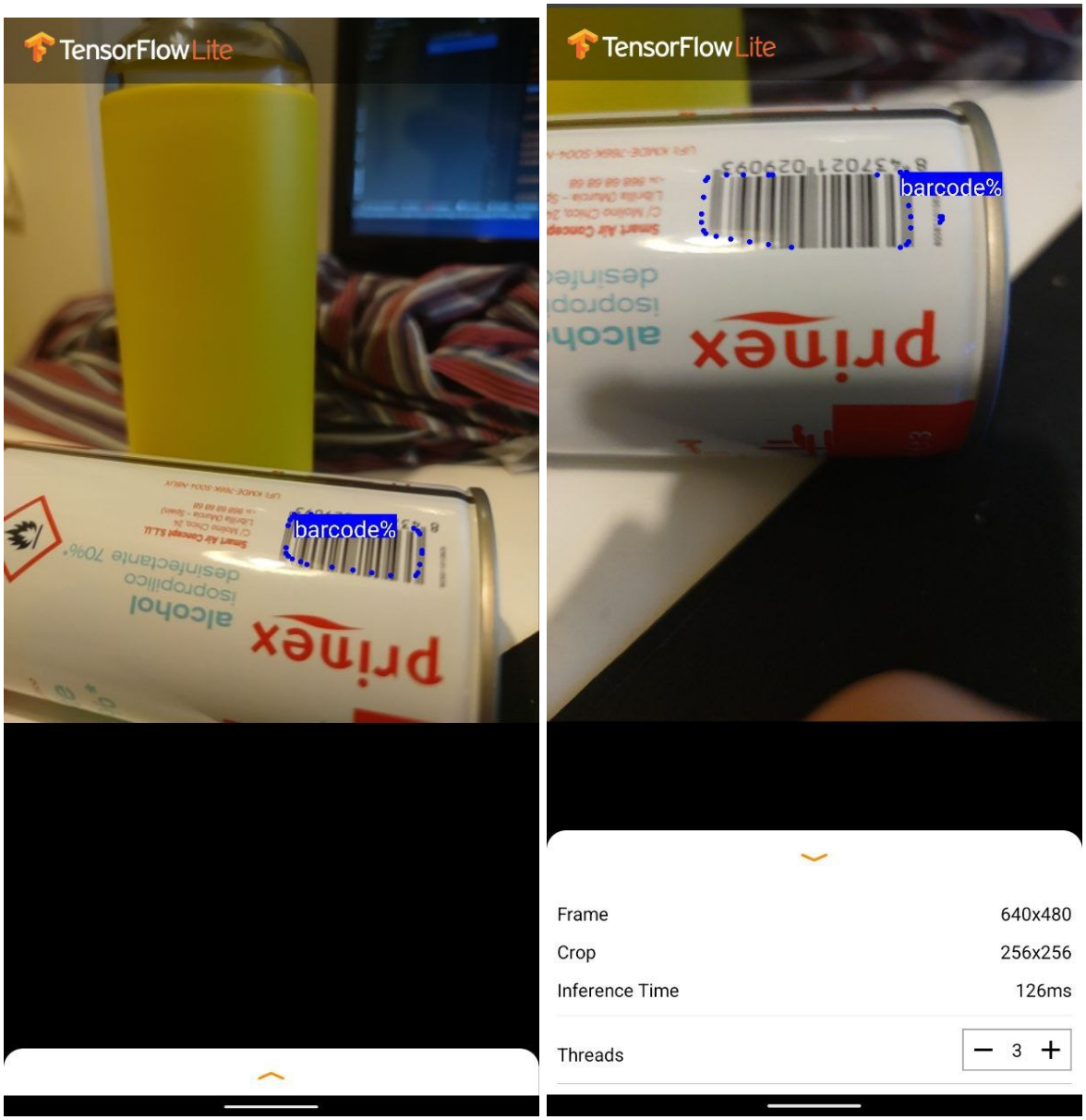
7.3. Aplicación para detección en tiempo real

Finalmente procederemos a crear una aplicación Android que nos permita detectar código de barras tan rápido como permita nuestro modelo. Nos basaremos en una

aplicación creada por *toniz* (56).

El proceso a seguir es bastante similar a la de la aplicación de demo. Primero visitamos `TFLiteSegmentionAPIModel.java` y modificamos el número de clases, luego visitamos `SegmentionActivity.java` y modificamos el nombre del archivo que contiene nuestro modelo y el tamaño de la imagen de entrada. Por ultimo, vamos a la carpetas `assets` y colocamos nuestro fichero `.tflite`, además modificaremos el archivo `deeplabv3.txt` y eliminaremos todas las clases menos *background* y añadiremos la clase *barcode* justo debajo.

Ya podemos lanzar la aplicación, enfocarla a un código de barras y nos encontraremos con algo similar a la figura 7.2a. De forma similar a la aplicación de ejemplo, ésta también cuenta con un desplegable que presenta estadísticas del modelo.



(a) Pantalla principal de la aplicación (b) Panel de estadísticas

Figura 7.2: Capturas de la aplicación de detección de códigos de barras en tiempo real

Capítulo 8

Conclusiones y líneas futuras

8.1. Conclusiones

Hemos evaluado la posibilidad, con las tecnologías actualmente disponibles, de detectar códigos de barras en tiempo real en un dispositivo móvil, sin saltarnos fotogramas. Hemos llegado al mismo punto de rendimiento en tiempos de inferencia que reportan artículos del estado científico del arte (14), e industrial (12).

De momento dicha tarea es inalcanzable en Android. No obstante, creemos firmemente que será viable en un futuro cercano, pues lo que sí demuestra este TFG es que ya es posible lograrlo en un ordenador de sobremesa relativamente modesto. Y si el pasado sirve para vaticinar el futuro, los avances en *machine learning* y en la potencia de computación de los dispositivos móviles, volverán a permitir realizar sobre éstos, tareas que hoy les están vetadas y exigen del uso de potentes ordenadores de escritorio.

Además, dado que este resultado era más o menos predecible, el TFG se orientó a elaborar una herramienta etiquetadora que ya ha permitido etiquetar un *dataset* de códigos de barra más amplio y sobre imágenes de más calidad que los disponibles públicamente entre la comunidad científica. A más, esta herramienta será reutilizable para expandirlo con facilidad en un futuro, adaptándolo a nuevos formatos de códigos de barras, o a imágenes tomadas con dispositivos mejorados.

Para cerrar estas conclusiones, podemos afirmar que todos los objetivos planteados en el anteproyecto del TFG se han alcanzado:

- Diseñar una herramienta de etiquetado no supervisada
- Entrenamiento de la red neuronal
- Validación y comparativa
- Despliegue para móviles

8.2. Líneas futuras

Posibles líneas futuras serían:

- Expandir el *dataset* con más imágenes y sobretodo con imágenes que representan una mayor variedad de entornos.
- Mejorar la herramienta etiquetadora, modificándola para que funcione con mayor certeza en las imágenes que presentan desenfoque.
- Ofrecer al usuario de la herramienta la opción de solventar, con interacción mínima, los casos donde una pequeña parte del código de barras no es detectada o en las situaciones donde se considera parte del código de barras a elementos cercanos pero ajenos a éste.

Capítulo 9

Conclusions and future lines

9.1. Conclusions

We have evaluated the possibility of detecting barcodes in real-time on a mobile phone without skipping frames using the currently available technology. In the end, we reached the same level of latency as the ones described in papers describing state of the art in both the scientific and the industrial world.

At the moment, this task seems unreachable on the Android platform. However, we firmly believe that it will be possible in the near future, as the present TFG proves that it is feasible in a home computer. Moreover, if the past can be used to predict the future, the improvements in machine learning and the increase in computational power of mobile phones will, once again, allow us to perform tasks on mobile devices that today are reserved for bigger computers.

Furthermore, given that this outcome was predictable to some degree, this TFG was reoriented to develop a tagging tool that has allowed to tag a bigger and better quality barcode dataset than the ones publicly available in the scientific community. This tool can be reutilized in the future to expand the dataset with new barcode formats and with images taken with devices with newer and better cameras.

To conclude, we can confirm that all objectives proposed in the draft of this TF were fulfilled:

- Develop a semiautomatic tagging tool.
- Train a neuronal network.
- Validation and comparative.
- Deploy in mobile phones.

9.2. Future lines

Possible future lines are:

- Expand the dataset with more images and, more importantly, images representing a more diverse set of environments.
- Improve the tagging tool, making it handle situations where the images show blur with better accuracy.
- Offer a way so users can solve with minimal interaction the situations where a small portion of the barcode is left unselected or the ones where the tool confuses nearby elements as part of the barcode.

Capítulo 10

Presupuesto

10.1. Licencias de software

Nombre del software	Coste	Anotaciones
Android	0.00 €	Gratuito. Código libre
Android Studio	0.00 €	Gratuito. Código libre
Detectron2	0.00 €	Gratuito. Código libre
Docker for Desktop	0.00 €	Licencia gratuita aplicable
Git	0.00 €	Gratuito. Código libre
Java (OpenJDK)	0.00 €	Gratuito. Código libre
Kotlin	0.00 €	Gratuito. Código libre
LaTeX (pdfLaTeX)	0.00 €	Gratuito. Código libre
Overleaf	0.00 €	Gratuito. Código libre
PyCharm	0.00 €	Gratuito. Licencia de estudiante o comunitaria
Python	0.00 €	Gratuito. Código libre
TensorFlow	0.00 €	Gratuito. Código libre
TensorFlow Lite	0.00 €	Gratuito. Código libre
Trello	0.00 €	Plan Gratuito
Widnows 11	0.00 €	Coste incluido en el precio del equipo

Cuadro 10.1: Coste de las licencias de software utilizadas.

10.2. Materiales

Material	Valor	Factor de amortización	Coste
Equipo sobremesa	1032.47 €	5 %	51.62 €
Xiaomi Mi 10	517.35 €	5 %	25.87 €
Total:			77.49 €

Cuadro 10.2: Coste de los materiales utilizados

10.3. Costes de personal

Asumiendo un coste por hora de 30€.

Tarea	Horas	Coste
Desarrollo del etiquetador	92	2,760.00 €
Investigación y comparación de modelos de *machine learning*	73	2,190.00 €
Entrenamiento y comprobación de los modelos	27	810.00 €
Elaboración de la memoria	16	480.00 €
Total	208	6,240.00 €

Cuadro 10.3: Costes de personal.

10.4. Presupuesto final del proyecto

Motivo	Coste
Licencias de software	0.00 €
Materiales	75.87
Mano de obra	6,240.00 €
Coste	6,315.87 €
Beneficio (6 %)	378.95 €
Presupuesto total del proyecto	6,694.82 €

Cuadro 10.4: Presupuesto final del proyecto.

Apéndice A

Códigos

A.1. Fragmento de la herramienta etiquetadora mediante gradientes

```
import math
import astropy
from astropy import stats
import cv2.cv2 as cv2
import scipy.stats
import numpy as np
import mask_from_gradients as mgradients
import utils

def square_corners_from_point(point: tuple[int, int], block_size: int) ->
↪  tuple[tuple[int, int], tuple[int, int]]:
    half_block_size = block_size // 2
    a = (point[0] - half_block_size, point[1] - half_block_size)
    b = (point[0] + half_block_size, point[1] + half_block_size)
    return a, b


def rec_corners_from_point(point: tuple[int, int], angle: float, half_long=40,
↪  half_short=15) -> tuple[
    tuple[int, int], tuple[int, int], float]:
    return point, (half_long * 2, half_short * 2), math.degrees(angle)


def rec_corners_from_edge_point(point: tuple[int, int], angle: float,
↪  half_long=40, half_short=5) -> tuple[
    tuple[int, int], tuple[int, int], float]:
    angle %= 2 * math.pi

    if math.isnan(angle):
        angle = 0

    sin = math.sin(angle)
    cos = math.cos(angle)

    if math.isnan(point[0]) or math.isnan(point[1]):
```

```
        point = 0, 0
    center = (
        int(point[0] + half_long * cos),
        int(point[1] + half_long * sin),
    )

    return center, (half_long * 2, half_short * 2), math.degrees(angle)


def get_roi_coordinates(angle, point, corners_method=rec_corners_from_point, half_long=40, half_short=15):
    rect = corners_method(point, np.pi - angle, half_long, half_short)
    box = cv2.boxPoints(rect)
    box = np.int0(box)
    return box, rect


# Base https://jdhao.github.io/2019/02/23/crop_rotated_rectangle_opencv/
def get_roi(src: np.ndarray, point: tuple[int, int], angle: float, corners_method=rec_corners_from_point, half_long=40,
            half_short=15) -> tuple[
    tuple[int, int], tuple[int, int]]:
    box, rect = get_roi_coordinates(angle, point, corners_method, half_long, half_short)
    add_roi_to_frame(box, (0, 100, 0))
    width = int(rect[1][0])
    height = int(rect[1][1])
    src_pts = box.astype("float32")
    dst_pts = np.array([[0, height - 1],
                        [0, 0],
                        [width - 1, 0],
                        [width - 1, height - 1]], dtype="float32")
    M = cv2.getPerspectiveTransform(src_pts, dst_pts)

    return cv2.warpPerspective(src, M, (width, height), flags=cv2.INTER_NEAREST)


def averaged_thetas_histogram(thetas_roi, moduli_roi) -> np.ndarray:
    hist, _ = np.histogram(thetas_roi, 18, weights=moduli_roi)
    avg_hist = np.empty(hist.shape)
    for i in range(hist.shape[0]):
        avg_hist[i] = np.mean([hist[i - 1], hist[i], hist[i % hist.shape[0]]])
    return avg_hist


def test_with_averaged_thetas_histogram(thetas_roi, moduli_roi, treshold) -> tuple[bool, np.ndarray]:
    avg_hist = averaged_thetas_histogram(thetas_roi, moduli_roi)
    if np.all(avg_hist < treshold):
        return False, avg_hist
    idx = np.argsort(avg_hist)

    # Test whether the second-highest value is neighbor of the first, otherwise we are detecting that is not part of
    # the barcode
    prev_neighbor = avg_hist.shape[0] - 1 if idx[-1] == 0 else idx[-1] - 1
    next_neighbor = 0 if idx[-1] == avg_hist.shape[0] - 1 else idx[-1] + 1

```

```
86      return idx[-2] == prev_neighbor or idx[-2] == next_neighbor, avg_hist
87
88  def last_black_bar_vertical_limit(thetas: np.ndarray, moduli: np.ndarray, point:
    ↪  tuple[int, int], angle: float,
89                                    angle_modifier1: float, angle_modifier2:
                                      ↪  float):
90      STEP_DISTANCE = 2
91      box, _ = get_roi_coordinates(angle, point,
        ↪  corners_method=rec_corners_from_edge_point)
92      #add_roi_to_frame(box, (0, 100, 0))
93      thetas_roi = get_roi(thetas, point, angle, rec_corners_from_edge_point)
94      moduli_roi = get_roi(moduli, point, angle, rec_corners_from_edge_point)
95      n_angle = (astropy.stats.circmean(thetas_roi.flatten() * 2,
        ↪  weights=moduli_roi.flatten()) / 2) + angle_modifier1
96      add_point_to_frame(point, (252, 207, 3), n_angle)
97
98      cos = math.cos(n_angle + angle_modifier2)
99      sin = -math.sin(n_angle + angle_modifier2)
100     #final_point = point
101     point = (
102         point[0] + cos * STEP_DISTANCE,
103         point[1] + sin * STEP_DISTANCE,
104     )
105     while True:
106         thetas_roi = get_roi(thetas, point, n_angle, rec_corners_from_edge_point,
            ↪  half_long=15, half_short=5)
107         moduli_roi = get_roi(moduli, point, n_angle, rec_corners_from_edge_point,
            ↪  half_long=15, half_short=5)
108         mean_moduli = np.mean(moduli_roi)
109         th_moduli = 0.08
110         if mean_moduli < th_moduli or math.isnan(mean_moduli):
111             if debug:
112                 print('stopping because mean_moduli = ', mean_moduli)
113             box, _ = get_roi_coordinates(n_angle, point,
                ↪  corners_method=rec_corners_from_edge_point, half_long=15,
                ↪  half_short=5)
114             #add_roi_to_frame(box, (0, 100, 0))
115             break
116
117         ok, avg_hist = test_with_averaged_thetas_histogram(thetas_roi,
            ↪  moduli_roi, 8)
118         if not ok:
119             box, _ = get_roi_coordinates(n_angle, point,
                ↪  corners_method=rec_corners_from_edge_point, half_long=15,
                ↪  half_short=5)
120             #add_roi_to_frame(box, (0, 100, 0))
121             if debug:
122                 print('stopping because avg_hist', avg_hist)
123             break
124         #final_point = point
125         add_point_to_frame(point, (0, 255, 255))
126         point = (
127             point[0] + cos * STEP_DISTANCE,
128             point[1] + sin * STEP_DISTANCE,
129         )
130
131     return point
132
```

```
133
134 def black_bar_vertical_limit(thetas: np.ndarray, moduli: np.ndarray, point:
    ↪ tuple[int, int], angle: float,
135                              angle_modifier: float) -> tuple[int, int]:
136     STEP_DISTANCE = 8
137     n_angle = angle + angle_modifier
138     cos = math.cos(n_angle)
139     sin = -math.sin(n_angle)
140     final_point = point
141     point = (
142         point[0] + cos * STEP_DISTANCE,
143         point[1] + sin * STEP_DISTANCE,
144     )
145     while True:
146         thetas_roi = get_roi(thetas, point, angle)
147         moduli_roi = get_roi(moduli, point, angle)
148         std_thetas = astropy.stats.circstd(thetas_roi.flatten() * 2,
            ↪ weights=moduli_roi.flatten())
149         mean_moduli = np.mean(moduli_roi)
150         if mean_moduli < 0.15 or std_thetas > 1.1 or math.isnan(mean_moduli) or
            ↪ math.isnan(std_thetas):
151             break
152         final_point = point
153         add_point_to_frame(point, (0, 255, 0))
154         point = (
155             point[0] + cos * STEP_DISTANCE,
156             point[1] + sin * STEP_DISTANCE,
157         )
158
159     return final_point
160
161
162 def partial_contour(thetas: np.ndarray, moduli: np.ndarray, point: tuple[int,
    ↪ int], angle: float,
163                     angle_modifier: float) -> np.ndarray:
164     STEP_DISTANCE = 8
165     MAX_STEPS_UNTIL_READJUSTMENT = 8
166
167     n_angle = angle + angle_modifier
168
169     cos = math.cos(n_angle)
170     # sign is inverted as the y axis is inverted
171     sin = -math.sin(n_angle)
172
173     box = np.empty((0, 2))
174     final_point = point
175     point = (
176         point[0] + cos * STEP_DISTANCE,
177         point[1] + sin * STEP_DISTANCE,
178     )
179     steps_until_readjustment = 0
180     recalculate_angle = False
181
182     while True:
183         thetas_roi = get_roi(thetas, point, n_angle)
184         moduli_roi = get_roi(moduli, point, n_angle)
185
```

```
        std_thetas = astropy.stats.circstd(thetas_roi.flatten() * 2,
        ↪  weights=moduli_roi.flatten())
        mean_moduli = np.mean(moduli_roi)
        if mean_moduli < 0.075 or std_thetas > 1.1 or math.isnan(mean_moduli) or
        ↪  math.isnan(std_thetas):
            break
        if recalculate_angle:
            nn_angle = (astropy.stats.circmean(thetas_roi.flatten() * 2,
                                               weights=moduli_roi.flatten()) / 2)
                                               ↪  + angle_modifier
            recalculate_angle = False
            if math.fabs(nn_angle - n_angle) > math.pi / 2:
                continue
            n_angle = nn_angle
            cos = math.cos(n_angle)
            sin = -math.sin(n_angle)
            add_point_to_frame(point, (0, 0, 255), n_angle)
            box, _ = get_roi_coordinates(n_angle, point)
            #add_roi_to_frame(box, (0, 100, 0))
        else:
            add_point_to_frame(point, (255, 0, 0))
        final_point = point
        if steps_until_readjustment > 0:
            steps_until_readjustment -= 1
            point = (
                point[0] + cos * STEP_DISTANCE,
                point[1] + sin * STEP_DISTANCE,
            )
        else:
            p1 = black_bar_vertical_limit(thetas, moduli, point, n_angle,
            ↪  math.pi / 2)
            p2 = black_bar_vertical_limit(thetas, moduli, point, n_angle,
            ↪  math.pi * 1.5)
            point = (
                (p1[0] + p2[0]) / 2,
                (p1[1] + p2[1]) / 2,
            )
            steps_until_readjustment = MAX_STEPS_UNTIL_READJUSTMENT
            recalculate_angle = True

    for _ in range(8):
        moduli_roi = get_roi(moduli, final_point, n_angle, half_long=4)
        thetas_roi = get_roi(thetas, final_point, n_angle, half_long=4)
        std_thetas = astropy.stats.circstd(thetas_roi.flatten() * 2,
        ↪  weights=moduli_roi.flatten())
        mean_moduli = np.mean(moduli_roi)
        add_point_to_frame(final_point, (255, 0, 255))
        if mean_moduli > 0.1 and std_thetas < 1.1:
            break
        final_point = (
            final_point[0] - cos * STEP_DISTANCE,
            final_point[1] - sin * STEP_DISTANCE,
        )
    p1 = last_black_bar_vertical_limit(thetas, moduli, final_point, n_angle,
    ↪  angle_modifier, math.pi / 2)
    p2 = last_black_bar_vertical_limit(thetas, moduli, final_point, n_angle,
    ↪  angle_modifier, math.pi * 1.5)
    return p1, p2
```

```
236
237
238 def contour_from_point(thetas: np.ndarray, moduli: np.ndarray, point: tuple[int,
    ↪  int]) -> np.ndarray:
239     a, b = square_corners_from_point(point, 32)
240     thetas_roi = thetas[a[1]:b[1], a[0]:b[0]]
241     angle = scipy.stats.circmean(thetas_roi, high=math.pi)
242
243     thetas_roi = get_roi(thetas, point, angle)
244     # Returns the angle of the bars not the code.
245     # The angle consider the x-axis 90° and the upper zone of the top of the
        ↪  image as 180°. Due to this differing from
246     # the python math lib assumption of 0° being the x-axis pointing right, when
        ↪  use as input of such library, the angle
247     # will be internally added +90°, so we can use it as the barcode's angle.
248     angle = scipy.stats.circmean(thetas_roi, high=math.pi)
249     add_point_to_frame(point, (255, 127, 255), angle)
250
251     half1 = partial_contour(thetas, moduli, point, angle, 0)
252     half2 = partial_contour(thetas, moduli, point, angle, math.pi)
253     box = np.append(
254         half1,
255         half2,
256         0
257     )
258     return np.int0(box)
259
```

Bibliografía

[1] W. N. J and B. Silver, "Classifying apparatus and method," Oct 1952.

[2] J. Swartz, "Bar code scanning - Scholarpedia," 2012. http://www.scholarpedia.org/article/Bar_code_scanning.

[3] "GS1 general specifications," standard, GS1, 2022. §5.1-§5.5. https://www.gs1.org/docs/barcodes/GS1_General_Specifications.pdf.

[4] E. Ohbuchi, H. Hanaizumi, and L. Hock, "Barcode readers using the camera device in mobile phones," in *2004 International Conference on Cyberworlds*, pp. 260–265, 2004.

[5] S. J. Russell and P. Norvig, *Artificial Intelligence: A modern approach*. Prentice Hall, 3 ed., 2009.

[6] "Papers with code: Semantic segmentation." https://paperswithcode.com/task/semantic-segmentation.

[7] A. Zamberletti, I. Gallo, M. Carullo, and E. Binaghi, "Artelab's Medium Barcode 1D Collection." http://artelab.dista.uninsubria.it/downloads/datasets/barcode/medium_barcode_1d/medium_barcode_1d.html.

[8] S. Wachenfeld, S. Terlunen, and X. Jiang, "The WWU Muenster Barcode Database." https://www.uni-muenster.de/PRIA/forschung/index.shtml.

[9] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (VOC) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, p. 303–338, 2009.

[10] J. West, D. Ventura, and S. Warnick, "A theoretical foundation for inductive transfer," in *Spring Research Presentation*.

[11] T. George Karimpanal and R. Bouffanais, "Self-organizing maps for storage and transfer of knowledge in reinforcement learning," *Adaptive Behavior*, vol. 27, p. 105971231881856, 12 2018.

[12] Google, "Ml kit: Barcode scanning." https://developers.google.com/ml-kit/vision/barcode-scanning.

[13] Scandit, "Use AR for last mile delivery optimization during peak season," Mar 2022.

[14] Y. Xiao and Z. Ming, "1D barcode detection via integrated deep-learning and geometric approach," *Applied Sciences*, vol. 9, no. 16, 2019.

[15] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development." https://agilemanifesto.org/, february 2001.

[16] "Trello." https://trello.com/.

[17] L. Torvalds, "git." https://git-scm.com/.

[18] P. S. Foundation, "Python." https://www.python.org/.

[19] JetBrains, "Pycharm." https://www.jetbrains.com/pycharm/.

[20] Oracle, "Java." https://www.java.com/es/.

[21] JetBrains, "Kotlin." https://kotlinlang.org/.

[22] Google, "Android studio." https://developer.android.com/studio.

[23] "Latex." https://www.latex-project.org/.

[24] "Overleaf." https://www.overleaf.com/.

[25] S. Beucher and C. Lantuéjoul, "Use of watersheds in contour detection," vol. 132, 01 1979.

[26] U. Ramer, "An iterative procedure for the polygonal approximation of plane curves," *Computer Graphics and Image Processing*, vol. 1, no. 3, pp. 244–256, 1972.

[27] K. Wada, "Labelme: Image Polygonal Annotation with Python." https://github.com/wkentaro/labelme.

[28] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and L. Zitnick, "Microsoft COCO: Common objects in context," in *ECCV*, European Conference on Computer Vision, September 2014.

[29] "labelme2coco: A lightweight package for converting your labelme annotations into COCO object detection format." https://github.com/fcakyon/labelme2coco.

[30] "Open container initiative (oci)." https://opencontainers.org/.

[31] "Docker desktop." https://www.docker.com/products/docker-desktop/.

[32] Microsoft, "Windows 11." https://www.microsoft.com/en-us/windows/.

[33] Microsoft, "Windows subsystem for linux 2." https://docs.microsoft.com/es-es/windows/wsl/.

[34] Nvidia, "Nvidia CUDA linux container image sources." https://hub.docker.com/r/nvidia/cudal/.

[35] "Tensorflow docker image." https://www.tensorflow.org/install/docker?hl=es-419.

[36] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2." https://github.com/facebookresearch/detectron2, 2019.

[37] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2980–2988, 2017.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Los Alamitos, CA, USA), pp. 770–778, IEEE Computer Society, jun 2016.

[39] M. Badawy, M. Siam, and M. Abdelrazek, "Shuffleseg: Real-time semantic segmentation network," 03 2018.

[40] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "Enet: A deep neural network architecture for real-time semantic segmentation," 06 2016.

[41] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6230–6239, 2017.

[42] G. Lin, A. Milan, C. Shen, and I. Reid, "Refinenet: Multi-path refinement networks for high-resolution semantic segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Los Alamitos, CA, USA), pp. 5168–5177, IEEE Computer Society, jul 2017.

[43] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," 02 2018.

[44] Google, "Tensorflow image segmentation for mobile and edge." https://www.tensorflow.org/lite/examples/segmentation/overview.

[45] Google, "Semantic image segmentation with deeplab in tensorflow." hhttps://ai.googleblog.com/2018/03/semantic-image-segmentation-with.html.

[46] "Keras." hhttps://keras.io/.

[47] "Multiclass semantic segmentation using deeplabv3+." https://keras.io/examples/vision/deeplabv3_plus/.

[48] Google, "Tensorflow." https://www.tensorflow.org/.

[49] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[50] E. Zakirov, "Keras implementation of deeplabv3+." https://github.com/bonlime/keras-deeplab-v3-plus.

[51] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[52] Google, "Tensorflow lite." https://www.tensorflow.org/lite.

[53] Google, “Conversor de tensorflow lite.” https://www.tensorflow.org/lite/convert/index?hl=es-419.

[54] Google, “Post-training integer quantization.” https://www.tensorflow.org/lite/performance/post_training_integer_quant#convert_using_integer-only_quantization.

[55] Google, “Image segmentation android sample.” https://github.com/tensorflow/examples/tree/master/lite/examples/image_segmentation/android.

[56] toniz, “Tensorflow lite segmention android demo.” https://github.com/toniz/deeplab-on-android.