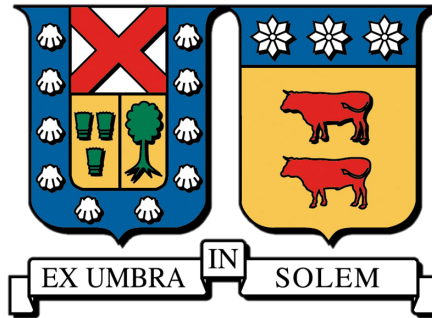


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE



**“MÓDULO DE VISIÓN POR COMPUTADORA
PARA EVALUACIÓN VISUAL DE PLATOS EN
APLICACIÓN MÓVIL DE APRENDIZAJE
CULINARIO”**

ANAYS SOFÍA CASTILLO PASTENES

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERA
CIVIL TELEMÁTICA**

PROFESOR GUÍA:

PATRICIO OLIVARES

PROFESOR CORREFERENTE:

NICOLÁS JARA

JULIO 2026



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título; ☐ Tesis de Postgrado;

Título del trabajo: MÓDULO DE VISIÓN POR COMPUTADORA PARA EVALUACIÓN VISUAL DE PLATOS EN APLICACIÓN MÓVIL DE APRENDIZAJE CULINARIO

Nombre del candidato(a): Anays Sofía Castillo Pastenes

Carrera / Grado: Ingeniería Civil Telemática

Campus: Casa Central Valparaíso; **Departamento:** Electrónica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Patricio Andrés Olivares Roncagliolo, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL

☒ El trabajo **NO contiene información que amerite confidencialidad** y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (embargo) por:

☐ 6 meses; ☐ 12 meses; ☐ 2 años; ☐ 3 años; ☐ 5 años; ☐ 10 años

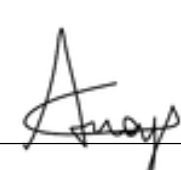
Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: _____ ; Firma: 

Estudiante o Candidato(a):

Fecha: 09/06/2026 ; Firma: 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Agradecimientos

Deseo expresar mi más sincero agradecimiento a mi profesor guía, Patricio Olivares, por su dirección constante y su disposición a acompañarme a lo largo de todo este proceso. Su retroalimentación y su mirada crítica fueron fundamentales para orientar el enfoque técnico de este trabajo y para impulsar una mejora continua en cada etapa. Le agradezco de corazón la confianza, la paciencia y el compromiso con que guió el desarrollo de esta memoria.

Quiero agradecer también, de manera profunda, a mi familia. A mi madre Ana María y a mi hermana Nicol, por su apoyo incondicional y por la confianza que siempre depositaron en mí, incluso cuando yo misma dudaba.

De manera muy especial, dedico este trabajo a mi padre, Juan Carlos, quien sembró en mí la curiosidad por aprender y el interés por la tecnología. De él heredé la fascinación por entender cómo funcionan las cosas, y fue su ejemplo el que me impulsó a entrar a la universidad y a seguir este camino. Aunque partió justo antes de que este trayecto comenzara, nunca dejé de sentir su presencia. Lo llevé conmigo en cada logro que celebré, en cada caída, y en cada pequeño paso que fui dando. Cada vez que algo me pareció demasiado difícil, encontré en su recuerdo la fuerza para continuar. Sé que habría estado orgulloso de verme llegar hasta aquí, y por eso este trabajo, y todo lo que representa, también es suyo.

No habría llegado hasta aquí sin las personas que me acompañaron en el día a día de este camino. A todos mis amigos, quiero agradecerles por estar siempre presentes, tanto en los momentos en que necesité apoyo como en aquellos en que simplemente necesitaba distraerme y reír. A quienes conocí en la universidad y compartieron conmigo el peso de esta etapa, y a quienes, desde fuera de ella, nunca dejaron de acompañarme: han sido un pilar fundamental en mi vida, y espero que sigan acompañándome durante muchos años más.

Una mención especial, a mi mejor amiga de toda la vida, Joshe, quien atravesó conmigo también este proceso con su compañía, sus ánimos y sus retos cuando hacían falta. Su lealtad y su cariño, lejos de desgastarse con el tiempo, no han hecho más que crecer, y por eso le estoy profundamente agradecida.

Un lugar igual de importante ocupa Andre, mi pareja, quien en poco tiempo se volvió fundamental en esta etapa. Su apoyo llegó en un momento difícil y me dio el impulso que necesitaba para seguir adelante. Gracias por creer en mí y por ayudarme a encontrar la fuerza para continuar.

Y no puedo cerrar estos agradecimientos sin mencionar a Kiku, mi gatita, quien me acompañó en la mayoría de este proceso universitario. Estuvo presente en cada estudio, en cada desarrollo, en cada entrega y también en cada colapso, siempre ahí, siempre cerca, recordándome con su sola presencia que todo estaría bien.

Resumen

Este trabajo presenta el diseño, implementación y validación de un módulo de Visión por Computadora integrado en SHEFU, una aplicación móvil orientada al aprendizaje culinario guiado y gamificado. El módulo tiene como propósito evaluar la presentación visual del plato final preparado por el usuario, entregando un puntaje cuantitativo en escala 0–100 que alimenta el ciclo de retroalimentación pedagógica de la plataforma.

La solución adoptó un *pipeline* de dos etapas ejecutado íntegramente *on-device*: un detector YOLOv8n localiza el plato en la fotografía capturada por el usuario y extrae la región de interés, que es evaluada por un regresor EfficientNet-Lite0 entrenado mediante transferencia de aprendizaje. Ambos modelos fueron exportados a TensorFlow Lite e integrados en la aplicación Flutter sin dependencia de conectividad, preservando la privacidad del usuario.

Para el entrenamiento del regresor se construyó un *dataset* propio de 197 imágenes de completos chilenos, anotadas por tres evaluadores siguiendo una rúbrica calibrada con el chef del proyecto. Se entrenaron y compararon dos versiones del modelo sobre el mismo conjunto de *test*: la versión seleccionada para despliegue (V2, reentrenada con anotaciones trazables) obtuvo un MAE de 19,77 puntos y un RMSE de 23,46 puntos en escala 0–100, con menor sesgo sistemático de sobreestimación respecto a la versión original. El detector alcanzó un mAP50 de 0,97 y un Recall perfecto (1,0) sobre el conjunto de *test* ($n = 20$ imágenes).

La validación experimental incluyó evaluación de robustez ante variabilidad de captura (20 fotografías con dispositivo iPhone, $\sigma = 2,96$ puntos en prueba de estabilidad),

medición de latencia en dispositivo físico (≈ 2 segundos por inferencia en Samsung Galaxy A52s 5G) y registro de consumo de recursos (incremento de ≈ 23 MB de RAM durante la inferencia, representando el 5,8 % del total disponible). El artefacto TFLite del regresor ocupa 6,74 MB, compatible con distribución móvil.

Los resultados confirman la viabilidad del módulo como componente operativo dentro de SHEFU. El margen de error del regresor refleja la dificultad intrínseca de la tarea: la evaluación visual de un plato a partir de una sola fotografía es un problema subjetivo con alta variabilidad incluso entre evaluadores humanos, y el tamaño acotado del *dataset* constituye la principal limitación del prototipo. Como trabajo futuro se propone ampliar el *dataset*, incorporar métricas de acuerdo inter-evaluador, desarrollar salidas explicables basadas en atributos visuales y reducir la latencia mediante optimización del módulo de *Test Time Augmentation*.

Palabras clave: *Visión por Computadora, evaluación visual de alimentos, TensorFlow Lite, inferencia on-device, transferencia de aprendizaje, EfficientNet-Lite0, YOLOv8, aplicación móvil, Flutter, retroalimentación pedagógica*

Índice de figuras

4.1. Diagrama de interacción entre módulos del sistema SHEFU. El módulo de Visión por Computadora recibe la imagen capturada al finalizar la sesión de cocina y entrega el puntaje visual al componente de Cálculo de Score.	25
4.2. Muestra representativa de 10 preparaciones utilizada para la calibración de la evaluación visual. Estas imágenes fueron analizadas por el chef experto para establecer el estándar de calidad.	33
4.3. Arquitectura del <i>pipeline</i> de inferencia <i>on-device</i> . La evaluación se divide en dos etapas secuenciales para aislar el plato del ruido de fondo antes de calcular el puntaje visual.	36
4.4. Flujo de captura de evidencia visual desde la aplicación móvil. La interfaz guía al usuario para asegurar que el plato quede centrado en el recuadro, mejorando las condiciones de entrada para el modelo.	54
4.5. Procesamiento e inferencia <i>on-device</i> . La aplicación consume los campos del contrato de salida estructurado (<code>bbox</code> , <code>confidence</code> y <code>score_100</code>) para renderizar la validación del recorte y la puntuación final del plato.	55
5.1. Matriz de confusión del detector YOLOv8n sobre el conjunto de <i>test</i> (conteos absolutos).	59
5.2. Matriz de confusión normalizada del detector YOLOv8n sobre el conjunto de <i>test</i>	59

5.3.	Curva Precisión-Recall del detector YOLOv8n sobre el conjunto de <i>test</i> (mAP@0.5 = 0,966).	60
5.4.	Curva F1-Confianza del detector YOLOv8n sobre el conjunto de <i>test</i> . El F1 máximo de 0,94 se alcanza en umbral 0,622; el umbral de producción (0,3) prioriza recall máximo.	61
5.5.	Curvas de Precisión (izquierda) y Recall (derecha) en función del umbral de confianza sobre el conjunto de <i>test</i> . A <code>conf_th= 0,3</code> , el Recall es prácticamente 1,0.	62
5.6.	Curvas de pérdida (MSE) durante el entrenamiento de V1 (izquierda, mejor <i>checkpoint</i> en época 18) y V2 (derecha, mejor <i>checkpoint</i> en época 11). El <i>spike</i> en época 6 corresponde al descongelamiento del <i>backbone</i>	64
5.7.	<i>Scatter plot</i> de score predicho vs real para V1 y V2 sobre el mismo conjunto de <i>test</i> ($n = 20$). V1 muestra sesgo sistemático de sobreestimación (puntos concentrados por encima de la diagonal); V2 distribuye los errores en ambas direcciones.	66
5.8.	Distribución del error absoluto para V1 y V2 sobre el conjunto de <i>test</i> ($n = 20$). V1 concentra errores en rangos bajos pero tiene cola extendida hacia errores grandes; V2 presenta distribución más uniforme. . . .	67
5.9.	Ejemplos de predicción del modelo V2 sobre el conjunto de <i>test</i> . Fila superior: mejores predicciones (error <7 pts). Fila media: predicciones con error cercano al MAE (≈ 20 pts). Fila inferior: peores casos (error >35 pts).	68
5.10.	Comportamiento heterogéneo del detector ante variaciones de captura. Cada cuadro muestra el <i>bounding box</i> seleccionado junto con la confianza y el score predicho.	71

Índice de cuadros

4.1. Extracto de las anotaciones realizadas por el chef experto sobre el subconjunto de referencia, utilizadas como base para la rúbrica de etiquetado.	34
4.2. Métricas de evaluación de los meta-regresores basados en atributos visuales.	38
4.3. Hiperparámetros de entrenamiento del detector YOLO.	46
4.4. Hiperparámetros de entrenamiento del regresor EfficientNet-Lite0. . . .	47
5.1. Métricas del detector YOLOv8n sobre el conjunto de <i>test</i> (época 100). .	58
5.2. Comparación de arquitecturas de regresión sobre el conjunto de <i>test</i> (escala 0–100, $n = 20$).	62
5.3. Comparación de V1 y V2 sobre el mismo conjunto de <i>test</i> (escala 0–100, $n = 20$).	63
5.4. Métricas del modelo deployado (EfficientNet-Lite0 V2, TFLite float16) sobre el conjunto de <i>test</i> (escala 0–100, $n = 20$).	65
5.5. Evaluación de robustez del pipeline ante variaciones de captura.	69
5.6. Latencia por etapa del pipeline de inferencia (en milisegundos, $n = 12$, dispositivo Samsung Galaxy A52s 5G).	72

5.7. Consumo de recursos del proceso durante la inferencia (dispositivo Samsung Galaxy A52s 5G, modo <i>debug</i>), modo que mantiene la instrumentación necesaria para captura de logs pero introduce sobrecarga adicional respecto al modo release de producción; los valores reportados constituyen una estimación moderada de la latencia real en deployment	73
1. Completo 1: variación de ángulo.	99
2. Completo 1: variación de fondo.	99
3. Completo 2: variación de ángulo.	100
4. Completo 2: variación de fondo.	100
5. Completo 2: estabilidad.	100
6. Mediciones individuales de consumo de recursos durante la inferencia (modo <i>profile</i> , dispositivo Samsung Galaxy A52s 5G).	101

Índice general

Resumen	IV
1. Introducción	5
1.1. Objetivos	7
1.1.1. Objetivo General	7
1.1.2. Objetivos Específicos	7
1.2. Estructura	8
2. Marco Teórico	10
2.1. Contexto del sistema: aprendizaje culinario guiado y gamificado	10
2.2. Visión por Computadora aplicada a la evaluación de alimentos	11
2.3. Redes Neuronales Convolucionales y representación visual	12
2.3.1. Transferencia de aprendizaje (<i>Transfer Learning</i>)	12
2.4. Modelos eficientes para dispositivos móviles	13
2.4.1. Arquitecturas livianas: MobileNet y EfficientNet-Lite	14
2.4.2. Detección de objetos con YOLO	15
2.5. Inferencia <i>on-device</i> y privacidad	16
2.6. Formulación del problema: criterios visuales y salidas del modelo	16
2.6.1. El <i>score</i> como señal pedagógica	17
2.7. Preprocesamiento y aumentación de datos	17
2.7.1. Preprocesamiento	17
2.7.2. Aumentación (<i>Data Augmentation</i>)	18

2.7.3.	PyTorch como framework de entrenamiento	18
2.8.	Optimización y despliegue con TensorFlow Lite	19
2.8.1.	Conversión del modelo	19
2.8.2.	Cuantización	19
2.8.3.	Latencia y restricciones operacionales	19
2.9.	Métricas de evaluación para modelos de VC	20
3.	Estado del Arte	21
3.1.	Gamificación aplicada al aprendizaje culinario	21
3.2.	Personalización y asistencia inteligente en cocina	22
3.3.	Visión por Computadora aplicada a alimentos	22
3.4.	Discusión: brecha y contribución de SHEFU	23
4.	Desarrollo de la plataforma	24
4.1.	Alcance del capítulo y rol del módulo de Visión por Computadora . . .	24
4.2.	Requerimientos del módulo de Visión por Computadora	26
4.2.1.	Requerimientos del módulo (funcionales y no funcionales) . . .	27
4.3.	Diseño de la solución	31
4.3.1.	Diseño del pipeline de datos (recolección y etiquetado)	31
4.3.2.	Diseño del modelo (arquitectura y enfoque de dos etapas) . . .	36
4.3.3.	Diseño de integración (interfaces: app - modelos)	40
4.4.	Implementación del módulo de Visión por Computadora	42
4.4.1.	Preprocesamiento e inferencia	42
4.4.2.	Entrenamiento y validación	45
4.4.3.	Exportación y verificación	49
4.4.4.	Contrato de salida formal	51
4.4.5.	Visualización de la integración <i>on-device</i>	53
5.	Resultados	56
5.1.	Metodología de evaluación	56

5.2.	Evaluación del detector YOLO	57
5.2.1.	Configuración experimental	57
5.2.2.	Métricas de detección	57
5.2.3.	Análisis gráfico	58
5.2.4.	Análisis cualitativo	60
5.3.	Evaluación y selección del regresor	61
5.3.1.	Comparación de arquitecturas	61
5.3.2.	Comparación entre versiones del regresor	62
5.3.3.	Comportamiento de entrenamiento	63
5.3.4.	Criterio de selección del modelo deployado	63
5.4.	Evaluación del <i>pipeline</i> integrado (TFLite)	64
5.4.1.	Configuración experimental	64
5.4.2.	Métricas en <i>test</i>	65
5.4.3.	Análisis del <i>scatter plot</i> predicho vs real	65
5.4.4.	Distribución del error absoluto	66
5.5.	Análisis cualitativo de predicciones	67
5.5.1.	Evaluación de robustez ante variabilidad de captura	69
5.6.	Verificación de latencia en dispositivo físico	72
5.6.1.	Consumo de recursos durante la inferencia	73
5.7.	Discusión de resultados y contraste con objetivos	74
5.8.	Limitaciones	76
6.	Conclusiones	78
6.1.	Sobre el cumplimiento de los objetivos	78
6.2.	Sobre el desempeño obtenido	79
6.3.	Sobre las decisiones de ingeniería	80
6.4.	Trabajo futuro	80
6.5.	Reflexión final	81

Anexos	83
.1. Scripts de entrenamiento	83
.1.1. Entrenamiento del detector YOLOv8n	83
.1.2. Entrenamiento del regresor EfficientNet-Lite0	84
.2. Pipeline de exportación	88
.2.1. Exportación PyTorch a ONNX	88
.2.2. Verificación del artefacto TFLite	88
.3. Scripts de evaluación	91
.3.1. Comparativa V1 vs V2	91
.3.2. Evaluación de robustez	95
.4. Resultados detallados de la evaluación de robustez	98
.5. Mediciones de consumo de recursos	101

Capítulo 1

Introducción

El aprendizaje culinario constituye una habilidad práctica clave para la autonomía personal y el bienestar cotidiano. No obstante, para un amplio segmento de la población, la formación en cocina ocurre de manera informal y autodidacta, principalmente mediante recetas en línea, videos y recomendaciones de terceros. Si bien este acceso masivo a contenido ha reducido barreras de entrada, la experiencia resultante suele ser pasiva: el usuario sigue instrucciones, pero carece de mecanismos que le permitan evaluar objetivamente su desempeño, identificar errores recurrentes y mejorar de forma sistemática.

En el contexto actual, las aplicaciones móviles orientadas a cocina se han consolidado como repositorios de recetas o asistentes de ejecución [1, 2, 3], pero rara vez estructuran el aprendizaje como un proceso progresivo, personalizado y medible. En consecuencia, la falta de retroalimentación inmediata sobre el resultado final del plato, sumada a la ausencia de señales claras de avance, impacta negativamente en la motivación y continuidad. Esta brecha es especialmente relevante en usuarios que buscan aprender desde cero o fortalecer habilidades básicas, donde la incertidumbre sobre “si quedó bien” se transforma en un factor de frustración y abandono.

SHEFU surge como un proyecto multidisciplinario que propone una aplicación móvil orientada a enseñar a cocinar mediante una experiencia guiada, personalizada y

gamificada [4], con el objetivo de convertir cada preparación en una instancia de aprendizaje verificable. En este marco, la incorporación de técnicas de Visión por Computadora y aprendizaje profundo permite abordar un componente crítico del sistema: la evaluación automática del plato final a partir de evidencia visual, habilitando retroalimentación práctica y un indicador cuantitativo de desempeño integrable al flujo de progresión del usuario.

Para lograr este objetivo, la plataforma se estructura como un sistema modular compuesto por múltiples componentes que operan de forma coordinada: un módulo de **gestión de usuarios y perfil**, que administra la identidad, el nivel de experiencia y las preferencias del aprendiz; un **orquestador de sesiones de cocina**, que guía al usuario paso a paso durante la preparación de cada receta; una **heurística de recomendación de recetas**, que sugiere preparaciones factibles según el nivel, los ingredientes y los utensilios disponibles; un **quiz inicial y un quiz final**, que evalúan conocimientos previos y refuerzan los conceptos asociados a cada receta; un módulo de **Visión por Computadora**, encargado de evaluar el plato final a partir de evidencia visual; un componente de **cálculo de score**, que pondera los resultados de la evaluación visual, el quiz final y la dificultad base de la receta; un **determinador de nivel**, que clasifica al usuario según su desempeño acumulado; y un **módulo de gamificación**, que gestiona ranking, tienda de accesorios virtuales y recompensas para sostener la motivación. El desarrollo de estos componentes fue distribuido entre los integrantes del equipo del proyecto.

El presente informe se centra en el diseño, implementación y validación del módulo de Visión por Computadora de SHEFU. Este módulo tiene como propósito analizar una fotografía del plato final capturada por el usuario al término de cada sesión de cocina, y estimar un puntaje cuantitativo asociado a su presentación visual. Dicho puntaje es consumido por el módulo de cálculo de score del sistema, donde se combina con el resultado del quiz final y la dificultad base de la receta para producir la calificación integrada que alimenta la progresión del usuario.

En suma, este trabajo aborda el desafío de transformar un proceso tradicionalmente subjetivo como lo es la evaluación visual de un plato en una capacidad técnica operativa

y útil para el usuario final, contribuyendo a que el aprendizaje culinario sea más claro, continuo y motivador. A partir de esta base, los capítulos siguientes profundizan en el estado del arte, el desarrollo de la solución y los resultados experimentales obtenidos, en coherencia con los objetivos y requerimientos del proyecto.

1.1. Objetivos

1.1.1. Objetivo General

El objetivo general del proyecto SHEFU es mejorar las habilidades culinarias de personas con distintos niveles de experiencia mediante una aplicación móvil que integra aprendizaje automático, gamificación y retroalimentación guiada, estructurando el proceso de cocina como una experiencia progresiva, personalizada y medible que fomente la confianza y la autonomía en la cocina.

En el marco de este proyecto multidisciplinario, el presente trabajo corresponde al aporte individual centrado en el módulo de Visión por Computadora, cuyo alcance y objetivos específicos se detallan a continuación.

1.1.2. Objetivos Específicos

- Definir un conjunto de criterios visuales y un esquema de evaluación que permitan operacionalizar la calidad de presentación de un plato en métricas interpretables.
- Construir un dataset de imágenes de platos con variabilidad realista de captura (iluminación, fondo, ángulo y dispositivo) mediante un proceso de recolección, curación y etiquetado controlado.
- Implementar un pipeline de entrenamiento basado en transferencia de aprendizaje con arquitecturas eficientes, evaluado mediante métricas de desempeño reproducibles.

- Validar el módulo mediante evaluación cuantitativa y análisis cualitativo de casos, contrastando los resultados con los objetivos definidos.
- Exportar el modelo entrenado a TensorFlow Lite para su despliegue en dispositivos móviles, verificando su viabilidad operativa en términos de tamaño y latencia.

1.2. Estructura

Dado que este trabajo corresponde al aporte individual dentro de una memoria multidisciplinaria, el documento se organiza en torno al módulo de Visión por Computadora, contextualizando su desarrollo dentro de la arquitectura y los flujos del sistema completo cuando resulta necesario para comprender las decisiones de diseño e integración.

La estructura es la siguiente: en el **Capítulo 2 (Marco Teórico)** se presentan los conceptos y fundamentos técnicos necesarios para contextualizar el desarrollo del módulo de Visión por Computadora, incluyendo bases de aprendizaje profundo aplicado a imágenes, criterios de eficiencia para despliegue en entornos móviles y componentes relevantes para la evaluación visual de alimentos.

En el **Capítulo 3 (Estado del Arte)** se revisan soluciones, aplicaciones e investigaciones relacionadas con aprendizaje culinario asistido, evaluación visual de platos y enfoques de retroalimentación automática, estableciendo un marco comparativo que permite identificar oportunidades y justificar las decisiones técnicas adoptadas en este trabajo.

En el **Capítulo 4 (Modelo)** se describe el desarrollo de la solución correspondiente al aporte individual, detallando el diseño del módulo, la definición de criterios visuales, la estrategia de datos (recolección, curación y etiquetado), la arquitectura del modelo, el pipeline de entrenamiento y los aspectos de implementación necesarios para su integración en la aplicación móvil.

En el **Capítulo 5 (Resultados)** se presenta la validación experimental del módulo,

incluyendo la metodología de evaluación, métricas utilizadas, resultados obtenidos y su análisis, junto con una discusión orientada a interpretar el desempeño, las limitaciones observadas y las condiciones de falla relevantes.

Finalmente, en el **Capítulo 6 (Conclusiones)** se sintetizan los principales aportes del trabajo, se evalúa el cumplimiento de los objetivos planteados y se proponen líneas de trabajo futuro para mejorar la robustez, precisión e integración del módulo dentro del sistema SHEFU.

Capítulo 2

Marco Teórico

Este capítulo presenta los fundamentos conceptuales y técnicos necesarios para sustentar el desarrollo del módulo de Visión por Computadora (VC) de SHEFU, el cual evalúa una fotografía del plato final para entregar retroalimentación y un indicador de desempeño. En particular, se abordan los principios de visión aplicada a alimentos, aprendizaje profundo con redes convolucionales, estrategias de eficiencia para despliegue móvil e inferencia *on-device*, además de los enfoques de modelamiento y métricas de evaluación pertinentes para este tipo de problema.

2.1. Contexto del sistema: aprendizaje culinario guiado y gamificado

SHEFU se enmarca en el campo del aprendizaje asistido por tecnología, donde el objetivo es estructurar una experiencia educativa que sea progresiva, medible y motivadora. Desde una perspectiva pedagógica, el aprendizaje de habilidades prácticas como la cocina se beneficia de tres elementos fundamentales: guía paso a paso que reduce la carga cognitiva del aprendiz, retroalimentación inmediata sobre el resultado de cada acción, y mecanismos de motivación que sostengan la continuidad del proceso [4].

La gamificación, entendida como la aplicación de dinámicas propias de los jue-

gos en contextos no lúdicos, ha demostrado ser efectiva para aumentar el compromiso y la perseverancia en experiencias educativas, especialmente cuando se combina con indicadores de progreso claros y recompensas tangibles. En SHEFU, estos principios se materializan en un sistema de niveles, ranking y tienda de accesorios virtuales que opera sobre la base de un score de desempeño calculado al finalizar cada receta.

El score que produce el módulo de Visión por Computadora cumple precisamente esta función: no es un fin en sí mismo, sino una señal cuantitativa que cierra el ciclo pedagógico entre la ejecución de la receta y la evaluación de su resultado visual, alimentando tanto la retroalimentación al usuario como los mecanismos de progresión y gamificación de la plataforma.

2.2. Visión por Computadora aplicada a la evaluación de alimentos

La Visión por Computadora (VC) corresponde al área que permite extraer información de una escena a partir de datos visuales (imágenes o video), con el objetivo de describir, reconocer o inferir atributos relevantes del contenido. En el dominio culinario, los problemas más estudiados se concentran en: (i) reconocimiento de platos, (ii) detección de ingredientes y (iii) estimación de porciones. La evaluación de calidad visual constituye un problema menos consolidado en la literatura, ya que se aproxima a un juicio humano donde intervienen criterios subjetivos y condiciones no controladas de captura (iluminación, encuadre, fondos, variabilidad de utensilios y emplatado), lo que representa el foco central del módulo desarrollado en este trabajo.

En el contexto de SHEFU, el propósito de la VC no es identificar el plato (la receta ya es conocida por el flujo de la aplicación), sino estimar un puntaje cuantitativo asociado a la **presentación visual del resultado final** y entregarlo como señal de retroalimentación para el usuario. Esto implica formular el problema como una tarea de regresión directa sobre la imagen del plato, donde el modelo aprende a mapear carac-

terísticas visuales a un score interpretable en escala 0–100.

2.3. Redes Neuronales Convolucionales y representación visual

Las Redes Neuronales Convolucionales (CNN) constituyen el enfoque predominante para tareas de visión basadas en aprendizaje profundo. Su ventaja radica en la capacidad de aprender jerarquías de características: desde bordes y texturas en capas tempranas, hasta patrones de mayor nivel en capas profundas. Para imágenes de alimentos, estas representaciones suelen capturar señales relacionadas con colorimetría, granularidad, brillo, patrones de cocción y organización espacial del emplatado.

El aprendizaje en una CNN ocurre mediante retropropagación, proceso en el que se calcula el gradiente de la función de pérdida respecto a cada peso de la red y se actualizan los pesos en la dirección que reduce el error. Los gradientes fluyen desde la capa de salida hacia las capas más profundas del modelo.

En términos operacionales, una CNN aplica filtros convolucionales que se desplazan por la imagen generando mapas de activación, seguidos por funciones de no linealidad y mecanismos de reducción de dimensionalidad (p. ej., *pooling*). En arquitecturas modernas, la agregación final puede realizarse mediante *global average pooling* y capas densas que proyectan a las salidas requeridas.

2.3.1. Transferencia de aprendizaje (*Transfer Learning*)

En escenarios con datasets limitados o altamente especializados, es habitual emplear transferencia de aprendizaje. Esta técnica reutiliza un modelo preentrenado en un gran conjunto de datos de imágenes como ImageNet [5], que contiene millones de imágenes etiquetadas en miles de categorías, y ajusta sus pesos parcial o totalmente a la tarea objetivo. En este contexto, se denomina *backbone* a la parte del modelo que actúa como extractor de características, cuyas capas pueden mantenerse congeladas o ajustar-

se selectivamente durante el entrenamiento. Congelar el backbone durante las primeras épocas evita que los gradientes inestables de la cabeza recién inicializada perturben los pesos preentrenados, estabilizando el entrenamiento antes de propagar el ajuste hacia capas más profundas. Los beneficios principales son:

- **Convergencia más rápida:** el modelo parte desde representaciones visuales generales ya aprendidas.
- **Mejor generalización:** reduce el riesgo de sobreajuste cuando el dataset objetivo no es masivo.
- **Eficiencia de entrenamiento:** habilita el ajuste fino (*fine-tuning*) de capas seleccionadas del modelo manteniendo el resto congelado, lo que reduce el costo computacional. Cuando se descongelan todas las capas incluyendo el backbone, se denomina *fine-tuning* completo, estrategia que permite adaptar todas las representaciones aprendidas al dominio objetivo.

En el módulo VC de SHEFU, se empleó esta estrategia para el entrenamiento del regresor de presentación, partiendo desde pesos preentrenados en ImageNet y ajustándolos al dominio específico de evaluación visual de platos con un dataset acotado.

2.4. Modelos eficientes para dispositivos móviles

El despliegue de modelos en aplicaciones móviles exige restricciones adicionales: tamaño del modelo, uso de memoria, consumo energético y latencia en inferencia. Se denomina inferencia al proceso de aplicar un modelo ya entrenado sobre datos nuevos para obtener predicciones, y se denomina *edge device* a un dispositivo con capacidad de procesamiento local como un smartphone, donde la inferencia ocurre sin depender de un servidor externo. Por ello, se priorizan arquitecturas eficientes diseñadas para este tipo de dispositivos. En el prototipo de SHEFU se evaluaron dos arquitecturas de este tipo: MobileNet V3 Small [6], basada en convoluciones separables en profundidad, y

EfficientNet-Lite0 [7], una variante de EfficientNet [8] optimizada para compatibilidad con TensorFlow Lite, seleccionándose esta última como arquitectura final del regresor.

2.4.1. Arquitecturas livianas: MobileNet y EfficientNet-Lite

Las familias MobileNet reducen el costo computacional mediante convoluciones separables en profundidad, que descomponen una convolución estándar en dos operaciones más simples: una convolución por canal y una convolución puntual, logrando una reducción significativa de parámetros sin pérdida relevante de capacidad representacional. Por su parte, EfficientNet introduce un escalamiento compuesto que ajusta simultáneamente la profundidad, el ancho y la resolución de la red para maximizar eficiencia. La variante EfficientNet-Lite elimina operaciones incompatibles con el runtime de TensorFlow Lite, como Squeeze-and-Excitation, garantizando compatibilidad y rendimiento estable en dispositivos móviles. Desde un punto de vista teórico, la selección de una arquitectura eficiente se justifica por el equilibrio entre:

- **Capacidad de representación:** calidad predictiva medida en métricas de desempeño sobre el conjunto de evaluación.
- **Compatibilidad con inferencia local:** soporte nativo para exportación a TensorFlow Lite sin operaciones incompatibles con el runtime móvil.
- **Tamaño del artefacto:** espacio ocupado por el modelo exportado, relevante para la distribución de la aplicación.

En el contexto de SHEFU, la selección de la arquitectura de regresión se basó en dos criterios principales: el desempeño predictivo medido mediante MAE y RMSE sobre el conjunto de evaluación, y la compatibilidad nativa con TensorFlow Lite para garantizar inferencia estable en dispositivos móviles. La comparación experimental entre MobileNet V3 Small y EfficientNet-Lite0 se presenta en el Capítulo 5.3.

2.4.2. Detección de objetos con YOLO

La detección de objetos consiste en localizar e identificar instancias de objetos dentro de una imagen, entregando como salida coordenadas de regiones de interés (*bounding boxes*) junto con niveles de confianza asociados. La familia YOLO (*You Only Look Once*) [9] es una de las arquitecturas más utilizadas para esta tarea, caracterizada por procesar la imagen completa en una sola pasada por la red, lo que la hace especialmente eficiente para aplicaciones en tiempo real. En el módulo VC de SHEFU se utiliza YOLOv8n, la variante nano de YOLOv8, seleccionada por su tamaño reducido y compatibilidad con exportación a TFLite, cumpliendo el rol de localizar el plato en la imagen capturada y extraer la región de interés antes de la etapa de evaluación.

YOLOv8 divide la imagen de entrada en una grilla de celdas y predice bounding boxes desde cada celda. Para capturar objetos de distintos tamaños, lo hace en tres escalas distintas simultáneamente: una grilla fina para objetos pequeños y grillas más gruesas para objetos grandes. Para una imagen de 320×320 píxeles esto genera $40 \times 40 + 20 \times 20 + 10 \times 10 = 2100$ celdas de detección en total. Cada celda produce una predicción con cinco valores: las coordenadas del centro del bounding box (x, y) , sus dimensiones (w, h) y un nivel de confianza que indica la probabilidad de que haya un objeto en esa región.

Una vez generadas las 2100 predicciones candidatas, se aplica supresión no máxima (NMS, *Non-Maximum Suppression*), proceso que elimina detecciones redundantes manteniendo solo la de mayor confianza cuando múltiples celdas detectan el mismo objeto con alta superposición. En el módulo VC de SHEFU, NMS se excluye del grafo exportado (`nms=False`) y se reemplaza por una selección manual basada en umbral de confianza, lo que garantiza compatibilidad con el runtime TFLite y permite mayor control sobre el criterio de selección.

Para evaluar el desempeño del detector se utiliza el F1-score, que combina precisión y recall en una sola métrica mediante su media armónica:

$$F1 = 2 \cdot \frac{\text{Precisión} \times \text{Recall}}{\text{Precisión} + \text{Recall}} \quad (2.1)$$

donde Precisión mide qué fracción de las detecciones son correctas y Recall mide qué fracción de los objetos reales fueron detectados. Un F1 alto indica buen balance entre ambas métricas, y su valor varía según el umbral de confianza utilizado para filtrar detecciones.

2.5. Inferencia *on-device* y privacidad

Un aspecto relevante en soluciones que procesan imágenes de usuarios es la privacidad. La inferencia *on-device* evita la transferencia de fotografías a un servidor, reduciendo riesgos de exposición y dependencia de red. Adicionalmente, mejora la experiencia de usuario al disminuir la latencia asociada a subida y procesamiento remoto, haciendo viable el requisito de respuesta rápida en el flujo de la aplicación. En SHEFU, el módulo VC opera *on-device* mediante TensorFlow Lite, alineándose con este criterio de privacidad y rendimiento [10].

2.6. Formulación del problema: criterios visuales y salidas del modelo

Un componente central del marco teórico es la forma de operacionalizar la “calidad visual” del plato. Existen múltiples formulaciones posibles:

- **Regresión:** el modelo estima un puntaje continuo (p. ej., 0–100) asociado a calidad global o a criterios específicos.
- **Clasificación multiclase:** el modelo asigna una categoría discreta (p. ej., “bueno/regular/malo”).
- **Clasificación multi-etiqueta (multi-label):** el modelo predice múltiples atributos binarios en paralelo (p. ej., presencia/ausencia de un problema visual), junto con probabilidades asociadas.

Para el prototipo de SHEFU se adoptó la formulación de regresión directa, considerando que las alternativas de clasificación multi-etiqueta y multiclase requieren mayor volumen de datos para garantizar representatividad por criterio y categoría. Con un *dataset* acotado, la regresión directa ofrece mayor estabilidad de entrenamiento y compatibilidad con inferencia *on-device*. La justificación experimental de esta decisión se presenta en el Capítulo 4.

2.6.1. El *score* como señal pedagógica

En un sistema educativo, el *score* cumple una función pedagógica y motivacional: permite medir progreso, identificar áreas de mejora y activar mecanismos de gamificación. En este contexto, la salida del módulo VC no es un fin en sí mismo sino una señal cuantitativa que alimenta el sistema de retroalimentación de SHEFU, contribuyendo a cerrar el ciclo de aprendizaje entre la ejecución de la receta y la evaluación del resultado final.

2.7. Preprocesamiento y aumentación de datos

En VC aplicada a usuarios finales, la robustez depende críticamente del manejo de variabilidad de captura. Dos componentes teóricos relevantes son:

2.7.1. Preprocesamiento

El preprocesamiento busca estandarizar las entradas al modelo. En términos generales, incluye:

- Redimensionamiento a la resolución esperada por la red.
- Normalización de valores (p. ej., escalamiento a rangos estándar).
- Estandarización de formato de color (RGB) y correcciones básicas.

El redimensionamiento se realiza mediante interpolación bilineal, técnica que calcula el valor de cada píxel nuevo promediando los píxeles vecinos de la imagen original, preservando la continuidad visual

2.7.2. Aumentación (*Data Augmentation*)

La aumentación aplica transformaciones que simulan condiciones reales para mejorar generalización. Entre las más relevantes para fotografías de platos destacan:

- Variaciones de brillo/contraste (iluminación no controlada).
- Rotaciones y recortes suaves (encuadres distintos).
- Pequeñas traslaciones y escalamiento (distancia variable).
- Perturbaciones leves de color (balance de blancos y cámara).

El objetivo es aproximar la distribución esperada en uso real, reduciendo sensibilidad a factores ajenos a la calidad del plato.

2.7.3. PyTorch como framework de entrenamiento

PyTorch [11] es un framework de aprendizaje profundo de código abierto que provee una interfaz imperativa para la definición y entrenamiento de modelos de redes neuronales. Su ecosistema incluye herramientas como `timm` [12], que ofrece implementaciones preentrenadas de arquitecturas eficientes como EfficientNet-Lite0. Para habilitar la interoperabilidad entre frameworks de entrenamiento y runtimes de despliegue, se utiliza ONNX [13] (Open Neural Network Exchange), un formato abierto que representa modelos de redes neuronales de forma independiente del framework, permitiendo convertir un modelo entrenado en PyTorch a un formato compatible con TensorFlow Lite. En el módulo VC de SHEFU, PyTorch fue utilizado como framework de entrenamiento del regresor, con posterior exportación mediante la cadena PyTorch $\rightarrow$ ONNX $\rightarrow$ TFLite para su despliegue en la aplicación móvil.

2.8. Optimización y despliegue con TensorFlow Lite

TensorFlow Lite (TFLite) es un marco de ejecución para inferencia eficiente en dispositivos móviles y embebidos [14]. El proceso típico de despliegue considera la conversión del modelo entrenado a un artefacto `.tflite`, que fija la arquitectura y habilita optimizaciones específicas del *runtime* móvil.

2.8.1. Conversión del modelo

Una vez obtenido el artefacto `.tflite`, la arquitectura del modelo queda fija y lista para su integración en aplicaciones móviles sin dependencia de frameworks de entrenamiento.

2.8.2. Cuantización

La cuantización reduce el tamaño del modelo y acelera la inferencia al representar pesos y/o activaciones con menor precisión numérica. Existen distintos niveles: la cuantización de pesos a float16 reduce el tamaño del artefacto aproximadamente a la mitad con impacto mínimo en precisión, mientras que la cuantización INT8 ofrece mayor reducción pero requiere un proceso de calibración con datos representativos para evitar degradación de desempeño. En el prototipo de SHEFU se aplicó cuantización de pesos a float16, obteniendo un modelo de 6,74 MB. La evaluación de cuantización INT8 queda propuesta como trabajo futuro.

2.8.3. Latencia y restricciones operacionales

La latencia de inferencia se evalúa típicamente con percentiles (p50/p95) sobre lotes representativos de imágenes. En aplicaciones interactivas, mantener tiempos acotados es clave para no romper el flujo de usuario. Asimismo, es relevante controlar el tamaño final del paquete (aplicación + assets + modelo), y el consumo de memoria durante la ejecución.

2.9. Métricas de evaluación para modelos de VC

Para la tarea de regresión continua de score implementada en el módulo VC de SHEFU, las métricas de evaluación utilizadas son el error absoluto medio (MAE) y la raíz del error cuadrático medio (RMSE), definidas como:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i| \quad (2.2)$$

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2} \quad (2.3)$$

donde $\hat{y}_i$ corresponde a la predicción del modelo e y_i al valor real en escala 0–100. El MAE reporta el error promedio sin penalizar casos extremos, mientras que el RMSE aplica mayor peso a errores grandes por su naturaleza cuadrática, siendo más sensible a predicciones muy alejadas del valor real. Durante el entrenamiento se utilizó el error cuadrático medio (MSE) como función de pérdida:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (2.4)$$

cuya minimización es consistente con la optimización de RMSE y favorece la penalización de predicciones muy alejadas del score real.

Capítulo 3

Estado del Arte

Este capítulo revisa el estado del arte relacionado con soluciones digitales de cocina y enfoques de Inteligencia Artificial aplicados al aprendizaje culinario, con foco en el componente que sustenta el aporte individual de esta memoria: la evaluación del plato final mediante Visión por Computadora. La revisión se organiza en tres ejes: (i) gamificación y aprendizaje culinario en aplicaciones móviles, (ii) personalización y asistencia inteligente en cocina, y (iii) Visión por Computadora para reconocimiento y evaluación visual de alimentos.

3.1. Gamificación aplicada al aprendizaje culinario

La gamificación ha demostrado ser una estrategia efectiva para aumentar el compromiso y la continuidad en experiencias educativas, especialmente cuando se incorporan mecanismos de progreso, recompensas y desafíos. En particular, estudios clásicos de gamificación muestran evidencia empírica de mejoras en motivación y participación bajo ciertos diseños de mecánicas lúdicas [4]. En el dominio culinario, se observa la existencia de aplicaciones que estructuran el aprendizaje como una progresión con niveles y retos; un ejemplo es Zest, cuya experiencia se analiza desde la óptica de diseño de interfaz y mecanismos de avance [15].

No obstante, incluso cuando se integran niveles y objetivos, la mayoría de estas pro-

puestas mantiene una limitación central: la medición del progreso suele depender de actividad (completar recetas) o autoevaluación, sin una verificación objetiva del resultado final. Esto deja incompleto el ciclo formativo *práctica–evaluación–retroalimentación*, que resulta crítico cuando se busca mejorar una habilidad práctica.

3.2. Personalización y asistencia inteligente en cocina

La personalización de recetas y asistencia contextual ha sido abordada principalmente desde la adaptación de contenidos a las preferencias, restricciones dietéticas e inventario disponible del usuario. SideChef, por ejemplo, incorporó generación de recetas a partir de fotografías de ingredientes como funcionalidad orientada a producto [1]. Plataformas como Yummly han desarrollado propuestas de personalización integradas a ecosistemas de cocina inteligente [2], mientras que Innit se ha enfocado en experiencias contextuales adaptadas al usuario y sus dispositivos [3].

En el ámbito de asistencia inteligente, existen antecedentes de patentes que describen la generación dinámica de recetas en función de recursos, historial y contexto del usuario [16]. Estas líneas confirman una tendencia: la industria ha invertido en personalización y soporte continuo. Sin embargo, gran parte de estas aproximaciones se concentra en *qué cocinar y cómo guiar*, más que en *cómo evaluar* el resultado. La evaluación automática del plato aparece menos resuelta y, cuando se aborda, suele ser tratada como comparación contra una referencia o bajo enfoques experimentales específicos.

3.3. Visión por Computadora aplicada a alimentos

El uso de Visión por Computadora en alimentos se ha desarrollado con fuerza en tareas de reconocimiento, segmentación y análisis de calidad visual, impulsado en parte por la disponibilidad creciente de datasets especializados. Entre estos, MIRcipe propone uno *stage-aware* para reconocimiento de ingredientes considerando distintas etapas

de preparación [17], lo que resulta relevante para modelar el proceso culinario aunque requiere capturas múltiples durante la ejecución de la receta.

En paralelo, el despliegue de modelos en dispositivos móviles ha sido impulsado por herramientas como TensorFlow Lite, que permiten ejecutar inferencia eficiente en el *edge*. Este punto es central en aplicaciones como SHEFU, donde la latencia y la privacidad condicionan la experiencia: ejecutar inferencia local evita la dependencia de conectividad y reduce la exposición de imágenes sensibles.

Respecto de la evaluación visual de platos, investigaciones recientes han abordado la estimación de color, condiciones de cocción y juicio de calidad visual mediante aprendizaje profundo [18, 19]. Estas líneas sostienen la viabilidad técnica de evaluar atributos como “quemado/sobrecocido”, consistencia visual o calidad aparente, aunque remarcan desafíos: variabilidad de iluminación, encuadre, fondos y subjetividad en el criterio de “buena presentación”.

3.4. Discusión: brecha y contribución de SHEFU

La revisión evidencia que existen avances relevantes en: (i) experiencias gamificadas para sostener la motivación, (ii) personalización de recetas y asistencia inteligente, y (iii) Visión por Computadora aplicada a alimentos con soporte para despliegue móvil. Sin embargo, sigue siendo limitada la integración de estas dimensiones en un sistema educativo completo que entregue retroalimentación automática y accionable del resultado final del plato en un entorno móvil accesible.

SHEFU busca cubrir esta brecha integrando guía paso a paso, motivación gamificada y evaluación visual del plato final como mecanismo de retroalimentación. Dentro del proyecto, los módulos de orquestación de sesiones, gamificación y recomendación de recetas abordan las dos primeras dimensiones, mientras que el aporte individual de esta memoria se concentra en la tercera: diseñar y validar un módulo de Visión por Computadora que traduzca evidencia visual del plato final en un puntaje cuantitativo, habilitando un ciclo de aprendizaje medible y progresivo.

Capítulo 4

Desarrollo de la plataforma

Este capítulo presenta el desarrollo técnico del módulo de Visión por Computadora dentro del contexto general de la plataforma SHEFU, abordando el alcance del trabajo, los requerimientos del módulo, las decisiones de diseño y los aspectos de implementación que permitieron su integración en la aplicación móvil.

4.1. Alcance del capítulo y rol del módulo de Visión por Computadora

SHEFU es una aplicación móvil de aprendizaje culinario desarrollada como proyecto multidisciplinario. Su flujo completo abarca desde el registro del usuario y la selección de recetas, pasando por la guía paso a paso de la preparación, hasta la evaluación final del plato, el quiz de cierre y la actualización del nivel y recompensas del usuario. Cada uno de estos componentes fue desarrollado por integrantes del equipo del proyecto. La Figura 4.1 ilustra la interacción entre los módulos del sistema.

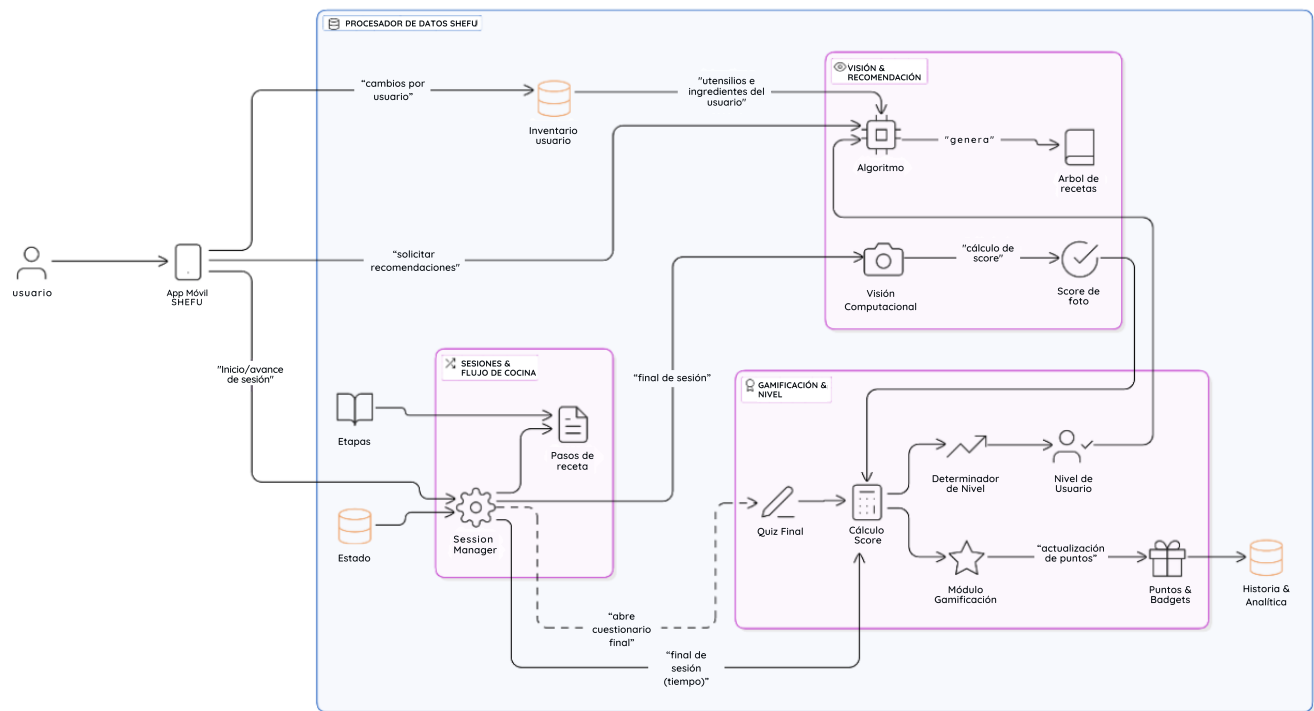


Figura 4.1: Diagrama de interacción entre módulos del sistema SHEFU. El módulo de Visión por Computadora recibe la imagen capturada al finalizar la sesión de cocina y entrega el puntaje visual al componente de Cálculo de Score.

El foco de este capítulo se concentra en el diseño, implementación y puesta en operación del componente VC, así como en las decisiones de ingeniería que permiten su integración coherente dentro de la arquitectura general.

El alcance incluye: (i) el levantamiento de requerimientos funcionales y no funcionales del módulo VC bajo restricciones propias de dispositivos móviles (latencia, consumo de recursos, tamaño de modelo y privacidad), (ii) el diseño de la solución, abarcando el pipeline de datos y el enfoque de modelamiento, (iii) la implementación del pipeline de entrenamiento y la preparación de artefactos para despliegue, incluyendo exportación a TensorFlow Lite (TFLite)[14] y verificación de inferencia, y (iv) pruebas unitarias y verificaciones técnicas orientadas a asegurar consistencia, reproducibilidad y viabilidad operativa.

Desde el punto de vista técnico, el módulo VC corresponde a un pipeline de inferencia compuesto por dos etapas principales ejecutadas en el dispositivo: una etapa de

detección que localiza el plato en la imagen capturada y extrae una región de interés (ROI), y una etapa de evaluación que analiza dicho recorte y produce un puntaje continuo interpretable para la plataforma. El detalle de cada etapa, sus modelos asociados y las decisiones de diseño se describen en las secciones siguientes.

Durante el desarrollo se exploraron alternativas de arquitectura y formulación del problema (por ejemplo, variantes de backbone y enfoques multi-salida), con el objetivo de maximizar el desempeño bajo las restricciones de despliegue móvil. La versión finalmente integrada en la aplicación se consolidó como un flujo simple y operativo *detectar* $\rightarrow$ *recortar* $\rightarrow$ *evaluar*, priorizando robustez, reproducibilidad y compatibilidad con inferencia *on-device*.

El módulo de Visión por Computadora cumple el rol de transformar evidencia visual del resultado final (una fotografía capturada por el usuario) en señales útiles para el aprendizaje: una estimación cuantitativa del desempeño visual del plato. Esta evaluación habilita mecanismos de progresión y motivación basados en desempeño, articulando el resultado visual del plato con el flujo de retroalimentación de la plataforma.

Quedan fuera del alcance de este capítulo los detalles de implementación del resto de componentes de la plataforma, salvo por las interfaces mínimas necesarias para comprender el flujo de datos y la interacción del módulo VC con la aplicación móvil. En consecuencia, cuando se mencionen otros módulos, se hará únicamente a nivel de contexto arquitectónico y de requisitos de integración.

4.2. Requerimientos del módulo de Visión por Computadora

Esta sección define los requerimientos del módulo de Visión por Computadora (VC) considerando su rol dentro de SHEFU: evaluar el plato final a partir de una fotografía y entregar un resultado cuantitativo integrable al flujo de retroalimentación y progresión del usuario. Los requerimientos se clasifican en funcionales y no funcionales, poniendo

énfasis en el contrato de entrada/salida que habilita su integración con la aplicación móvil y en las restricciones propias de ejecución *on-device*.

4.2.1. Requerimientos del módulo (funcionales y no funcionales)

El módulo de Visión por Computadora debe operar como el componente encargado de transformar la evidencia visual (la fotografía capturada por el usuario al finalizar la sesión) en un indicador de desempeño. Dado que esta captura ocurre en condiciones no controladas de iluminación, fondo y encuadre, la solución debe ser robusta ante dicha variabilidad, manteniendo a su vez viabilidad para su ejecución local en dispositivos móviles.

A continuación se detallan las especificaciones del contrato de entrada/salida, seguidas por los requisitos funcionales, las restricciones técnicas y los criterios de robustez que el módulo debe satisfacer.

Entrada/salida esperada (fotografía $\rightarrow$ *score*)

Entrada. La entrada del módulo corresponde a una fotografía digital del plato final, capturada desde la aplicación móvil. El sistema debe aceptar imágenes en formatos estándar soportados por smartphones (JPEG/PNG) y operar bajo condiciones de uso real, donde existen variaciones de iluminación, ángulo, distancia y fondo. A nivel de requisito, la imagen debe ser convertida a RGB y sometida a un preprocesamiento consistente con el utilizado en entrenamiento e integración:

- **Etapas de detección (YOLO).** La imagen capturada puede tener resolución variable según el dispositivo del usuario, por lo que se redimensiona a 320×320 píxeles mediante interpolación bilineal para garantizar compatibilidad con la entrada fija del modelo. Esta resolución fue definida durante el entrenamiento del detector (`imgsz=320`), seleccionada por su balance entre eficiencia computacional y capacidad de detección para objetos de tamaño moderado como el plato. Posteriormente se normaliza al rango $[0, 1]$ dividiendo por 255, requisito estándar de

la arquitectura YOLOv8.

- **Etapas de evaluación (EfficientNet-Lite0).** A partir del *bounding box* entregado por el detector, se extrae el recorte (ROI) correspondiente al plato en coordenadas absolutas de la imagen original. Este recorte se redimensiona a 240×240 píxeles mediante interpolación bilineal, resolución estándar requerida por la arquitectura EfficientNet-Lite0. Luego se aplica normalización por canal RGB utilizando los valores estadísticos calculados sobre el dataset ImageNet: media $[0,485, 0,456, 0,406]$ y desviación estándar $[0,229, 0,224, 0,225]$. Esta normalización es obligatoria para cualquier modelo preentrenado en ImageNet. Sin ella, el modelo recibiría una distribución de valores distinta a la que vio durante el preentrenamiento, lo que degradaría significativamente las predicciones.

Salida. La salida principal del módulo corresponde a un *score* numérico en escala 0–100 que representa la evaluación visual del plato final. Internamente, el modelo de regresión entrega una predicción continua normalizada (rango $[0, 1]$), la cual se escala a $[0, 100]$ para su interpretación por la aplicación. De manera opcional, el módulo puede exponer información auxiliar para depuración y validación técnica, como: coordenadas del *bounding box* seleccionado, nivel de confianza de detección y recorte final utilizado para inferencia.

Para propósitos de integración, el módulo debe exponer un contrato de salida estructurado que incluya al menos:

$$\{ score_{100}, status \}$$

y, opcionalmente:

$$\{ bbox, confidence \}$$

donde *status* indica éxito o error controlado. En el prototipo integrado, el puntaje

final se define directamente como:

$$score_{100} = \hat{y} \times 100$$

sin reglas adicionales de agregación, dado que la salida del regresor corresponde al indicador consumido por la aplicación. El puntaje global (0–100) fue asignado por los **tres desarrolladores del proyecto** siguiendo una **guía/rúbrica provista por el chef del proyecto**, la cual se calibró inicialmente con un conjunto de ejemplos (10 completos evaluados con el mismo esquema). Posteriormente, se realizó una **revisión consolidada** para asegurar consistencia entre anotaciones y corregir casos ambiguos. Esta guía operacionaliza criterios como tostado/apariencia general, orden/distribución, presencia/visibilidad de ingredientes y percepción de consistencia visual.

Requerimientos funcionales

- **RF1 (Captura y entrega de evidencia).** El sistema debe permitir capturar o seleccionar una fotografía del plato final desde la aplicación móvil y entregarla al módulo VC para inferencia.
- **RF2 (Detección del plato).** El módulo VC debe ejecutar la etapa de detección sobre la imagen completa para localizar el plato y obtener un ROI candidato.
- **RF3 (Selección de ROI).** El módulo debe seleccionar la detección más probable según la confianza del modelo y extraer un recorte válido (no vacío) para evaluación.
- **RF4 (Evaluación del recorte).** El módulo debe ejecutar inferencia del modelo de regresión sobre el recorte (ROI) y retornar una predicción continua escalada a 0–100.
- **RF5 (Consistencia de preprocesamiento).** El módulo debe aplicar un preprocesamiento consistente con el utilizado durante entrenamiento y exportación, ga-

garantizando equivalencia entre entrenamiento e inferencia.

- **RF6 (Salida estructurada).** El módulo debe retornar el resultado en un formato estructurado consumible por la aplicación (puntaje y estado), con posibilidad de adjuntar información auxiliar (bbox/confianza) para verificación técnica.
- **RF7 (Manejo de error controlado).** Ante entradas inválidas (imagen corrupta, recorte vacío) o fallas de procesamiento (carga de modelo, inferencia), el módulo debe retornar un estado de error controlado para que la aplicación gestione el flujo (p. ej., solicitar recaptura) sin interrupciones críticas.

Restricciones móviles (latencia, tamaño, privacidad)

El módulo debe ser viable para ejecución en dispositivos móviles, lo que implica restricciones de rendimiento y despliegue:

- **RNF1 (Latencia interactiva).** La evaluación completa (detección + recorte + regresión) debe ejecutarse en un tiempo compatible con una experiencia de usuario fluida al finalizar la sesión.
- **RNF2 (Consumo acotado de recursos).** El proceso de inferencia debe mantener un uso razonable de CPU y memoria, considerando dispositivos de gama media.
- **RNF3 (Compatibilidad de despliegue).** Los modelos deben ser distribuibles como artefactos TensorFlow Lite (`.tflite`) e integrables en la aplicación móvil basada en el framework flutter sin dependencias de servidor para inferencia.
- **RNF4 (Ejecución *on-device*).** El procesamiento debe realizarse localmente en el dispositivo (*on-device*) para reducir dependencia de conectividad y resguardar la privacidad, evitando el envío de imágenes a servidores externos.

Robustez (iluminación, fondos, ángulos)

Dado que la captura ocurre en entornos no controlados, el módulo debe mantener estabilidad ante variaciones típicas de uso:

- **RNF5 (Variabilidad de iluminación).** El desempeño debe tolerar cambios razonables de iluminación (interior/exterior, sombras, balance de blancos).
- **RNF6 (Variabilidad de fondo y contexto).** El módulo debe ser robusto ante fondos heterogéneos (mesón, platos de distintos colores, utensilios visibles), priorizando el análisis del ROI detectado.
- **RNF7 (Variabilidad de encuadre).** El desempeño debe tolerar variaciones moderadas de ángulo, distancia y rotación, mientras el plato permanezca detectable y el ROI sea representativo.
- **RNF8 (Estabilidad del puntaje).** El puntaje debe ser estable frente a variaciones pequeñas de captura, evitando oscilaciones abruptas que el usuario perciba como inconsistentes.

4.3. Diseño de la solución

Esta sección describe el diseño del módulo de Visión por Computadora (VC) como parte de la plataforma SHEFU, detallando las decisiones adoptadas para habilitar una evaluación visual del plato final con factibilidad de despliegue móvil. El diseño se organiza en tres componentes: (i) pipeline de datos, (ii) diseño del modelo y (iii) diseño de integración con la aplicación.

4.3.1. Diseño del pipeline de datos (recolección y etiquetado)

El pipeline de datos se diseñó para soportar un escenario realista de uso: imágenes capturadas por usuarios con alta variabilidad en condiciones de iluminación, encuadre

y contexto. Dado que la evaluación del plato final depende directamente de la calidad y representatividad del dataset, el diseño prioriza: (i) consistencia entre el objetivo del sistema y las etiquetas recolectadas y (ii) diversidad de condiciones de captura.

Estrategia de recolección

La recolección se definió en función del flujo de uso de SHEFU: una única fotografía del plato final al término de la sesión. En consecuencia, el dataset se compone de imágenes de **completos** terminados capturadas bajo condiciones reales de uso (interior/exterior, distintos dispositivos, distancias y fondos). Para reducir sesgos y mejorar la capacidad de generalización del modelo, se buscó incorporar variabilidad en:

- Iluminación y balance de blancos.
- Ángulos y distancia de captura.
- Tipos de plato/recipientes y fondo visible.
- Presentaciones con diferencias plausibles en el resultado (por ejemplo, variaciones en orden, distribución y visibilidad de ingredientes).

Diseño y composición del dataset

El dataset utilizado para entrenar el módulo de Visión por Computadora fue construido específicamente para el dominio del prototipo integrado: **completos** (plato final). Cada registro se asocia a una imagen y a una anotación manual registrada en formato tabular.

Para establecer un criterio de evaluación visual estandarizado y reducir el sesgo en el etiquetado, la rúbrica se calibró inicialmente con la ayuda del chef del proyecto. Se seleccionó una muestra representativa de 10 imágenes con distintas variaciones de presentación, orden, cocción y balance visual (Figura 4.2). Estas imágenes sirvieron como conjunto de anclaje (*anchor set*) para alinear las percepciones de los tres desarrolladores que posteriormente etiquetaron el resto del *dataset*.



Figura 4.2: Muestra representativa de 10 preparaciones utilizada para la calibración de la evaluación visual. Estas imágenes fueron analizadas por el chef experto para establecer el estándar de calidad.

El experto evaluó este conjunto asignando una nota general en escala 1–10 y marcando la presencia o ausencia de atributos binarios específicos (`coccion_inadecuada`, `falta_ingredient`, `desorden_presentacion`, `buen_balance_visual`, `textura_inconsistente`), además de proveer retroalimentación cualitativa. Un extracto de este proceso de calibración se detalla en la Tabla 4.1. A partir de esta retroalimentación, el equipo consolidó la guía operativa aplicada a la totalidad de las 197 imágenes.

Cuadro 4.1: Extracto de las anotaciones realizadas por el chef experto sobre el subconjunto de referencia, utilizadas como base para la rúbrica de etiquetado.

ID	Nota	Cocción	Falta	Desorden	Balance	Textura	Comentario
1	8	0	0	1	1	0	Se ve bien dorado y con buen color, pero un poco desordenado por la salsa.
2	5	1	0	1	0	1	Pan ligeramente quemado en los bordes. La palta está muy líquida.
3	10	0	0	0	1	0	Excelente presentación, atractiva y bien lograda. Proporciones perfectas.
4	3	0	1	1	0	0	Falta tomate visible. Ingredientes muy mezclados, presentación deficiente.

Dado que el flujo desplegado considera una etapa de detección previa, el proceso de entrenamiento del evaluador se diseñó para operar sobre la región de interés (*ROI*) del plato, obtenida a partir del *bounding box* entregado por el detector. Las imágenes del dataset se almacenaron como imágenes completas y fueron anotadas en dos procesos secuenciales sobre el mismo conjunto: primero se delimitaron manualmente los *bounding boxes* del plato mediante la plataforma Roboflow [20], utilizados para entrenar el detector; posteriormente se asignaron los puntajes globales de presentación por los tres evaluadores, utilizados para entrenar el regresor.

Definición de etiquetas y objetivo de aprendizaje

Para la versión final integrada en la aplicación, el objetivo de aprendizaje se definió como una tarea de **regresión directa del puntaje global**. En consecuencia, el modelo final se entrenó utilizando el campo `puntaje` (escala 0–100) como variable objetivo. Para entrenamiento, dicho puntaje se normalizó al rango $[0, 1]$ mediante:

$$y = \frac{\text{puntaje}}{100},$$

y en inferencia se re-escala a $[0, 100]$ para su presentación en la interfaz:

$$\text{score}_{100} = \hat{y} \times 100.$$

En el prototipo integrado, el contrato de salida expone únicamente el puntaje. Los atributos binarios y comentarios asociados se mantuvieron como información auxiliar para análisis, depuración y evaluación de alternativas no desplegadas (enfoques multi-salida o basados en *flags*), priorizándose finalmente un flujo operativo simple y estable *on-device* mediante regresión directa sobre el ROI detectado.

Particionado y control de calidad

Para entrenamiento y evaluación se definió un particionado del dataset en subconjuntos (entrenamiento/validación/prueba) asegurando representatividad de condiciones de captura. El dataset final contiene **197 imágenes** de completos anotadas bajo el esquema descrito, con **3 anotadores**. Para evaluación reproducible se utilizó una partición fija en **train/valid/test** definida a nivel de dataset, evitando mezclas entre subconjuntos. Adicionalmente, se aplicaron controles básicos de calidad para reducir ruido en el proceso de aprendizaje y en la etapa de inferencia, incluyendo: eliminación de imágenes corruptas o duplicadas, y descarte de casos donde el plato no es predominante en el encuadre o no es detectable por la etapa de detección.

4.3.2. Diseño del modelo (arquitectura y enfoque de dos etapas)

El diseño del módulo de Visión por Computadora se orientó a dos objetivos simultáneos: (i) lograr un desempeño predictivo suficiente para entregar retroalimentación útil al usuario y (ii) asegurar viabilidad de ejecución *on-device* en smartphones. En consecuencia, se adoptó un enfoque en **dos etapas** con modelos exportados a TensorFlow Lite, separando explícitamente la localización del plato de su evaluación visual:

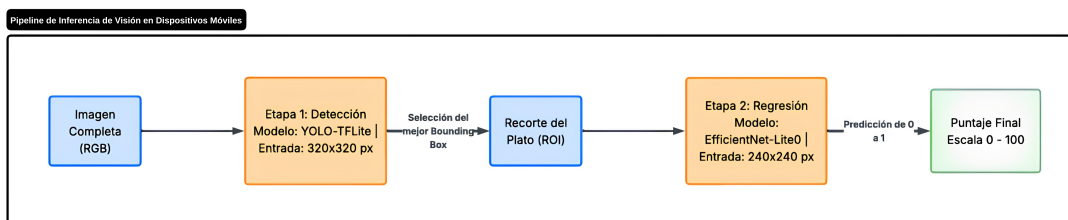


Figura 4.3: Arquitectura del *pipeline* de inferencia *on-device*. La evaluación se divide en dos etapas secuenciales para aislar el plato del ruido de fondo antes de calcular el puntaje visual.

- **Detección (YOLO-TFLite).** Procesa la imagen completa y entrega uno o más *bounding boxes* con nivel de confianza, permitiendo seleccionar y recortar la región de interés (ROI) correspondiente al plato.
- **Evaluación (EfficientNet-Lite0-TFLite).** Recibe el ROI y predice un valor continuo asociado al puntaje visual del plato.

El uso de detección previa responde a un beneficio técnico directo: **reduce el ruido de fondo y la dependencia del encuadre**, centrando la evaluación en la región relevante del plato. Esto resulta particularmente útil bajo condiciones no controladas de captura (fondos heterogéneos, utensilios visibles, variaciones de distancia y ángulo) y con un dataset acotado, donde pasarle la imagen completa al regresor lo obligaría a aprender simultáneamente a ignorar el contexto irrelevante y a evaluar la calidad de presentación, tarea que requiere muchos más ejemplos para realizarse correctamente. Al recortar el plato antes de evaluar, el regresor recibe solo la región relevante y puede

concentrarse en los atributos visuales de presentación. En términos operacionales, el flujo *detectar* $\rightarrow$ *recortar* $\rightarrow$ *evaluar* desacopla la localización del objeto de su evaluación visual, aumentando la robustez del sistema ante variabilidad de captura.

Alternativas evaluadas y criterio de selección

Previo a consolidar la arquitectura final del módulo, se evaluaron distintas alternativas de formulación y de backbone para la etapa de evaluación, buscando maximizar el desempeño bajo restricciones de despliegue móvil. Las principales variantes exploradas fueron:

- **Evaluación en una etapa (end-to-end).** Se consideró evaluar directamente sobre la imagen completa sin detección previa. Si bien este enfoque simplifica el pipeline, obliga al regresor a aprender simultáneamente a ignorar el contexto irrelevante (fondo, utensilios, encuadre) y a evaluar la calidad de presentación del plato. Con un dataset acotado y alta variabilidad de condiciones de captura, esta doble exigencia incrementa el riesgo de inestabilidad en las predicciones, por lo que se descartó en favor del enfoque de dos etapas que aísla la región del plato antes de evaluarla.
- **Backbones móviles alternativos.** Se evaluaron extractores eficientes compatibles con inferencia local, comparando alternativas del estilo MobileNet y EfficientNet-Lite0 para la etapa de evaluación, considerando el compromiso entre capacidad representacional, costo computacional y facilidad de exportación a TensorFlow Lite.
- **Salidas auxiliares (*multi-output*/atributos).** Se exploraron dos enfoques con salidas auxiliares basadas en los atributos binarios anotados. El primero consistió en un pipeline donde un modelo predice *flags* binarios de presentación y un meta-regresor los convierte en score. El segundo utilizó los embeddings internos del *backbone* (vector de 1280 dimensiones) como entrada al meta-regresor

en lugar de los *flags*. En ambos casos los resultados fueron insatisfactorios: el meta-regresor obtuvo R^2 negativo tanto en validación como en *test* (Tabla 4.2), indicando que sus predicciones son peores que predecir la media del *dataset*. Estos resultados se atribuyen al tamaño acotado del *dataset* y al desbalance entre clases en cada atributo binario, que impiden aprender predictores confiables para cada *flag* individual. Por esta razón se descartó este enfoque en favor de regresión directa sobre el puntaje global, que resultó más estable con los datos disponibles.

Cuadro 4.2: Métricas de evaluación de los meta-regresores basados en atributos visuales.

Modelo	MAE Val	MAE Test	R ² Val	R ² Test
Baseline (media)	27,1 pts	—	−0,064	—
Meta-Regresor (flags)	26,1 pts	20,5 pts	−0,006	−0,006
Meta-Regresor (embeddings)	25,2 pts	19,8 pts	−0,021	−0,037

La arquitectura finalmente integrada (detección YOLO-TFLite seguida de evaluación por regresión con EfficientNet-Lite0-TFLite) se seleccionó por ofrecer un mejor compromiso entre robustez ante variabilidad de captura, viabilidad de ejecución *on-device* y compatibilidad con despliegue mediante TensorFlow Lite. En particular, la detección previa permitió centrar la evaluación en el ROI del plato, reduciendo ruido de contexto y mejorando la consistencia del puntaje.

La comparación cuantitativa de alternativas y el análisis experimental que respalda esta decisión se presentan en el Capítulo 5

Criterios de selección de la arquitectura

La arquitectura final se seleccionó considerando criterios de ingeniería alineados con despliegue móvil, más que maximización aislada de métricas:

- **Compatibilidad con inferencia local (TFLite).** Ambos modelos deben ejecutarse sin conectividad y ser integrables como artefactos `.tflite` en la aplicación.

- **Latencia y consumo.** El pipeline completo debe mantener tiempos de respuesta compatibles con experiencia interactiva, con uso acotado de CPU/memoria en dispositivos de gama media.
- **Tamaño del modelo.** El tamaño del paquete debe ser razonable para distribución móvil.
- **Robustez a condiciones reales.** La detección previa debe mitigar variabilidad de fondo y encuadre, aumentando estabilidad del puntaje ante pequeñas variaciones de captura.

Arquitectura base y transferencia de aprendizaje

Para la etapa de evaluación se seleccionó EfficientNet-Lite0 como extractor de características, utilizando transferencia de aprendizaje con pesos preentrenados en ImageNet [5]. Sobre la salida del extractor se implementó una cabeza de regresión compuesta por una capa lineal con salida escalar seguida de activación Sigmoid, que garantiza que la predicción permanezca acotada en el rango $[0, 1]$ independiente de la entrada. Esta salida se escala posteriormente a $[0, 100]$ para su presentación en la aplicación. La elección de EfficientNet-Lite0 se justificó por su compromiso entre capacidad representacional y eficiencia computacional, manteniendo compatibilidad con exportación a TensorFlow Lite y desempeño adecuado en inferencia *on-device*.

Criterio de pérdida y entrenamiento

La tarea de regresión se entrenó con error cuadrático medio (MSELoss) como función de pérdida, seleccionado por su penalización cuadrática sobre errores grandes, lo que incentiva al modelo a evitar predicciones muy alejadas del score real. Para monitoreo de desempeño durante el entrenamiento se utilizó el error absoluto medio (MAE), más interpretable en escala 0–100.

Para mejorar la generalización frente a la variabilidad de captura del dataset se incorporaron aumentación de datos y regularización mediante weight decay. El entre-

namiento se realizó en dos fases: durante las primeras épocas el backbone se mantuvo congelado y solo se entrenó la cabeza de regresión, permitiendo que esta convergiera antes de propagar gradientes hacia capas más profundas; posteriormente se descongeló el backbone completo para fine-tuning fino. El detalle de hiperparámetros y duración de cada fase se presenta en las siguientes secciones.

4.3.3. Diseño de integración (interfaces: app - modelos)

El diseño de integración define los requerimientos de conexión entre el módulo VC y la aplicación móvil dentro del flujo de SHEFU. Se priorizó una integración desacoplada, donde los modelos se consumen como componentes locales que reciben una imagen y retornan un resultado estructurado. Los parámetros de entrada (resolución y normalización) se consideran **decisiones del prototipo** y se implementan de manera consistente en la etapa de inferencia descrita en la sección de Implementación.

Interfaz de inferencia

La interfaz de inferencia se define como un componente que:

- Recibe una imagen capturada por la aplicación (ruta/bytes).
- Ejecuta preprocesamiento para detección (prototipo: 320×320 , normalización $[0, 1]$).
- Ejecuta inferencia YOLO-TFLite y selecciona una detección válida (umbral de confianza configurable).
- Extrae el ROI y aplica preprocesamiento para regresión (prototipo: 240×240 , normalización tipo ImageNet).
- Ejecuta inferencia EfficientNet-Lite0-TFLite y retorna el resultado.

Para formalizar el contrato de salida consumido por la aplicación, la interfaz retorna un objeto con la siguiente estructura mínima:

$$\{ status, score_{100} \},$$

y, de manera opcional para soporte técnico y depuración:

$$\{ bbox, confidence \}.$$

En este contrato, `status` indica éxito o error controlado; `score_100` corresponde al puntaje final; y `bbox/confidence` describen la detección seleccionada.

Manejo de errores y condiciones inválidas

La integración considera escenarios donde la evaluación no pueda ejecutarse adecuadamente (imagen inválida, ausencia de detecciones sobre el umbral, recorte vacío, error de carga de modelo). En dichos casos, la interfaz retorna un estado de error controlado para que la aplicación gestione la contingencia (por ejemplo, solicitar recaptura o informar la falla), evitando interrupciones críticas y preservando la continuidad del flujo de usuario.

Consideraciones de despliegue

Los modelos se integran como artefactos `.tflite` dentro del paquete de la aplicación. El despliegue *on-device* evita dependencia de conectividad y reduce riesgos de privacidad, dado que las imágenes no requieren ser enviadas a servicios externos para evaluación.

4.4. Implementación del módulo de Visión por Computadora

Esta sección describe en detalle los pasos de implementación del módulo de Visión por Computadora (VC) integrado en SHEFU. Se especifican las rutinas de preprocesamiento, los hiperparámetros de entrenamiento, la cadena de exportación a TensorFlow Lite y el contrato de salida del módulo.

4.4.1. Preprocesamiento e inferencia

El *pipeline* de inferencia consta de dos etapas secuenciales ejecutadas completamente *on-device*. A continuación se describe el preprocesamiento aplicado en cada una y el flujo de decisión entre etapas. Una visión esquemática del flujo completo se muestra a continuación:

$$\text{Imagen} \rightarrow \underbrace{\text{YOLO-TFLite}}_{\text{Detección}} \rightarrow \text{ROI} \rightarrow \underbrace{\text{EffNet-Lite0-TFLite}}_{\text{Regresión}} \rightarrow \hat{y} \rightarrow \text{score}_{100} = \hat{y} \times 100$$

Etapas 1 - Detector YOLO (`best_float32.tflite`)

La imagen capturada desde la aplicación móvil se transforma a espacio de color RGB y se redimensiona a 320×320 píxeles mediante interpolación bilineal. Posteriormente, se normaliza al rango $[0, 1]$ dividiendo los valores de cada canal por 255. Como resultado, se obtiene un tensor de entrada $[1, 320, 320, 3]$ de tipo `float32`, consistente con la especificación de entrada del artefacto `.tflite`.

La inferencia del detector produce un tensor `float32` de forma $[1, 5, 2100]$, cuyos atributos por predicción corresponden a $[x, y, w, h, \text{conf}]$ en coordenadas normalizadas. El valor 2100 corresponde al número total de celdas de detección generadas por YOLOv8 para una imagen de 320×320 píxeles, producto de las tres escalas de detección de la arquitectura ($40 \times 40 + 20 \times 20 + 10 \times 10 = 2100$ celdas). Dado que el modelo fue exportado sin supresión no máxima (`nms=False`), la selección de la detección más relevante se implementa a nivel de aplicación, lo que otorga mayor control

sobre el criterio de selección y evita incompatibilidades con operaciones no soportadas por el runtime TFLite.

Selección del *bounding box*

Una vez obtenidas las detecciones brutas, se filtran aquellas cuya confianza supera el umbral mínimo configurable. Las detecciones válidas se ordenan de mayor a menor confianza y se selecciona la de mayor *score* (criterio: máxima confianza sobre umbral). Si ninguna detección supera el umbral o el tensor de salida está vacío, el módulo retorna un estado de error controlado, permitiendo que la aplicación solicite al usuario una recaptura.

En el prototipo integrado se utiliza un umbral de confianza fijo `conf_th = 0,3`. Si bien el F1 máximo del detector se alcanza en umbral 0,622, se optó por un valor inferior porque en el contexto de este pipeline es más costoso no detectar un plato que detectar una región incorrecta. Un falso negativo impide completamente la evaluación del usuario, mientras que un falso positivo puede ser descartado o absorbido por la etapa de regresión posterior. Por ello se priorizó maximizar el recall aunque esto implique una leve reducción de precisión. El umbral es un parámetro configurable que requeriría re-validación ante cualquier ajuste.

Extracción y validación del ROI

A partir del *bounding box* seleccionado, se calcula el recorte de la imagen original en coordenadas de píxel absoluto. Dado que YOLO puede predecir coordenadas que excedan ligeramente los bordes de la imagen, se aplica *clipping*, que consiste en ajustar las coordenadas a los límites válidos de la imagen antes de extraer el recorte. Se verifica que el recorte resultante no sea vacío (dimensión positiva en ambos ejes); si el área resultante es nula, el módulo retorna un estado de error controlado. No se aplica *padding* (relleno de bordes con píxeles neutros) explícito al recorte; la normalización de tamaño se delega al redimensionamiento de la etapa siguiente.

Etapa 2 - Regresor EfficientNet-Lite0 (`efficientnet_lite0.tflite`)

El recorte (ROI) obtenido se redimensiona a 240×240 píxeles mediante interpolación bilineal. A continuación se aplica normalización tipo ImageNet: se sustrae la media canal a canal y se divide por la desviación estándar correspondiente, utilizando los valores estándar de ImageNet descritos en la sección anterior. El tensor resultante tiene forma $[1, 3, 240, 240]$ con `dtype float32`, donde el primer valor corresponde al tamaño del lote (1 imagen), el segundo a los canales de color (3 canales RGB), y los dos últimos a las dimensiones espaciales de la imagen (240×240 píxeles). Este preprocesamiento es idéntico al aplicado durante el entrenamiento, garantizando consistencia entre entrenamiento e inferencia.

Los valores exactos de normalización son:

$$\mu = [0,485, 0,456, 0,406]$$

$$\sigma = [0,229, 0,224, 0,225]$$

La salida del modelo regresor es un escalar de forma $[1, 1]$ con `dtype float32`, cuyo valor se encuentra en el rango $[0, 1]$. El puntaje final se obtiene re-escalando linealmente:

$$\text{score}_{100} = \hat{y} \times 100 \quad (4.1)$$

donde $\hat{y}$ es la predicción directa del modelo TFLite.

Aumentación en tiempo de inferencia (TTA)

Dado que el regresor opera sobre un dataset acotado y bajo condiciones de captura variables, la predicción de una sola pasada del ROI por EfficientNet-Lite0 puede ser sensible a pequeñas variaciones de encuadre, iluminación o ángulo. Para mitigar esta sensibilidad, en el prototipo integrado la inferencia de la etapa de regresión se realiza mediante *Test Time Augmentation* (TTA): en lugar de evaluar el ROI una única vez, se generan 8 variantes del recorte mediante transformaciones de aumentación, cada una

se evalúa de forma independiente con el modelo y las 8 predicciones resultantes se combinan en un único valor que se utiliza como $\hat{y}$ para el cálculo de `score_100`. Esta estrategia se incluyó porque, en pruebas con el prototipo, mejoró la estabilidad y consistencia del puntaje entregado ante pequeñas variaciones en la captura de la foto, a costa de un mayor tiempo de inferencia (cuantificado en la Sección 5).

Esta operación de combinación de las 8 predicciones es la única transformación aplicada sobre la salida cruda del regresor. El valor resultante es el que se expone en el contrato de salida del módulo VC.

4.4.2. Entrenamiento y validación

Entrenamiento del detector YOLO

El detector se entrenó utilizando la librería Ultralytics YOLOv8[9] con el modelo base preentrenado `yolov8n.pt` (variante *nano*). Esta inicialización corresponde a pesos preentrenados en el conjunto COCO, incorporados en el modelo base `yolov8n.pt`[9], y constituye una aplicación de transferencia de aprendizaje, análoga a la empleada en el regresor: en lugar de entrenar la red desde cero, se parte de un modelo que ya aprendió a localizar objetos genéricos y se reentrena sobre el dominio específico de los completos. A diferencia del regresor, donde el ajuste de la cabeza y el esquema de congelamiento se implementaron manualmente, en el detector la librería Ultralytics gestiona internamente la adaptación de las capas preentrenadas. El *dataset* se descargó desde Roboflow[20] (workspace: `anays`, proyecto: `dataset-comida-chilena-8wtna`, versión 1) en formato YOLOv8, el cual incluye el particionado `train/valid/test` mediante la carpeta `data.yaml`. La fuente de imágenes corresponde al dominio de completos utilizado en el módulo VC.

Los hiperparámetros de entrenamiento del detector se resumen en la Tabla 4.3.

El entrenamiento se ejecutó en **CPU** debido a errores de ejecución asociados a los controladores NVIDIA/CUDA en el equipo disponible, que impidieron utilizar la GPU (RTX 2050) de forma estable. El tamaño de batch de 8 fue seleccionado considerando

Cuadro 4.3: Hiperparámetros de entrenamiento del detector YOLO.

Parámetro	Valor
Modelo base	yolov8n.pt
imgsz	320 × 320 px
epochs (máximo)	100
batch	8
patience	10
close_mosaic	10
Dispositivo	CPU
Formato de exportación	TFLite float32 (nms=False, int8=False, dynamic=False)

las limitaciones de memoria del equipo disponible. El parámetro `close_mosaic=10` desactiva el augmentation de mosaico en las últimas 10 épocas. El mosaico es una técnica que combina 4 imágenes en una durante el entrenamiento para mejorar robustez, pero puede introducir ruido en las épocas finales cuando el modelo está próximo a converger. Desactivarlo permite que el modelo consolide con imágenes individuales antes de detenerse, mejorando las métricas finales según la documentación oficial de Ultralytics [9]. El early stopping con `patience=10` detiene el entrenamiento si el desempeño en validación no mejora durante 10 épocas consecutivas, evitando sobreajuste y reduciendo tiempo de entrenamiento innecesario.

Las métricas del detector sobre el conjunto de *test* se reportan en la Tabla 5.1 del Capítulo 5.

Entrenamiento del regresor EfficientNet-Lite0

El regresor se implementó en PyTorch [11] utilizando la librería `timm` [12] con el *backbone* `efficientnet_lite0` [7] e inicialización con pesos preentrenados en ImageNet [5]. En `timm`, la capa `classifier` corresponde a la cabeza de clasificación original del modelo, que fue reemplazada por una cabeza de regresión compuesta por una capa lineal con salida escalar seguida de activación Sigmoid, garantizando que la predicción permanezca en $[0, 1]$.

La estrategia de entrenamiento incluyó dos fases diferenciadas: (i) durante las primeras 5 épocas el *backbone* se mantuvo congelado (*freeze*) y solo se entrenó la cabeza de regresión; (ii) a partir de la época 6 se descongelaron todos los parámetros del *backbone* para realizar *fine-tuning* completo. Esta estrategia responde al tamaño acotado del dataset disponible: congelar el backbone inicialmente evita que los gradientes inestables de la cabeza recién inicializada perturben los pesos preentrenados, mientras que el descongelamiento posterior permite adaptar las representaciones del backbone al dominio específico de evaluación visual de completos.

Los hiperparámetros de entrenamiento del regresor se resumen en la Tabla 4.4.

Cuadro 4.4: Hiperparámetros de entrenamiento del regresor EfficientNet-Lite0.

Parámetro	Valor
IMG_SIZE	240 × 240 px
BATCH_SIZE	32
EPOCHS (máximo)	100
LR (learning rate inicial)	10^{-3}
Optimizador	Adam[21] (weight_decay= 10^{-4})
Función de pérdida	MSELoss (error cuadrático medio)
Scheduler	ReduceLROnPlateau (mode='min', factor=0.5, patience=3)
PATIENCE (early stopping)	10 (sobre val_loss)
SEED	42
Épocas <i>freeze</i> backbone	Épocas 1–5 (cabeza solamente)
Descongelamiento	Época 6 (<i>fine-tuning</i> completo)
Partición	Fija (columna split del dataset Roboflow)

El learning rate inicial de 10^{-3} es el valor estándar recomendado para el optimizador Adam en tareas de regresión con transfer learning. El weight decay de 10^{-4} actúa como regularización L2, penalizando pesos muy grandes y reduciendo el riesgo de sobreajuste con dataset acotado. El batch size de 32 equilibra estabilidad del gradiente y uso de memoria. Las 100 épocas máximas representan un techo generoso que en la práctica no se alcanza gracias al early stopping con patience=10, que detiene el

entrenamiento cuando el `val_loss` no mejora durante 10 épocas consecutivas. El scheduler `ReduceLROnPlateau` reduce el learning rate a la mitad (`factor=0.5`) cuando el `val_loss` se estanca durante 3 épocas, permitiendo convergencia más fina sin intervención manual.

Aumentación de datos

Aumentación del detector (Roboflow/YOLOv8). El subconjunto de entrenamiento del detector fue ampliado mediante las transformaciones de aumentación incluidas en el *pipeline* de entrenamiento de Ultralytics: `flipplr=0.5`, `hsv_h=0.015`, `hsv_s=0.7`, `hsv_v=0.4`, `translate=0.1`, `scale=0.5`, `mosaic=1.0`, `erasing=0.4`. Adicionalmente, Roboflow generó una aumentación 3x sobre las imágenes de entrenamiento al exportar el *dataset*, expandiendo el subconjunto de 137 a 409 imágenes (manteniendo fijos los 40 de validación y 20 de prueba, sin aumentar).

Aumentación del regresor. Durante el entrenamiento se aplicaron las siguientes transformaciones de aumentación sobre el subconjunto de *train*:

- `RandomHorizontalFlip` con probabilidad $p = 0,5$.
- `RandomRotation` con rango $\pm 10^\circ$.
- `ColorJitter` con `brightness= 0,15`, `contrast= 0,15`, `saturation= 0,1`, `hue= 0,02`.

Las transformaciones de validación y *test* se limitaron a `Resize(240×240)` y normalización ImageNet, sin aumentación aleatoria, para asegurar la reproducibilidad de las métricas de evaluación.

Cabe notar que las transformaciones de aumentación de PyTorch se aplican en tiempo de entrenamiento (*on-the-fly*) sobre el conjunto ya expandido por Roboflow (197 imágenes base $\rightarrow$ 409 imágenes de entrenamiento tras la aumentación 3x de Roboflow), sin generar archivos adicionales.

Métricas de entrenamiento observadas

El proceso de entrenamiento presentó el siguiente comportamiento:

- El entrenamiento se detuvo anticipadamente en la época 21 (de 100 configuradas), tras 10 épocas consecutivas sin mejora en `val_loss` (EarlyStopping activado).
- Mejor `val_loss` (MSE en rango $[0, 1]$): 0,0506, obtenido en la época 11.
- VAL MAE reportado por PyTorch (rango $[0, 1]$): 0,1906, equivalente a $\approx 19,06$ puntos en escala 0–100.

Sobre el conjunto de *test* ($n = 20$), la evaluación con el modelo TFLite arrojó:

$$\text{MAE}_{\text{test}} = 19,77 \text{ pts}, \quad \text{RMSE}_{\text{test}} = 23,46 \text{ pts} \quad (\text{escala } 0\text{--}100) \quad (4.2)$$

Dado que el conjunto de *test* contiene únicamente 20 muestras, los valores deben interpretarse con cautela; su análisis detallado, incluyendo la justificación de la selección del modelo V2 sobre V1, se presenta en el Capítulo 5.

4.4.3. Exportación y verificación

Exportación del detector YOLO a TFLite

Una vez finalizado el entrenamiento, el modelo YOLO se exporta a TFLite directamente desde Ultralytics mediante la siguiente instrucción:

```
model.export(format="tflite", nms=False, dynamic=False,
             int8=False)
```

La configuración `nms=False` excluye el bloque NMS del grafo exportado, delegando la selección del *bounding box* a código de la aplicación. Esto mantiene el grafo TFLite simple y portable, evitando potenciales incompatibilidades con operaciones específicas del runtime que pueden aparecer al incluir NMS en el grafo. El parámetro

`dynamic=False` fija la forma de entrada a `[1, 320, 320, 3]`, y `int8=False` mantiene los pesos en precisión `float32`.

El artefacto resultante es `best_float32.tflite`. Su tensor de entrada es `[1, 320, 320, 3]` `float32` y su tensor de salida es `[1, 5, 2100]` `float32`.

Exportación del regresor: PyTorch → ONNX → TF SavedModel → TFLite

Dado que PyTorch no exporta modelos directamente a formato TFLite, se implementó una cadena de conversión en cuatro etapas para habilitar la inferencia del regresor en el cliente móvil.

1. **PyTorch → ONNX.** El modelo se exportó a ONNX [13] utilizando un tensor de referencia con forma `[1, 3, 240, 240]` (canales primero) y fijando nombres estables para entrada y salida (`"input"` y `"score01"`). Se utilizó `opset_version=17`, que corresponde a la versión del conjunto de operaciones soportadas por ONNX, seleccionada para maximizar compatibilidad con `onnx2tf` en la etapa siguiente.
2. **ONNX → TF SavedModel.** El grafo ONNX se transformó a *SavedModel* mediante la herramienta `onnx2tf` [22]. Este paso intermedio es necesario porque el convertidor oficial de TFLite acepta modelos en formato TensorFlow SavedModel como entrada, no directamente desde ONNX. La herramienta preserva los nombres de entrada y salida definidos en la etapa anterior para facilitar su trazabilidad en el artefacto TFLite final.
3. **TF SavedModel → TFLite.** El *SavedModel* se convirtió a TFLite utilizando `tf.lite.TFLiteConverter`, el convertidor oficial de TensorFlow, con cuantización de pesos a `float16` mediante `tf.lite.Optimize.DEFAULT`. Esta configuración reduce el tamaño del artefacto aproximadamente a la mitad respecto a `float32` puro sin degradación de precisión, obteniendo el artefacto final `efficientnet_lite0.tflite` de 6,74 MB para despliegue en Flutter.

4. **Verificación de consistencia.** Para controlar degradación atribuible a la conversión, el modelo TFLite se validó cargándolo con `tf.lite.Interpreter` y verificando que las formas de entrada y salida coincidieran con lo esperado (`[1, 3, 240, 240]` y `[1, 1]`). Además, se comprobó consistencia numérica (salidas en rango `[0, 1]` y ausencia de valores `NaN/Inf`). Finalmente, se ejecutó inferencia sobre el conjunto *test* y se calcularon MAE/RMSE en escala 0–100, comparando contra el modelo de referencia en PyTorch como indicador de degradación por conversión. La evidencia cuantitativa y su discusión se presentan en la sección de Resultados (ecuación (4.2)).

Los scripts y llamadas específicas de exportación/conversión se incluyen como material complementario (Anexo) para asegurar reproducibilidad sin sobrecargar el cuerpo del informe.

Postura sobre cuantización

La cuantización INT8 **no se aplicó** en la versión integrada del prototipo. El regresor se exportó en float16 y solo el detector quedó en float32. Esta decisión se tomó por razones de estabilidad y para evitar la degradación de desempeño que puede introducir la cuantización sin una calibración adecuada con datos representativos. La evaluación del impacto de cuantización INT8 en latencia, tamaño del artefacto y calidad predictiva queda propuesta como trabajo futuro.

4.4.4. Contrato de salida formal

El módulo de Visión por Computadora expone un contrato de salida estructurado, consistente con el diseño establecido en la Sección 4.3.3. Los campos obligatorios son `status` y `score_100`; los campos opcionales `bbox` y `confidence` se incluyen para soporte técnico y depuración:

Listing 4.1: Contrato de salida del módulo VC.

```
{
  "status":      "ok" | "error",
  "score_100":   <float, rango [0.0, 100.0]>,
  "bbox":        [x1, y1, x2, y2],    # coordenadas en
                                     píxeles absolutos
  "confidence":  <float, rango [0.0, 1.0]>
}
```

La semántica de cada campo es la siguiente:

status

Indica si la evaluación se completó correctamente (`.ok`) o si ocurrió un error controlado (`.error`). Los escenarios que producen `.error` incluyen: imagen inválida o corrupta, ausencia de detecciones sobre el umbral de confianza configurado, recorte ROI degenerado (área nula) y fallo en la carga o ejecución de alguno de los modelos TFLite.

score_100

Puntaje visual del plato en escala 0–100, calculado como $\hat{y} \times 100$ a partir de la salida directa del regresor EfficientNet-Lite0-TFLite (ecuación (4.1)). Este es el campo principal consumido por la lógica de negocio de la aplicación SHEFU.

bbox Coordenadas del *bounding box* seleccionado por el detector, en formato $[x_1, y_1, x_2, y_2]$ en píxeles absolutos respecto a la imagen original. Permite superponer la región analizada sobre la fotografía mostrada al usuario.

confidence

Nivel de confianza de la detección YOLO seleccionada, en rango $[0, 1]$. Útil para depuración y para implementar lógica de advertencia ante detecciones de baja certeza.

En el prototipo integrado, la aplicación Flutter consume únicamente los campos `status` y `score_100` para el flujo principal del usuario. Los campos `bbox` y `confidence` se utilizan internamente para verificación técnica durante el desarrollo y las pruebas del módulo.

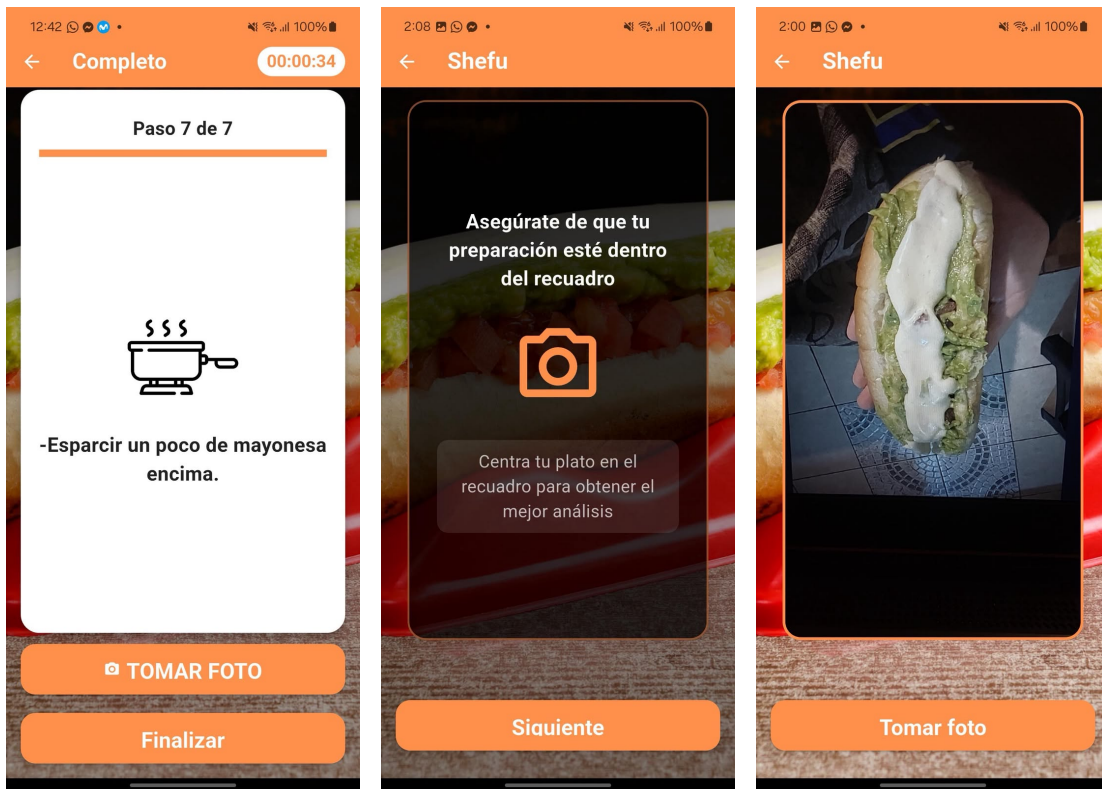
4.4.5. Visualización de la integración *on-device*

Para materializar los conceptos técnicos descritos a lo largo de este capítulo, a continuación se presenta la secuencia gráfica de la interfaz de usuario durante la evaluación del plato. Este flujo ilustra cómo la aplicación móvil consume el contrato de salida del módulo de Visión por Computadora (VC) detallado en la sección 4.4.4.

El proceso inicia una vez que el usuario finaliza los pasos de preparación de la receta (Figura 4.4a). Antes de habilitar la cámara, la aplicación despliega una capa de asistencia en pantalla (Figura 4.4b) para guiar el encuadre, mitigando proactivamente los riesgos asociados a la variabilidad de captura (RNF7). Luego, el usuario toma la fotografía de su preparación (Figura 4.4c).

Una vez capturada la imagen, se ejecuta la primera etapa del *pipeline* (detector YOLO). La Figura 4.5a muestra la pantalla de verificación, donde la aplicación hace uso de los campos opcionales `bbox` y `confidence` del contrato de salida, superponiendo el recorte extraído (ROI) e indicando la certeza de la detección (92,0 %). Esto permite al usuario confirmar que el recorte del plato es correcto o volver a tomar la foto si la detección falla.

Finalmente, si el usuario decide continuar, se ejecuta la etapa de regresión (EfficientNet-Lite0) sobre el recorte validado. El puntaje se renderiza en la pantalla de resultados del usuario (Figura 4.5b). Para efectos de gamificación y experiencia de usuario en este prototipo, el campo `score_100` entregado por el modelo se adapta a una escala visual de 0 a 10 (mostrando un 5.1 / 10), confirmando la integración exitosa del módulo dentro del ciclo de retroalimentación de la plataforma.



(a) Botón de captura

(b) Guía de encuadre

(c) Captura fotográfica

Figura 4.4: Flujo de captura de evidencia visual desde la aplicación móvil. La interfaz guía al usuario para asegurar que el plato quede centrado en el recuadro, mejorando las condiciones de entrada para el modelo.



(a) Verificación de detección (YOLO)



(b) Resultado de evaluación (EfficientNet-Lite0)

Figura 4.5: Procesamiento e inferencia *on-device*. La aplicación consume los campos del contrato de salida estructurado (`bbox`, `confidence` y `score_100`) para rendear la validación del recorte y la puntuación final del plato.

Capítulo 5

Resultados

Este capítulo presenta la validación experimental del módulo de Visión por Computadora de SHEFU, describiendo la metodología de evaluación empleada, las métricas obtenidas para cada componente del *pipeline* y el análisis de los resultados, incluyendo aciertos, condiciones de falla y limitaciones observadas. La presentación se organiza en seis ejes: (i) evaluación del detector YOLO, (ii) evaluación comparativa del regresor y selección de arquitectura, (iii) evaluación del *pipeline* integrado con el modelo final deployado, (iv) análisis cualitativo mediante ejemplos de predicción, (v) evaluación de robustez ante variabilidad de captura, y (vi) medición de latencia y consumo de recursos en dispositivo físico.

5.1. Metodología de evaluación

La evaluación del módulo se realizó en dos niveles: por componente y de forma integrada. Para el detector YOLO se utilizaron las métricas estándar de detección de objetos sobre el conjunto de validación provisto por Roboflow. Para el regresor se evaluaron y compararon dos versiones del modelo: V1 (entrenamiento original) y V2 (reentrenado con criterios del chef) sobre el mismo conjunto de *test*, garantizando consistencia de preprocesamiento en ambas evaluaciones. El modelo deployado en la aplicación corresponde a V2, seleccionado por criterios metodológicos y de desempeño que se deta-

llan en la Sección 5.3.4. Finalmente, el *pipeline* integrado se validó simulando el flujo completo de inferencia *on-device* mediante `tf.lite.Interpreter`, replicando el comportamiento del runtime TFLite utilizado por la aplicación Flutter.

En todos los casos, tanto para el detector como para el regresor, se utilizó el conjunto de *test* definido a nivel de *dataset* mediante la columna `split`, evitando mezclas con los subconjuntos de entrenamiento y validación. Las métricas de regresión se reportan en escala 0–100 para facilitar su interpretación en el contexto del sistema.

Cabe señalar que el conjunto de *test* del regresor contiene 20 imágenes, lo que limita la significancia estadística de los valores absolutos reportados. Los resultados deben interpretarse como indicadores del comportamiento del prototipo y no como estimaciones definitivas del desempeño en producción. La distribución de scores reales en el conjunto de *test* es la siguiente: 4 imágenes con score menor a 40, 10 imágenes entre 40 y 70, y 6 imágenes con score mayor a 70, con media de 56,45 puntos.

5.2. Evaluación del detector YOLO

5.2.1. Configuración experimental

El detector YOLOv8n fue entrenado durante 100 épocas sobre el *dataset* de completos obtenido desde Roboflow (409 imágenes de entrenamiento, 40 de validación, 20 de prueba), con los hiperparámetros descritos en la Tabla 4.3. La evaluación final se realizó sobre el conjunto de *test*, cargando explícitamente el *checkpoint* `best.pt` correspondiente al modelo integrado en el *pipeline* y ejecutando `model.val(split='test')` sobre las 20 imágenes de prueba.

5.2.2. Métricas de detección

Los resultados del detector al finalizar el entrenamiento se resumen en la Tabla 5.1.

El modelo alcanzó un mAP50 de 0,97, lo que indica una capacidad de detección muy alta cuando se evalúa con un umbral de IoU de 0,5. El Recall de 1,0 es espe-

Cuadro 5.1: Métricas del detector YOLOv8n sobre el conjunto de *test* (época 100).

Métrica	Valor
mAP50	0,9661
mAP50-95	0,7096
Precision	0,8904
Recall	1,0000

cialmente relevante para el rol que cumple esta etapa en el *pipeline*: dado que los falsos negativos (platos no detectados) impiden completamente la evaluación del usuario, maximizar el recall es prioritario frente a reducir los falsos positivos; sobre el conjunto de *test* el detector localizó la totalidad de los platos presentes, sin falsos negativos. Cabe señalar que este recall perfecto se observa sobre un conjunto de *test* pequeño ($n = 20$ imágenes), por lo que debe interpretarse como un indicador favorable y no como una garantía estadística de recall perfecto en producción. La Precision de 0,890 implica que en aproximadamente un 11 % de las detecciones el modelo puede identificar una región incorrecta; sin embargo, al seleccionarse únicamente la detección de mayor confianza y descartarse aquellas bajo el umbral configurado (`conf_th= 0,3`), el impacto de estos casos en la evaluación final es acotado.

El mAP50-95 de 0,710 refleja un desempeño más moderado bajo umbrales de IoU más estrictos, esperable dado el tamaño del *dataset* y la variabilidad de encuadre en las imágenes capturadas. En el contexto del *pipeline*, no se requiere precisión milimétrica del *bounding box*: el recorte del ROI incluye márgenes tolerables y la etapa de evaluación opera sobre regiones con margen suficiente para capturar el plato completo.

5.2.3. Análisis gráfico

El conjunto de *test* del detector está compuesto por 20 imágenes que totalizan 25 instancias de completo, dado que algunas fotografías contienen más de un plato. La matriz de confusión (Figura 5.1) muestra que el detector identificó correctamente las 25 instancias, sin falsos negativos (`recall = 1,0`), y registró 4 detecciones de fondo donde

no existía un completo válido (falsos positivos). Estos falsos positivos son consistentes con la precisión de 0,890 y, dado que el *pipeline* selecciona únicamente la detección de mayor confianza, su impacto sobre la evaluación final es acotado.

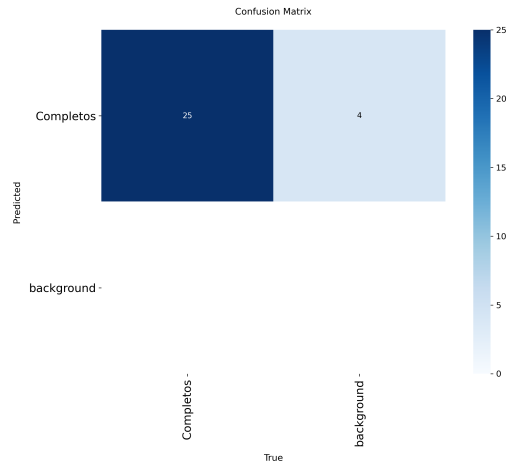


Figura 5.1: Matriz de confusión del detector YOLOv8n sobre el conjunto de *test* (conteos absolutos).

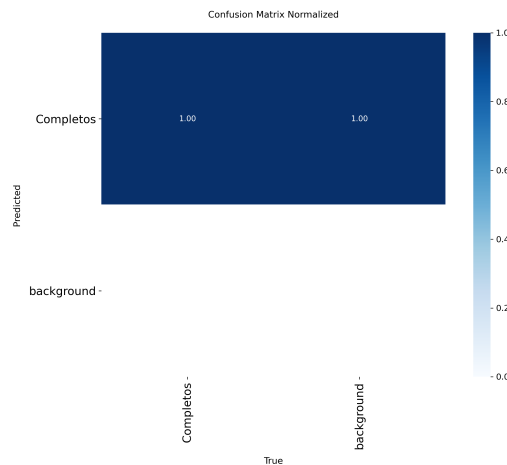


Figura 5.2: Matriz de confusión normalizada del detector YOLOv8n sobre el conjunto de *test*.

La Figura 5.3 muestra la curva Precisión-Recall del detector, con un área bajo la curva (mAP@0.5) de 0,966. La curva se mantiene en Precisión= 1,0 hasta Recall≈ 0,64, lo que confirma que el modelo mantiene alta precisión incluso cuando se exige

recuperar una parte importante de las instancias.

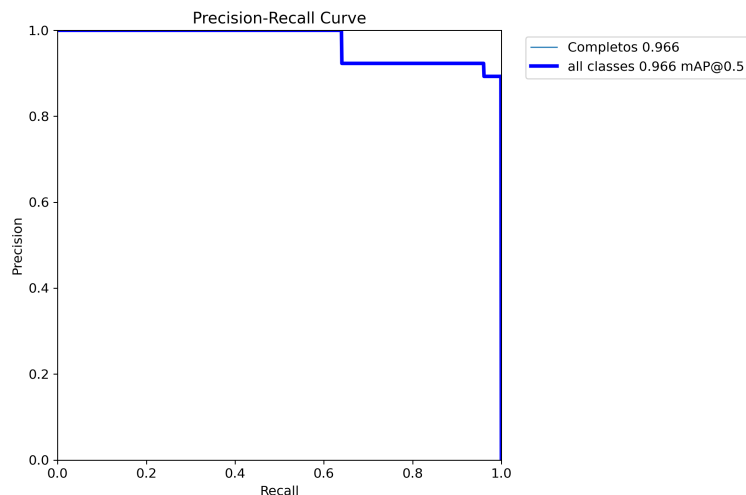


Figura 5.3: Curva Precisión-Recall del detector YOLOv8n sobre el conjunto de *test* ($\text{mAP}@0.5 = 0.966$).

La Figura 5.4 presenta la curva F1 en función del umbral de confianza. El F1 máximo de 0,94 se alcanza en umbral 0,622. El umbral de producción utilizado (`conf_th=0,3`) se sitúa por debajo del óptimo de F1, decisión que prioriza el recall máximo garantizando que prácticamente todos los platos presentes sean detectados a costa de una leve reducción de precisión, coherente con el rol de esta etapa como primer filtro del *pipeline*.

Las curvas de Precisión y Recall en función del umbral de confianza se presentan en la Figura 5.5. A `conf_th=0,3`, la Precisión se sitúa en $\approx 0,83$ y el Recall en $\approx 1,0$, confirmando que el umbral seleccionado no sacrifica detecciones reales.

5.2.4. Análisis cualitativo

El modelo presentó un desempeño consistente en condiciones típicas de captura, detectando correctamente el plato en presencia de fondos heterogéneos (mesón, superficie de cocina, utensilios visibles) y bajo variaciones moderadas de iluminación y ángulo. Los casos de falla más frecuentes correspondieron a imágenes con encuadres extremos (plato muy lateral o parcialmente fuera del *frame*) o con iluminación muy

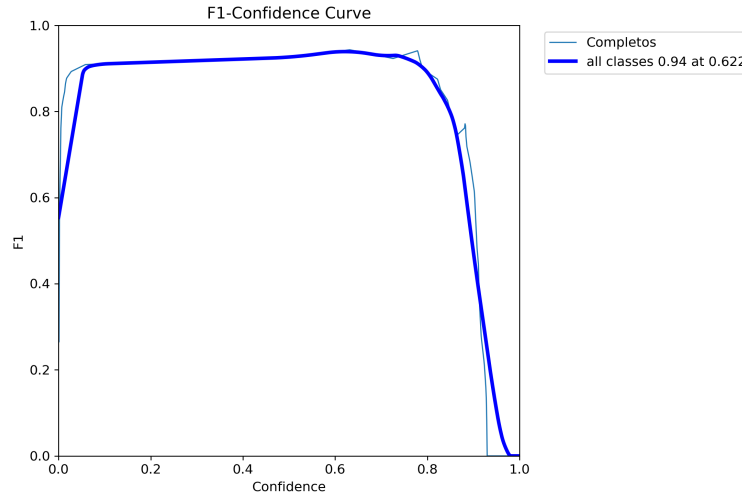


Figura 5.4: Curva F1-Confianza del detector YOLOv8n sobre el conjunto de *test*. El F1 máximo de 0,94 se alcanza en umbral 0,622; el umbral de producción (0,3) prioriza recall máximo.

deficiente, condiciones que en el flujo real de la aplicación pueden mitigarse mediante instrucciones al usuario para centrar el plato antes de fotografiar.

5.3. Evaluación y selección del regresor

5.3.1. Comparación de arquitecturas

Para la selección del *backbone* de regresión se evaluaron dos arquitecturas eficientes compatibles con TFLite: MobileNet V3 Small[6] y EfficientNet-Lite0. Ambos modelos fueron entrenados con la misma estrategia (transferencia de aprendizaje, *fine-tuning* progresivo, `MSELoss`, optimizador Adam) y evaluados sobre el mismo conjunto de *test* con el preprocesamiento correspondiente a cada arquitectura. Los resultados se presentan en la Tabla 5.2.

EfficientNet-Lite0 obtuvo mejores resultados en ambas métricas. La selección de esta arquitectura como base del regresor se fundamentó adicionalmente en criterios de ingeniería: fue diseñada específicamente para despliegue con TFLite, eliminando operaciones incompatibles con el runtime móvil, lo que simplifica la cadena de exportación

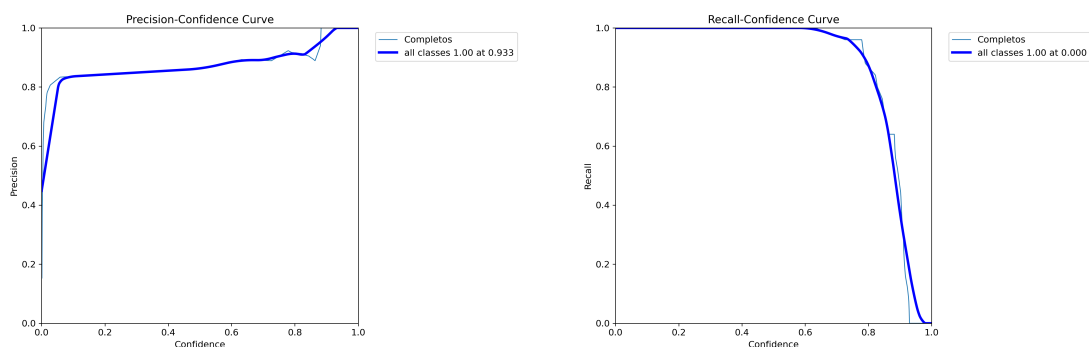


Figura 5.5: Curvas de Precisión (izquierda) y Recall (derecha) en función del umbral de confianza sobre el conjunto de *test*. A `conf_th= 0,3`, el Recall es prácticamente 1,0.

Cuadro 5.2: Comparación de arquitecturas de regresión sobre el conjunto de *test* (escala 0–100, $n = 20$).

Modelo	Resize	Parámetros	MAE	RMSE
MobileNet V3 Small	224×224	$\approx 2,5\text{M}$	19,89	26,89
EfficientNet-Lite0	240×240	$\approx 3,4\text{M}$	18,90	26,10

y garantiza mayor estabilidad en inferencia *on-device*.

5.3.2. Comparación entre versiones del regresor

Sobre la arquitectura EfficientNet-Lite0 se entrenaron dos versiones del regresor con distintos conjuntos de anotaciones:

- **V1 (original):** entrenado con el *dataset* inicial, sin identificación de evaluador.
- **V2 (reentrenado):** entrenado con las evaluaciones de tres anotadores identificados (Anays, Caro, Javier) siguiendo criterios explícitos provistos por el chef del proyecto.

Ambas versiones fueron evaluadas sobre el **mismo conjunto de *test*** ($n = 20$, distribuido equitativamente entre scores bajos, medios y altos) para garantizar comparabilidad. Los resultados se presentan en la Tabla 5.3.

Cuadro 5.3: Comparación de V1 y V2 sobre el mismo conjunto de *test* (escala 0–100, $n = 20$).

Versión	Anotaciones	MAE	RMSE
V1 - original	Sin identificar	18,90	26,10
V2 - reentrenado	Anays, Caro, Javier	19,76	23,45

La diferencia de 0,9 puntos en MAE no es estadísticamente significativa dado el tamaño del conjunto de *test*. Sin embargo, V2 obtiene un RMSE menor (23,45 vs 26,10), lo que indica que comete menos errores graves.

5.3.3. Comportamiento de entrenamiento

Las curvas de pérdida durante el entrenamiento de ambas versiones se presentan en la Figura 5.6. En V1, el mejor *checkpoint* se alcanza en la época 18 con `val_loss`= 0,048, tras una fase de oscilación prolongada en validación. En V2, la convergencia ocurre más temprano. Esto se percibe observando que hay un mejor *checkpoint* en época 11 con `val_loss`= 0,051 y que la curva de validación desciende de forma más consistente. Este comportamiento sugiere que las anotaciones con criterios guiados por el chef proveen una señal de aprendizaje más coherente, facilitando la convergencia del modelo.

En ambas versiones se observa el *spike* característico en la época 6, correspondiente al momento en que el *backbone* se descongela para *fine-tuning* completo. Este es un comportamiento esperado y coherente con la estrategia de *freeze/unfreeze* descrita en la Sección 4.4.2.

5.3.4. Criterio de selección del modelo deployado

La selección de V2 como modelo deployado en la aplicación se fundamentó en los siguientes criterios:

Menor RMSE. V2 presenta menor penalización por errores graves (23,45 vs 26,10), lo que se traduce en mayor estabilidad del puntaje ante casos difíciles. En un sistema

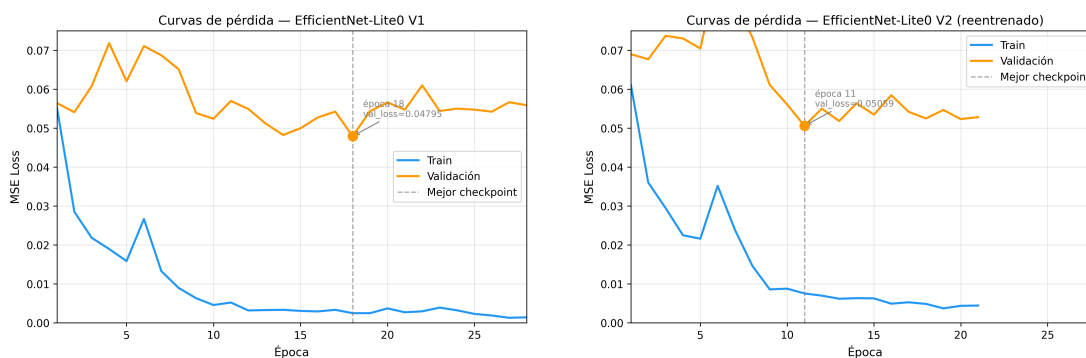


Figura 5.6: Curvas de pérdida (MSE) durante el entrenamiento de V1 (izquierda, mejor *checkpoint* en época 18) y V2 (derecha, mejor *checkpoint* en época 11). El *spike* en época 6 corresponde al descongelamiento del *backbone*.

de retroalimentación pedagógica, los errores grandes son especialmente perjudiciales para la confianza del usuario.

Ausencia de sesgo sistemático. El análisis del *scatter plot* (Figura 5.7) revela que V1 presenta un sesgo sistemático de sobreestimación: la mayoría de las predicciones quedan por encima del score real, independiente del rango. V2 distribuye los errores de forma más equitativa en ambas direcciones, lo que indica menor sesgo estructural. Un modelo que consistentemente infla los puntajes entrega retroalimentación incorrecta al usuario sobre su desempeño real.

Calidad metodológica de las anotaciones. V2 fue entrenado con evaluaciones trazables: tres anotadores identificados siguiendo una rúbrica calibrada con el chef del proyecto, lo que aporta mayor solidez metodológica al proceso de construcción del *dataset*.

5.4. Evaluación del *pipeline* integrado (TFLite)

5.4.1. Configuración experimental

La evaluación del *pipeline* integrado se realizó simulando el flujo completo de inferencia *on-device* mediante `tf.lite.Interpreter` en Python, replicando exactamente el preprocesamiento implementado en la aplicación Flutter: imagen completa →

detección YOLO-TFLite → recorte del ROI → evaluación EfficientNet-Lite0-TFLite (V2, float16) → `score_100`. El modelo TFLite deployado fue generado mediante la cadena PyTorch → ONNX → SavedModel → TFLite con cuantización de pesos a float16, obteniendo un artefacto de 6,74 MB.

5.4.2. Métricas en *test*

Los resultados del modelo deployado sobre el conjunto de *test* se resumen en la Tabla 5.4.

Cuadro 5.4: Métricas del modelo deployado (EfficientNet-Lite0 V2, TFLite float16) sobre el conjunto de *test* (escala 0–100, $n = 20$).

Métrica	Valor
MAE	19,77 pts
RMSE	23,46 pts

Las métricas del modelo TFLite son idénticas a las del *checkpoint* PyTorch V2, confirmando que la cuantización de pesos a float16 no introdujo degradación de desempeño. Las diferencias menores que pudieran observarse entre el modelo PyTorch y el TFLite en inferencia individual son atribuibles a variaciones de punto flotante del *backend* XNNPACK, sin impacto operacional.

5.4.3. Análisis del *scatter plot* predicho vs real

La Figura 5.7 presenta el *scatter plot* de score predicho versus score real para V1 y V2 sobre el mismo conjunto de *test*. Cada punto representa una imagen del conjunto de evaluación; los colores indican el error absoluto (verde = bajo, rojo = alto); la diagonal punteada representa la predicción perfecta.

El gráfico evidencia claramente el sesgo sistemático de V1: prácticamente todas las predicciones quedan por encima de la diagonal, lo que indica que el modelo infla consistentemente los puntajes. Este comportamiento es problemático para el objetivo

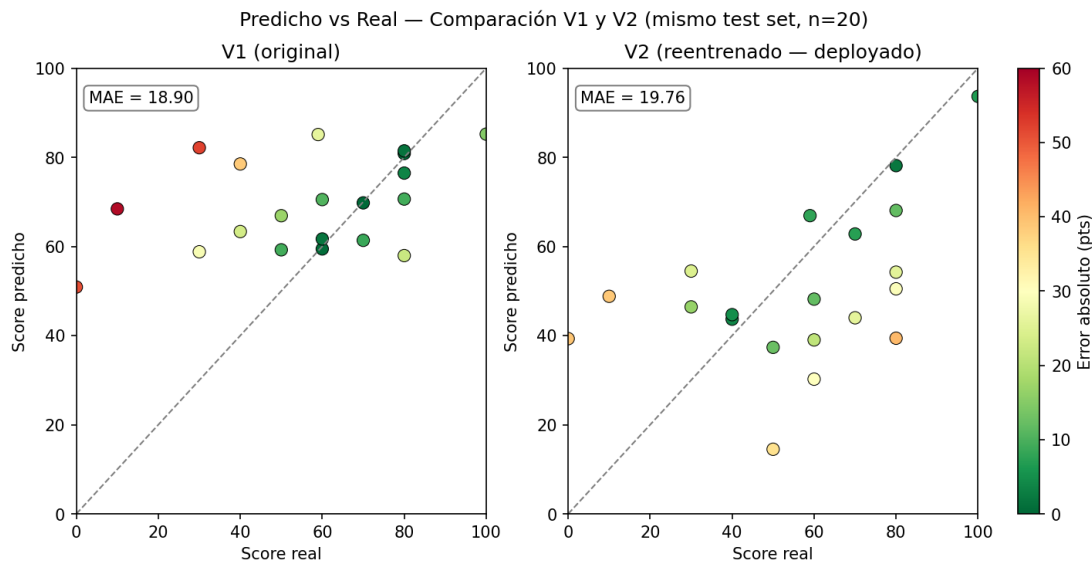


Figura 5.7: *Scatter plot* de score predicho vs real para V1 y V2 sobre el mismo conjunto de *test* ($n = 20$). V1 muestra sesgo sistemático de sobreestimación (puntos concentrados por encima de la diagonal); V2 distribuye los errores en ambas direcciones.

pedagógico del sistema, debido a que puede provocar que un usuario con un plato de baja calidad siempre recibirá una evaluación más favorable de la real.

V2, en contraste, distribuye los errores a ambos lados de la diagonal sin un patrón sistemático claro. Si bien presenta tres casos con error absoluto superior a 38 puntos, estos corresponden a imágenes con características atípicas respecto al conjunto de entrenamiento y no representan un patrón estructural del modelo.

5.4.4. Distribución del error absoluto

La Figura 5.8 presenta los histogramas de error absoluto para V1 y V2. V1 concentra la mayoría de sus predicciones en el rango de error bajo (0–10 puntos), lo que explica su menor MAE global. Sin embargo, presenta una cola extendida hacia errores altos, con cuatro casos entre 38 y 58 puntos, reflejada en su mayor RMSE. V2 muestra una distribución más uniforme, notándose en la distribución de los errores entre 0 y 41 puntos sin concentración marcada y sin valores extremos tan pronunciados como los de V1, lo cual es coherente con la ausencia de sesgo sistemático observada en el *scatter*

plot.

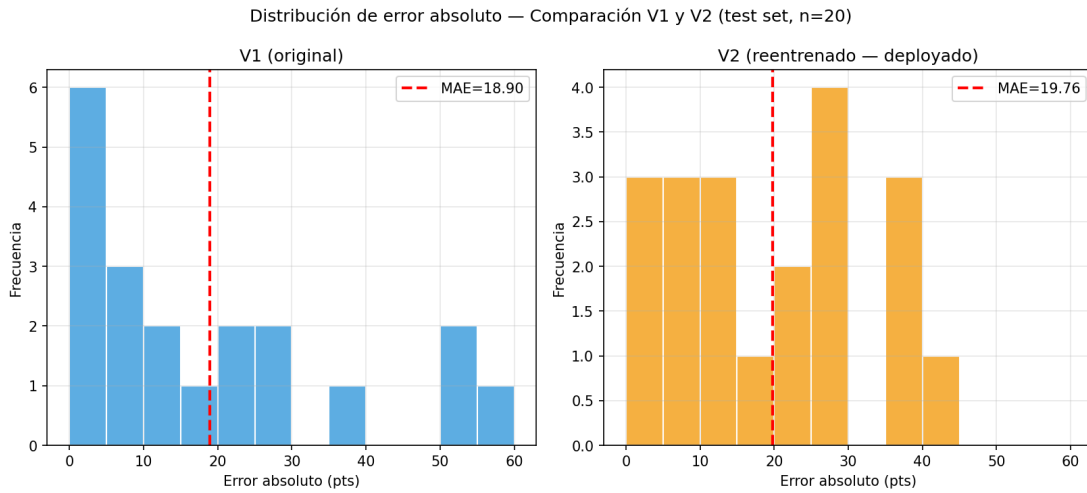


Figura 5.8: Distribución del error absoluto para V1 y V2 sobre el conjunto de *test* ($n = 20$). V1 concentra errores en rangos bajos pero tiene cola extendida hacia errores grandes; V2 presenta distribución más uniforme.

La aparente ventaja de V1 en MAE se explica parcialmente por su sesgo de sobreestimación: al predecir consistentemente hacia arriba en un conjunto de *test* con scores concentrados en el rango medio-alto (media ≈ 56), el error promedio resulta menor. V2, al no tener ese sesgo, comete errores en ambas direcciones y su MAE global es levemente mayor, aunque su distribución de errores es más representativa del comportamiento real del modelo ante datos variados.

5.5. Análisis cualitativo de predicciones

La Figura 5.9 presenta una selección de 12 imágenes del conjunto de *test*, organizadas de mejor a peor predicción. Para cada imagen se muestra el score real asignado por los evaluadores, la predicción del modelo V2 y el error absoluto. Los colores indican la calidad de la predicción: verde para error menor a 15 puntos, amarillo para error entre 15 y 25 puntos, y rojo para error superior a 25 puntos.

Los mejores casos (fila superior) corresponden a imágenes bien encuadradas con iluminación adecuada, donde el completo es claramente visible y los atributos de pre-

Ejemplos de predicción — EfficientNet-Lite0 V2 (test set)



Figura 5.9: Ejemplos de predicción del modelo V2 sobre el conjunto de *test*. Fila superior: mejores predicciones (error < 7 pts). Fila media: predicciones con error cercano al MAE (≈ 20 pts). Fila inferior: peores casos (error > 35 pts).

sentación son fácilmente discriminables. El modelo predice con error inferior a 7 puntos en estos casos.

Los casos con error cercano al MAE (fila media) presentan condiciones de captura más complejas como una iluminación irregular, ángulos poco convencionales o presentaciones con atributos visuales ambiguos. Destaca el caso con score real 30 y predicción 55: el completo tiene una presentación visualmente elaborada (abundantes ingredientes, composición organizada) que el modelo evalúa favorablemente, mientras que los evaluadores asignaron un score bajo posiblemente por criterios de cocción o proporciones que no son directamente visibles en la imagen.

Los peores casos (fila inferior) revelan las principales condiciones de falla del modelo. El caso con score real 0 y predicción 39 corresponde a un completo con ingredien-

tes atípicos (kiwi, semillas de chía) dispuestos de forma completamente desordenada. Esta es una presentación que el modelo no reconoce como baja calidad por la escasa representación de este tipo de casos en el *dataset* de entrenamiento. El caso con score real 10 y predicción 49 corresponde a una imagen en primer plano, con encuadre poco convencional, donde abundante salsa cubre gran parte de los ingredientes; pese a la baja evaluación de los anotadores, el modelo lo califica favorablemente, lo que sugiere que asocia la cobertura visual de salsas con una presentación de mayor calidad. Estos casos ilustran la principal limitación del modelo: la escasa representación de platos de baja calidad en el *dataset* limita su capacidad de discriminar en ese rango.

5.5.1. Evaluación de robustez ante variabilidad de captura

Para complementar la evaluación realizada sobre el conjunto de *test* original, se condujo una evaluación adicional con 20 fotografías capturadas con dispositivo iPhone sobre dos completos distintos, variando intencionalmente fondo, ángulo y pose. El objetivo no fue medir la precisión absoluta del modelo (ya cuantificada en la Sección 5.4.2), sino verificar la **estabilidad del puntaje** y el comportamiento del pipeline ante condiciones de captura representativas del uso real del usuario final. Los resultados se resumen en la Tabla 5.5.

Cuadro 5.5: Evaluación de robustez del pipeline ante variaciones de captura.

Completo	Condición	Score medio	Observaciones
Completo 1	Variación de ángulo	57,68	$n = 5$ imágenes
Completo 1	Variación de fondo	62,93	$n = 5$ imágenes
Completo 2	Variación de ángulo	59,45	$n = 3$ imágenes
Completo 2	Variación de fondo	59,58	$n = 4$ imágenes
Completo 2	Estabilidad	59,18	$\sigma = 2,96$, rango= 6,48
Global ($n = 20$)		59,86	Rango: 54,2 – 67,3

Nota sobre el umbral de confianza. Para esta evaluación se utilizó un umbral de confianza reducido ($\text{conf_th} = 0,05$) con el objetivo de garantizar que todas las imágenes obtuvieran una detección procesable, dado que las fotografías iPhone presentaron distribuciones de píxeles distintas al *dataset* de entrenamiento del detector. De las 20 imágenes evaluadas, 19 habrían superado igualmente el umbral de producción ($\text{conf_th} = 0,3$).

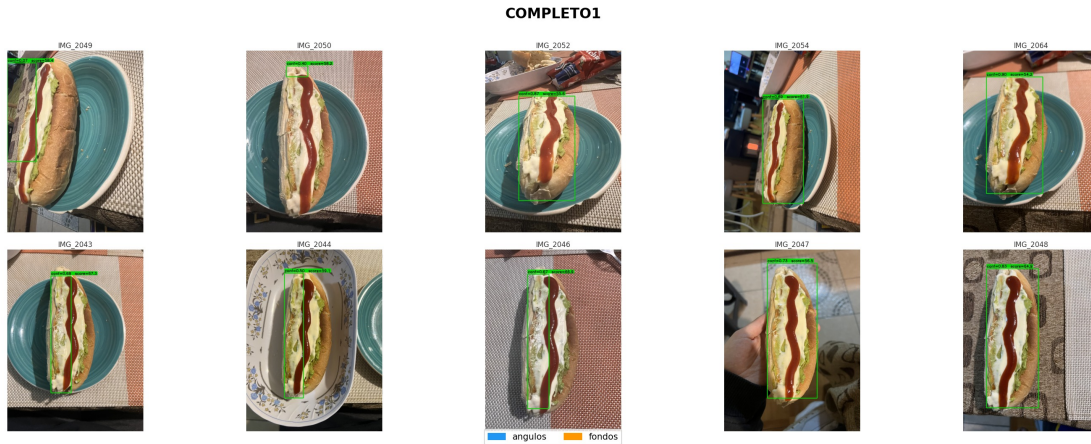
Detección. El detector localizó al plato en las 20 imágenes evaluadas, con niveles de confianza superiores a $\approx 0,4$ en 19 de ellas.

Estabilidad del puntaje. La desviación estándar del puntaje obtenida al fotografiar el mismo plato bajo variaciones mínimas fue de $\sigma = 2,96$ puntos sobre la escala 0–100, con un rango total de 6,48 puntos. Este resultado confirma que el pipeline entrega predicciones consistentes ante variaciones no intencionales del usuario, cumpliendo con el requerimiento de estabilidad definido para la experiencia de uso.

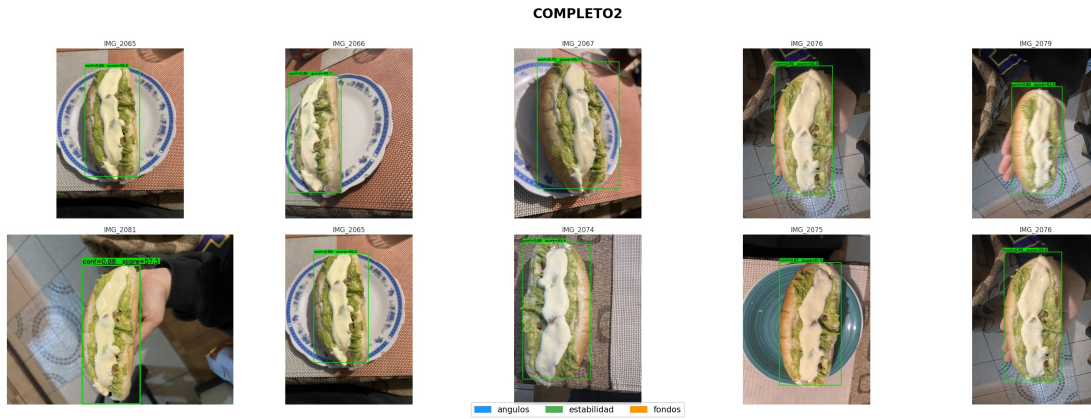
Invariancia a fondo y ángulo. Las medias de score se mantuvieron en un rango acotado al variar estos factores sobre el mismo plato, validando empíricamente el rol del detector como mecanismo de aislamiento del contexto antes de la etapa de evaluación.

Análisis cualitativo de las detecciones. La inspección visual de los *bounding boxes* generados reveló un comportamiento heterogéneo del detector entre ambos completos. En el completo 1, el detector tendió a generar recortes parciales que abarcaban solo una sección del plato en varias de las imágenes evaluadas, particularmente en las variaciones de fondo. En contraste, en el completo 2 el detector localizó correctamente el plato completo en las 10 imágenes evaluadas, con confianzas consistentemente superiores a 0,70. Cuando el detector entrega un recorte parcial, la etapa de regresión opera sobre una entrada incompleta, lo que propaga esta limitación al score final. La causa específica de este comportamiento diferenciado requeriría un análisis adicional con un conjunto mayor de imágenes que controle variables como textura, contraste interno del plato y composición visual. La Figura 5.10 ilustra este comportamiento.

Observación sobre el rango de predicciones (RNF8) Los scores obtenidos se concentraron entre 54,2 y 67,3 puntos sobre la escala 0–100. Esta concentración ha-



(a) Completo 1: detecciones parciales en varias imágenes.



(b) Completo 2: detecciones consistentes en todas las imágenes.

Figura 5.10: Comportamiento heterogéneo del detector ante variaciones de captura. Cada cuadro muestra el *bounding box* seleccionado junto con la confianza y el score predicho.

cia la media es consistente con la limitación del modelo ya identificada previamente: el regresor tiende a comprimir sus predicciones hacia el rango medio del puntaje, comportamiento atribuido al tamaño acotado del *dataset* y al desbalance hacia presentaciones de calidad media-alta. La validación de la capacidad discriminativa del modelo entre platos de calidad sustancialmente distinta requeriría un *dataset* más amplio y balanceado.

5.6. Verificación de latencia en dispositivo físico

Como parte de la verificación del módulo deployado, se instrumentó el código de inferencia (`vision_service.dart`) con marcas de tiempo en cada etapa del pipeline para medir la latencia real en dispositivo físico. La medición se realizó sobre 12 ejecuciones en un dispositivo Samsung Galaxy A52s 5G (SM-A528B, Android) en modo *profile*. Los resultados se resumen en la Tabla 5.6.

Cuadro 5.6: Latencia por etapa del pipeline de inferencia (en milisegundos, $n = 12$, dispositivo Samsung Galaxy A52s 5G).

Etapa	Media	Mediana	Mín	Máx	σ
Carga y decodificación	85,9	69,5	62	153	29,5
Preprocesamiento YOLO	17,6	6,0	6	63	18,9
Inferencia YOLO	353,6	354,5	343	362	6,5
Selección bbox y crop	10,7	10,5	7	15	2,2
Preprocesamiento EfficientNet	7,6	7,0	6	13	2,0
Inferencia EfficientNet (TTA 8×)	1507,5	1496,5	1461	1575	38,3
Total pipeline	1986,6	1954,5	1921	2136	74,0

El tiempo total medio fue de 1.987 ms (aproximadamente 2 segundos), con una mediana de 1.955 ms y desviación estándar de 74 ms. La etapa de mayor costo es la inferencia de EfficientNet-Lite0 con *Test Time Augmentation* (TTA) de 8 pasadas, que representa el 75,9 % del tiempo total. Sin TTA, el tiempo estimado por pasada individual de EfficientNet sería de aproximadamente 188 ms, reduciendo el total del pipeline a ≈ 664 ms.

La inferencia YOLO es la etapa más estable ($\sigma = 6,5$ ms), lo que confirma un comportamiento predecible del detector independiente de la imagen procesada. Las etapas de preprocesamiento combinadas contribuyen con solo 25 ms en promedio, confirmando que el cuello de botella está exclusivamente en la inferencia del regresor con TTA.

El tiempo total de ≈ 2 segundos es aceptable para el flujo de uso previsto en SHE-FU, donde la evaluación ocurre una sola vez al finalizar la preparación del plato. La

reducción del número de pasadas TTA a 2–3 permitiría llevar el tiempo total por debajo de 1 segundo, lo que queda propuesto como optimización de trabajo futuro.

En relación con el requerimiento RNF1, el tiempo total obtenido es compatible con una experiencia de usuario fluida en el contexto de uso de SHEFU, donde la evaluación del plato ocurre una única vez al finalizar la sesión de cocina y no requiere respuesta en tiempo real. El pipeline completo se ejecuta en menos de 2,2 segundos en el peor caso observado, lo que resulta aceptable para este flujo de uso.

5.6.1. Consumo de recursos durante la inferencia

Adicionalmente a la medición de latencia, se registró el consumo de CPU y memoria RAM del proceso durante la ejecución del pipeline mediante `adb shell top`, capturando el comportamiento del proceso en intervalos de un segundo. Los resultados se resumen en la Tabla 5.7.

Cuadro 5.7: Consumo de recursos del proceso durante la inferencia (dispositivo Samsung Galaxy A52s 5G, modo *debug*), modo que mantiene la instrumentación necesaria para captura de logs pero introduce sobrecarga adicional respecto al modo *release* de producción; los valores reportados constituyen una estimación moderada de la latencia real en *deployment*

Recurso	Baseline	Pico inferencia
CPU (%)	37	156
RAM - RSS (MB)	291	314
RAM (% del total)	5,4	5,8

El uso de CPU alcanzó un pico de 156 % durante la inferencia, valor que en Android indica activación simultánea de aproximadamente 1,6 núcleos del procesador. Este comportamiento es coherente con el uso del *backend* XNNPACK de TFLite, que paraleliza la ejecución de las 8 pasadas TTA del regresor sobre múltiples núcleos disponibles en el dispositivo. En reposo la app mantiene un consumo de ≈ 37 % de CPU, atribuible al *rendering* de la interfaz y al *thread* principal de Flutter.

El consumo de memoria RAM alcanzó un pico de 314 MB durante la inferencia,

partiendo de una línea base de ≈ 291 MB. El incremento de ≈ 23 MB es atribuible a los buffers de los modelos TFLite cargados en memoria y a los tensores intermedios del pipeline (entradas, salidas, *crops*). El uso total de memoria representa únicamente el 5,8 % del total disponible en el dispositivo de prueba, valor acotado y compatible con dispositivos Android de gama media.

Cabe señalar que las mediciones se realizaron en modo *profile*, que mantiene la optimización AOT del modo *release* pero introduce instrumentación mínima para captura de métricas. Los valores de consumo en deployment *release* pueden ser ligeramente inferiores.

En relación con el requerimiento RNF2, el incremento de memoria RAM durante la inferencia fue de ≈ 23 MB sobre una línea base de 291 MB, representando el 5,8 % del total disponible en el dispositivo de prueba. Este resultado confirma que el módulo es viable en dispositivos Android de gama media sin comprometer la estabilidad del sistema.

5.7. Discusión de resultados y contraste con objetivos

Los resultados obtenidos permiten evaluar el cumplimiento de los objetivos específicos definidos en el Capítulo 1.

El **primer objetivo "Definir criterios visuales y un esquema de evaluación"** se cumplió mediante la construcción de una rúbrica operacional elaborada con el chef del proyecto y aplicada por tres evaluadores identificados sobre el *dataset* de completos. El esquema consolidado operacionaliza atributos de tostado/apariencia, orden, presencia de ingredientes y consistencia visual en un puntaje continuo 0–100.

El **segundo objetivo recolección y curación de datos** se cumplió con la recolección, curación y etiquetado de 197 imágenes de completos bajo condiciones reales de captura, con variabilidad en iluminación, ángulo, distancia y fondo. El *dataset* fue particionado de forma fija y anotado por tres evaluadores bajo el esquema descrito.

El **tercer objetivo implementar un *pipeline* de entrenamiento basado en trans-**

ferencia de aprendizaje con métricas reproducibles" se cumplió para ambos modelos del *pipeline*. El entrenamiento del regresor con EfficientNet-Lite0 sobre PyTorch es reproducible mediante semilla fija ($SEED=42$) y *checkpoint* documentado. Se evaluaron y compararon dos versiones del modelo, seleccionando la más adecuada para el objetivo pedagógico del sistema.

El **cuarto objetivo "validar el módulo mediante evaluación cuantitativa y análisis cualitativo"** se cumplió. Se reportan métricas cuantitativas (MAE, RMSE) para ambos componentes del *pipeline*, se comparan dos versiones del regresor sobre el mismo conjunto de *test*, y se presentan análisis visuales (scatter plot, histograma, ejemplos de predicción) que caracterizan el comportamiento del modelo e identifican las condiciones de falla principales.

El **quinto objetivo "preparar el modelo para despliegue móvil"** se cumplió íntegramente. Ambos modelos fueron exportados exitosamente a TFLite, verificados mediante evaluación con `tf.lite.Interpreter`, e integrados en la aplicación Flutter como artefactos locales `.tflite`. El modelo deployado (V2 float16, 6,74 MB) es consistente con el *checkpoint* PyTorch en términos de métricas, confirmando que la cuantización no introdujo degradación.

Respecto a los requerimientos no funcionales definidos en la Sección 4.2.1, la validación experimental verificó: cumplimiento de RNF1 (latencia interactiva acotada en 2s para evaluación única), RNF2 (consumo acotado de recursos: 5,8 de gama media), RNF3 y RNF4 (despliegue TFLite on-device exitoso), RNF8 (estabilidad del puntaje, $\sigma = 2,96$ puntos). RNF5, RNF6 y RNF7 fueron verificados parcialmente a través de la evaluación de robustez (Sección 5.5.1).

Desde la perspectiva del sistema completo, el módulo VC fue verificado de extremo a extremo dentro de la aplicación Flutter: el puntaje generado es consumido por el módulo de Cálculo de Score, donde se combina con el resultado del Quiz Final y el puntaje base de la receta según la fórmula $\text{Score final} = 0,15 \cdot VC + 0,25 \cdot QF + 0,60 \cdot RB$, alimentando la determinación de nivel y los mecanismos de gamificación del usuario. Este flujo completo fue validado funcionalmente en el prototipo integrado del proyecto.

5.8. Limitaciones

Las principales limitaciones del módulo de Visión por Computadora en su versión prototipo son las siguientes.

Tamaño del *dataset* y desbalance. El *dataset* de 197 imágenes es acotado para una tarea de regresión visual con alta variabilidad de captura. El desbalance hacia scores altos limita la capacidad del modelo de discriminar en el rango bajo (0–40), que es precisamente el rango más relevante pedagógicamente para identificar áreas de mejora. Este desbalance se manifiesta en los casos de mayor error observados en el análisis cualitativo.

Conjunto de *test* reducido. Con 20 muestras de *test*, los valores de MAE y RMSE reportados tienen alta varianza. La comparación entre V1 y V2 es indicativa pero no estadísticamente concluyente. Un conjunto de evaluación de al menos 50–100 muestras permitiría extraer conclusiones más robustas.

Ausencia de métrica de acuerdo inter-evaluador. El *dataset* fue anotado por tres evaluadores siguiendo una rúbrica común, pero no se reporta formalmente una métrica de consistencia entre ellos (por ejemplo, correlación de Pearson o coeficiente de correlación intraclase). La variabilidad entre evaluadores representa una fuente de ruido en las etiquetas que afecta el techo de desempeño del modelo.

Interpretabilidad limitada. Se exploró un enfoque basado en predicción de atributos visuales intermedios (flags binarios) para proveer retroalimentación explicable al usuario. Sin embargo, el *dataset* acotado y el desbalance entre clases impidieron aprender predicciones útiles. Este enfoque queda como línea de trabajo futuro prioritaria, condicionada a la disponibilidad de un *dataset* más amplio y balanceado.

Dominio restringido a completos. El módulo fue diseñado y validado exclusivamente para el plato *completo* chileno. La generalización a otras recetas de SHEFU requeriría nuevos *datasets* y reentrenamiento del regresor, aunque el detector podría adaptarse con anotaciones adicionales.

Cuantización INT8 no evaluada. El regresor se desplegó con cuantización de pe-

los a float16, mientras que el detector se mantuvo en precisión float32. La evaluación del impacto de cuantización INT8 en ambos modelos queda pendiente como trabajo futuro, con potencial de reducir el tamaño de los artefactos y mejorar la latencia de inferencia.

Comportamiento heterogéneo del detector. La evaluación de robustez (Sección 5.5.1) reveló que el detector puede generar *bounding boxes* parciales en ciertos completos, lo que degrada la entrada de la etapa de regresión. La caracterización completa de las condiciones que producen este comportamiento requeriría una evaluación con un conjunto más amplio de imágenes y queda propuesta como trabajo futuro.

Latencia del pipeline con TTA. La evaluación en dispositivo físico reportó una latencia total de aproximadamente 2 segundos por inferencia, atribuible principalmente al uso de *Test Time Augmentation* con 8 pasadas de EfficientNet-Lite0. Si bien este tiempo es aceptable para el flujo de uso previsto en SHEFU, limita la viabilidad del sistema para escenarios de evaluación en tiempo real. La reducción del número de pasadas TTA o su eliminación queda propuesta como optimización de trabajo futuro.

Sensibilidad a la nitidez de la imagen de entrada. Durante la evaluación en dispositivo físico se observó que imágenes capturadas con alta nitidez (cámara completamente estática) tendieron a producir scores más bajos que imágenes con movimiento leve o *blur* natural de captura. Esta observación sugiere que el modelo fue entrenado con imágenes de nitidez variable, condicionando sus predicciones a esa distribución. Imágenes con frecuencias altas no representadas en el *dataset* de entrenamiento pueden producir predicciones menos confiables. Se propone como trabajo futuro incorporar al *dataset* imágenes capturadas en condiciones controladas de alta nitidez, o aplicar *augmentation* de *blur* durante el entrenamiento para hacer el modelo más robusto a esta variación.

Capítulo 6

Conclusiones

En este trabajo se abordó el desafío de diseñar, implementar y validar un módulo de Visión por Computadora para la aplicación móvil SHEFU. El objetivo central fue evaluar visualmente el plato final del usuario para entregarle un puntaje cuantitativo, cerrando así el ciclo de retroalimentación pedagógica de la plataforma. Para lograrlo bajo las estrictas restricciones de un entorno móvil, se construyó una solución basada en un *pipeline* de dos etapas (detección y regresión) que se ejecuta íntegramente *on-device*.

6.1. Sobre el cumplimiento de los objetivos

Al revisar el trabajo realizado, es posible afirmar que los cinco objetivos específicos trazados al inicio fueron cumplidos con éxito. No solo se logró levantar un *dataset* propio de 197 imágenes, sino que estas fueron anotadas de manera trazable gracias a una rúbrica construida en conjunto con el chef del proyecto. Además, se estableció un proceso de entrenamiento reproducible basado en transferencia de aprendizaje. Esto permitió comparar objetivamente distintas arquitecturas (MobileNet V3 Small vs EfficientNet-Lite0) y versiones del modelo para fundamentar las decisiones de diseño. Finalmente, el ciclo se cerró exportando los modelos a TensorFlow Lite e integrándolos en la aplicación Flutter, comprobando empíricamente su funcionamiento en un

dispositivo físico real.

6.2. Sobre el desempeño obtenido

A nivel de resultados, el comportamiento de las etapas del *pipeline* fue contrastante. Por un lado, el detector YOLOv8n superó las expectativas con un mAP50 de 0,97 y un Recall perfecto (1,0) sobre el conjunto de *test* ($n = 20$ imágenes, muestra pequeña que invita a cautela antes de extrapolar a producción). Este nivel de precisión demostró ser más que suficiente para cumplir su rol principal: encontrar el plato y aislarlo del ruido de fondo.

Por otro lado, el regresor EfficientNet-Lite0 obtuvo resultados más modestos, registrando un Error Absoluto Medio (MAE) de 19,77 puntos y un RMSE de 23,46 en la escala de 0 a 100. Es fundamental interpretar estas cifras con madurez y honestidad. Un error promedio de 20 puntos es considerable e indica que al modelo aún le cuesta hilar fino entre preparaciones de calidad similar. Sin embargo, esto no representa un fallo del diseño, sino que refleja la verdadera complejidad intrínseca del problema abordado.

Evaluar la estética de un plato con una sola fotografía es una tarea altamente subjetiva, donde incluso dos evaluadores humanos rara vez están completamente de acuerdo. Si a esta dificultad se suma que el modelo fue entrenado con apenas 197 imágenes, lo cual representa un volumen muy acotado para aprender toda la variabilidad de fondos, luces y ángulos, el margen de error resulta esperable. A pesar de esto, el sistema logra ser estable frente a cambios menores al momento de capturar la foto ($\sigma = 2,96$), responde en un tiempo interactivo razonable (≈ 2 segundos) y apenas suma ≈ 23 MB al consumo de memoria RAM. En definitiva, cumple su propósito como un primer componente operativo y funcional para SHEFU.

6.3. Sobre las decisiones de ingeniería

Visto en retrospectiva, separar la tarea en dos fases (“detectar primero, evaluar después”) fue una de las decisiones estratégicas más acertadas del proyecto. Si se hubiera intentado que un solo modelo hiciera todo el trabajo (*end-to-end*), el ruido del entorno habría arruinado las predicciones dado el volumen de datos disponible. Esta separación también facilitó enormemente diagnosticar dónde fallaba el sistema al momento de hacer pruebas visuales.

Asimismo, la cadena de exportación diseñada (PyTorch → ONNX → SavedModel → TFLite) demostró ser muy sólida. Al aplicar cuantización a `float16`, se logró comprimir el modelo a solo 6,74 MB sin sacrificar su rendimiento predictivo frente a la versión original. Otro acierto metodológico fue instrumentar el código para medir los recursos directamente en el celular físico; gracias a esto se descubrió que el 75 % del tiempo de inferencia se lo lleva el *Test Time Augmentation* (TTA), obteniendo un mapa claro de qué optimizar a futuro.

6.4. Trabajo futuro

Dado que este módulo sienta las bases operativas del sistema, se proponen las siguientes líneas de trabajo prioritarias para futuras iteraciones:

- **Ampliación y balanceo del *dataset*:** Es urgente superar la barrera de las 600 a 800 imágenes. Específicamente, es necesario buscar más ejemplos de platos mal preparados (puntajes menores a 40), ya que actualmente están subrepresentados y es precisamente allí donde el modelo más se equivoca.
- **Métricas de acuerdo inter-evaluador:** Sería muy valioso calcular formalmente el coeficiente de correlación (como ICC o Pearson) entre los distintos humanos que etiquetan. Esto permitiría establecer cuál es el verdadero “techo” de precisión humana con el que compete el modelo.

- **Interpretabilidad del puntaje:** Con un *dataset* más grande, se debería retomar el intento de predecir atributos específicos (como “falta ingrediente” o “presencia desordenada”). Para el usuario de SHEFU, es mucho más enriquecedor recibir un consejo accionable que simplemente ver un número frío.
- **Optimización de latencia:** Bajar las pasadas de TTA de 8 a 2 o 3 reduciría el tiempo de espera a menos de 1 segundo. También valdría la pena probar una cuantización más agresiva (INT8) para bajar el peso del modelo a la mitad, previa calibración.
- **Nuevas recetas y robustez:** Expandir el alcance visual del detector y comenzar a recolectar datos para otras recetas del catálogo es un paso vital para sacar a SHEFU de su fase de prototipo y llevarlo a un producto a escala.

6.5. Reflexión final

Más allá de las métricas puras, el mayor logro de esta memoria es haber entregado un sistema de software real que cierra el ciclo de aprendizaje dentro de la aplicación. A lo largo de este desarrollo quedó en evidencia que problemas que suenan sencillos en el papel, como “ponerle nota a un plato”, ocultan una enorme complejidad técnica cuando se enfrentan a las restricciones de memoria, privacidad y datos ruidosos del mundo móvil.

El valor de este trabajo no radica en haber presentado una solución mágica e infalible, sino en la transparencia con la que se ha delimitado qué puede hacer el sistema con confianza hoy, y qué le falta. Las contribuciones técnicas concretas, tales como un *pipeline* de dos etapas funcionando estable *on-device*, una cadena de exportación verificada y una validación honesta en hardware real, conforman un cimiento sólido. Hoy, el módulo de Visión por Computadora es una base viva y funcional, lista para crecer de forma incremental a medida que la plataforma suma más usuarios y más datos. Es importante destacar que este valor se concreta dentro de un sistema más amplio: el módulo

VC opera como un componente integrado en la plataforma multidisciplinaria SHEFU, cuyo funcionamiento completo —desde la guía de cocina hasta la gamificación— fue desarrollado en conjunto por el equipo del proyecto. El prototipo resultante constituye una prueba de concepto funcional de extremo a extremo, donde cada módulo, incluido el de Visión por Computadora, cumple su rol dentro de un ciclo de aprendizaje coherente y verificable.

Anexos

.1. Scripts de entrenamiento

.1.1. Entrenamiento del detector YOLOv8n

El detector se entrenó utilizando la librería Ultralytics con el siguiente código, extraído del notebook `YOLOv8_Detector_completos_ordenado.ipynb`:

Listing 1: Configuración e invocación del entrenamiento del detector YOLO.

```
IMGSZ          = 320
EPOCHS         = 100
BATCH          = 8
PROJECT        = "detector_completos"
RUN_NAME       = "rf_completos"
CLOSE_MOSAIC   = 10
PATIENCE       = 10

DEVICE = 0 if torch.cuda.is_available() else 'cpu'

BASE_WEIGHTS = "yolov8n.pt"
model = YOLO(BASE_WEIGHTS)

results = model.train(
    data=DATA_YAML,
```

```

    imgsz=IMGSZ,
    epochs=EPOCHS,
    batch=BATCH,
    device=DEVICE,
    project=PROJECT,
    name=RUN_NAME,
    close_mosaic=CLOSE_MOSAIC,
    patience=PATIENCE,
    plots=False
)

```

.1.2. Entrenamiento del regresor EfficientNet-Lite0

El regresor se implementó en PyTorch con la librería `timm`. El código a continuación fue extraído del notebook `EfficientNetLite0_regresion_reentrenado.ipynb` y corresponde al modelo V2 deployado en la aplicación.

Definición del modelo

Listing 2: Carga del backbone y reemplazo de la cabeza de regresión.

```

from timm import create_model
import torch.nn as nn

model = create_model("efficientnet_lite0", pretrained=True)
in_f = model.classifier.in_features
model.classifier = nn.Sequential(
    nn.Linear(in_f, 1),
    nn.Sigmoid()
)
model = model.to(device)

```

Optimizador, loss y scheduler

Listing 3: Configuración del optimizador y criterios de parada.

```
EPOCHS = 100
PATIENCE = 10

criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LR,
                              weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=3
)

def mae(y_pred, y_true):
    """MAE en rango [0, 1]."""
    return torch.mean(torch.abs(y_pred - y_true))

best_val = float("inf")
best_state = None
history = defaultdict(list)
no_improve = 0
```

Loop de entrenamiento

Listing 4: Loop de entrenamiento con freeze/unfreeze y early stopping.

```
# Freeze backbone
for param in model.blocks.parameters():
    param.requires_grad = False

for epoch in range(1, EPOCHS + 1):
```

```

t0 = time.time()

# Descongelar a partir de la epoca 6
if epoch == 6:
    for param in model.blocks.parameters():
        param.requires_grad = True

# ---- Train ----
model.train()
tr_loss, tr_mae = 0.0, 0.0
for x, y in tqdm(train_loader, leave=False):
    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()
    pred = model(x)
    loss = criterion(pred, y)
    loss.backward()
    optimizer.step()
    tr_loss += loss.item() * x.size(0)
    tr_mae += mae(pred, y).item() * x.size(0)

tr_loss /= len(train_ds)
tr_mae /= len(train_ds)

# ---- Validation ----
model.eval()
vl_loss, vl_mae = 0.0, 0.0
with torch.no_grad():
    for x, y in val_loader:
        x, y = x.to(device), y.to(device)
        pred = model(x)

```

```

        loss = criterion(pred, y)
        vl_loss += loss.item() * x.size(0)
        vl_mae += mae(pred, y).item() * x.size(0)

    vl_loss /= len(val_ds)
    vl_mae /= len(val_ds)

    scheduler.step(vl_loss)

    history["train_loss"].append(tr_loss)
    history["val_loss"].append(vl_loss)
    history["train_mae"].append(tr_mae)
    history["val_mae"].append(vl_mae)

    # ---- Early stopping ----
    monitor = vl_loss
    improved = monitor < best_val - 1e-6
    if improved:
        best_val = monitor
        best_state = model.state_dict()
        torch.save(best_state, str(PT_BEST_PATH))
        no_improve = 0
    else:
        no_improve += 1

    if no_improve >= PATIENCE:
        print(f"EarlyStopping: {PATIENCE} epochs sin
              mejora.")
        break

```

```
# Restaurar mejor modelo
if best_state is not None:
    model.load_state_dict(best_state)
```

.2. Pipeline de exportación

.2.1. Exportación PyTorch a ONNX

Listing 5: Exportación del regresor a formato ONNX.

```
model.eval()
dummy = torch.randn(1, 3, IMG_SIZE, IMG_SIZE, device=device)

torch.onnx.export(
    model, dummy, str(ONNX_PATH),
    input_names=["input"],
    output_names=["score01"],
    dynamic_axes={
        "input": {0: "batch"},
        "score01": {0: "batch"}
    },
    opset_version=17
)
```

Donde ONNX_PATH corresponde a runs/effnet_lite0_regresion_reentrenado/model_reentrenado.onnx e IMG_SIZE = 240.

.2.2. Verificación del artefacto TFLite

El script verify_tflite_f16.py verifica la integridad del artefacto TFLite exportado, comprobando shapes, dtypes, ausencia de valores inválidos y rango de sali-

da.

Listing 6: Verificación del artefacto TFLite float16.

```
import os
os.environ["TF_USE_LEGACY_KERAS"] = "1"
import numpy as np
import tensorflow as tf
from pathlib import Path

TFLITE_PATH = Path(__file__).parent / \
    "runs/effnet_lite0_regresion_reentrenado/" \
    "efficientnet_lite0_v2_f16.tflite"

with open(TFLITE_PATH, "rb") as f:
    model_content = f.read()

interp = tf.lite.Interpreter(model_content=model_content)
interp.allocate_tensors()

inp = interp.get_input_details()[0]
out = interp.get_output_details()[0]

# Shapes y dtypes
print(f"Input -> shape={inp['shape']} "
      f"dtype={inp['dtype']}")
print(f"Output -> shape={out['shape']} "
      f"dtype={out['dtype']}")

# Conteo de dtypes internos
tensors = interp.get_tensor_details()
dtype_counts = {}
```

```

for t in tensors:
    k = str(t["dtype"])
    dtype_counts[k] = dtype_counts.get(k, 0) + 1
print(f"Dtypes internos: {dtype_counts}")

# Inferencia de prueba
dummy = np.random.rand(1, 3, 240, 240).astype(np.float32)
interp.set_tensor(inp["index"], dummy)
interp.invoke()
output = interp.get_tensor(out["index"])

# Checks
has_nan = bool(np.any(np.isnan(output)))
has_inf = bool(np.any(np.isinf(output)))
score = float(output[0][0])
in_range = 0.0 <= score <= 1.0

print(f"NaN: {has_nan}   Inf: {has_inf}   "
      f"Rango [0,1]: {in_range}")

```

La salida obtenida al ejecutar este script fue:

```

Tamaño   : 6.7399 MB
Input  -> shape=[ 1   3 240 240]   dtype=float32
Output -> shape=[1 1]               dtype=float32
Dtypes internos: {float32: 204, int32: 8, float16: 100}
NaN: False   Inf: False   Rango [0,1]: True

```

Los 100 tensores internos en float16 confirman que la cuantización se aplicó correctamente sobre los pesos, mientras que la entrada y salida permanecen en float32 como requiere el runtime TFLite.

.3. Scripts de evaluación

.3.1. Comparativa V1 vs V2

El script `eval_both_models.py` evalúa ambas versiones del regresor sobre el mismo conjunto de *test*, garantizando comparabilidad al utilizar idéntico preprocesamiento y particionado.

Listing 7: Script de comparación V1 vs V2 sobre el mismo test set.

```
import os, ast, math
import numpy as np
import pandas as pd
from pathlib import Path
from PIL import Image

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import timm

BASE = Path(__file__).parent
CSV_PATH = BASE / "../completos_evaluacion/" \
               "dataset_regresion_extendido.csv"
CROP_DIR = BASE / "dataset/crops"
MODEL_V1 = BASE /
    "runs/effnet_lite0_regresion/best_model.pth"
MODEL_V2 = BASE /
    "runs/effnet_lite0_regresion_reentrenado/" \
    "best_model_reentrenado.pth"
```

```

IMG_SIZE = 240
BATCH_SIZE = 32
IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD = [0.229, 0.224, 0.225]

val_tfms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=IMAGENET_MEAN,
                          std=IMAGENET_STD),
])

device = torch.device(
    "cuda" if torch.cuda.is_available() else "cpu")

def clean_score(x):
    try:
        if isinstance(x, str):
            data = ast.literal_eval(x)
            if isinstance(data, list) and data \
                and "number" in data[0]:
                return float(data[0]["number"])
            return float(x)
        return float(x)
    except Exception:
        return None

df = pd.read_csv(CSV_PATH)
df["score_clean"] = df["score"].apply(clean_score)
df = df.dropna(subset=["img_path", "score_clean"])

```

```

def build_crop_path(row):
    base = os.path.basename(str(row["img_path"]))
    split = row.get("split", "train")
    return str(BASE / "dataset/crops" /
               f"crop_{split}_{base}")

df["crop_path"] = df.apply(build_crop_path, axis=1)
df["score01"] = (df["score_clean"] / 100.0).clip(0, 1)
test_df = df[df["split"] == "test"].reset_index(drop=True)

class ScoreDataset(Dataset):
    def __init__(self, frame, tfms):
        self.tfms = tfms
        frame = frame.copy()
        frame["exists"] = frame["crop_path"].apply(
            lambda p: Path(p).exists())
        self.frame = frame[frame["exists"]]\
            .reset_index(drop=True)

    def __len__(self):
        return len(self.frame)

    def __getitem__(self, idx):
        row = self.frame.iloc[idx]
        img = Image.open(row["crop_path"]).convert("RGB")
        x = self.tfms(img)
        y = torch.tensor([np.float32(row["score01"])],
                        dtype=torch.float32)

        return x, y

```

```

test_ds = ScoreDataset(test_df, val_tfms)
test_loader = DataLoader(test_ds, batch_size=BATCH_SIZE,
                          shuffle=False, num_workers=0)

def build_model():
    model = timm.create_model("efficientnet_lite0",
                              pretrained=False)

    in_f = model.classifier.in_features
    model.classifier = nn.Sequential(
        nn.Linear(in_f, 1), nn.Sigmoid())
    return model.to(device)

def evaluate(model, loader):
    model.eval()
    preds, reals = [], []
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device)
            p = model(x).cpu().numpy().ravel()
            preds.extend(p.tolist())
            reals.extend(y.numpy().ravel().tolist())
    preds = np.array(preds) * 100
    reals = np.array(reals) * 100
    mae = np.mean(np.abs(preds - reals))
    rmse = math.sqrt(np.mean((preds - reals) ** 2))
    return mae, rmse

# V1
m1 = build_model()

```

```

m1.load_state_dict(
    torch.load(str(MODEL_V1), map_location=device))
mae1, rmse1 = evaluate(m1, test_loader)

# V2
m2 = build_model()
m2.load_state_dict(
    torch.load(str(MODEL_V2), map_location=device))
mae2, rmse2 = evaluate(m2, test_loader)

print(f"V1 | MAE: {mae1:.2f} | RMSE: {rmse1:.2f}")
print(f"V2 | MAE: {mae2:.2f} | RMSE: {rmse2:.2f}")

```

3.2. Evaluación de robustez

El script `evaluar_robustez.py` carga los modelos TFLite deployados, recorre las imágenes organizadas por completo y condición, ejecuta el pipeline completo (detección, recorte y regresión) y reporta los resultados en formato tabular. El script también genera las visualizaciones con *bounding boxes* mostradas en la Figura 5.10. A continuación se presentan las dos funciones principales del pipeline:

Listing 8: Funciones principales del script de evaluación de robustez.

```

YOLO_SIZE = 320
EFFN_SIZE = 240
CONF_THRESH = 0.05
IMAGENET_MEAN = np.array([0.485, 0.456, 0.406],
                           dtype=np.float32)
IMAGENET_STD = np.array([0.229, 0.224, 0.225],
                          dtype=np.float32)

```

```

def detect_and_crop(img_bgr, det_interp):
    """
    YOLO TFLite sobre img_bgr.
    Retorna (crop_bgr, conf, (x1,y1,x2,y2)) o
    (None, 0.0, None) si no detecta.

    Salida YOLO: [1, 5, 2100] -> transpose a [2100, 5]
    -> [cx, cy, w, h, conf] en px del input 320x320.
    """
    H, W = img_bgr.shape[:2]

    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
    img_resized = cv2.resize(img_rgb,
                              (YOLO_SIZE, YOLO_SIZE))
    input_data = (img_resized.astype(np.float32)
                  / 255.0)[np.newaxis, ...]

    det_in = det_interp.get_input_details()[0]
    det_out = det_interp.get_output_details()[0]
    det_interp.set_tensor(det_in["index"], input_data)
    det_interp.invoke()

    raw = det_interp.get_tensor(
        det_out["index"])[0]          # (5, 2100)
    dets = raw.T                      # (2100, 5)

    mask = dets[:, 4] >= CONF_THRESH
    if not mask.any():
        return None, 0.0, None

```

```

dets = dets[mask]
best = dets[np.argmax(dets[:, 4])]
cx, cy, bw, bh, conf = best

scale_x = W / YOLO_SIZE
scale_y = H / YOLO_SIZE
x1 = int(np.clip((cx - bw/2) * scale_x, 0, W-1))
y1 = int(np.clip((cy - bh/2) * scale_y, 0, H-1))
x2 = int(np.clip((cx + bw/2) * scale_x, 0, W-1))
y2 = int(np.clip((cy + bh/2) * scale_y, 0, H-1))

if x2 <= x1 or y2 <= y1:
    return None, 0.0, None

return img_bgr[y1:y2, x1:x2], float(conf), \
    (x1, y1, x2, y2)

def predict_score(crop_bgr, reg_interp):
    """
    EfficientNet-Lite0 TFLite sobre el recorte.
    Retorna score en [0, 100].
    Input: [1, 3, 240, 240] NCHW float32,
    normalizado con ImageNet.
    """
    reg_in = reg_interp.get_input_details()[0]
    reg_out = reg_interp.get_output_details()[0]

    img_rgb = cv2.cvtColor(crop_bgr,

```

```

cv2.COLOR_BGR2RGB)

img_res = cv2.resize(img_rgb,
                      (EFFN_SIZE, EFFN_SIZE)).astype(np.float32) \
                      / 255.0

img_norm = (img_res - IMAGENET_MEAN) / IMAGENET_STD
img_nchw = np.transpose(img_norm, (2, 0, 1)) \
            [np.newaxis, ...]

reg_interp.set_tensor(reg_in["index"],
                      img_nchw.astype(np.float32))

reg_interp.invoke()

score = float(reg_interp.get_tensor(
    reg_out["index"]).flatten()[0])

return round(score * 100.0, 2)

```

.4. Resultados detallados de la evaluación de robustez

Las tablas a continuación presentan los resultados individuales por imagen de la evaluación de robustez reportada en la Sección 5.5.1.

Completo 1

Cuadro 1: Completo 1: variación de ángulo.

Archivo	Score	Conf. YOLO	Estado
IMG_2049	58,39	0,069	OK
IMG_2050	58,19	0,399	OK
IMG_2052	55,65	0,867	OK
IMG_2054	61,94	0,685	OK
IMG_2064	54,23	0,903	OK
Media	57,68	—	—

Cuadro 2: Completo 1: variación de fondo.

Archivo	Score	Conf. YOLO	Estado
IMG_2043	67,30	0,681	OK
IMG_2044	59,07	0,500	OK
IMG_2046	66,86	0,670	OK
IMG_2047	56,51	0,733	OK
IMG_2048	64,92	0,827	OK
Media	62,93	—	—

Completo 2

Cuadro 3: Completo 2: variación de ángulo.

Archivo	Score	Conf. YOLO	Estado
IMG_2065	56,54	0,894	OK
IMG_2066	66,11	0,890	OK
IMG_2067	55,71	0,704	OK
Media	59,45	—	—

Cuadro 4: Completo 2: variación de fondo.

Archivo	Score	Conf. YOLO	Estado
IMG_2065	56,54	0,894	OK
IMG_2074	63,41	0,884	OK
IMG_2075	61,51	0,872	OK
IMG_2076	56,87	0,887	OK
Media	59,58	—	—

Cuadro 5: Completo 2: estabilidad.

Archivo	Score	Conf. YOLO	Estado
IMG_2076	56,87	0,887	OK
IMG_2079	63,35	0,886	OK
IMG_2081	57,31	0,882	OK
Media	59,18	—	—
Desv. estándar	2,96	—	—
Rango	6,48	—	—

.5. Mediciones de consumo de recursos

Las mediciones de consumo de CPU y RAM reportadas en la Sección 5.6.1 se obtuvieron mediante el comando `adb shell top -n 1` ejecutado en intervalos de un segundo durante el procesamiento de imágenes en la aplicación. La Tabla 6 presenta las mediciones individuales utilizadas para el cálculo de las estadísticas reportadas.

Cuadro 6: Mediciones individuales de consumo de recursos durante la inferencia (modo *profile*, dispositivo Samsung Galaxy A52s 5G).

Tiempo	CPU (%)	RSS (MB)	MEM (%)	Fase
00:45:10	37,9	271	5,0	Baseline
00:45:11	37,9	263	4,8	Baseline
00:45:13	16,6	268	4,9	Baseline
00:45:14	44,8	273	5,0	Baseline
00:45:16	134,0	307	5,7	Inferencia
00:45:17	156,0	312	5,8	Inferencia (pico)
00:45:19	41,3	301	5,5	Post- inferencia
00:45:20	20,6	307	5,7	Post- inferencia
00:45:21	31,0	309	5,7	Post- inferencia
00:45:23	48,2	314	5,8	Post- inferencia
00:45:24	37,9	302	5,6	Post- inferencia
00:45:27	55,1	305	5,6	Post- inferencia

Bibliografía

- [1] SideChef. Recipegen ai: Generate recipes from photos. Business Wire, 2024.
- [2] Whirlpool Corporation. Whirlpool and yummlly: Smart cooking platform. Press release, 2020.
- [3] Innit. Innit: Personalized cooking experience platform. Company website, 2018.
- [4] Juho Hamari, Jonna Koivisto, and Harri Sarsa. Does gamification work? – a literature review of empirical studies on gamification. In *2014 47th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2014.
- [5] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.
- [6] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.
- [7] Renjie Liu. Higher accuracy on vision models with EfficientNet-Lite. <https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html>, 2020. TensorFlow Blog.

- [8] Mingxing Tan and Quoc V. Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.
- [9] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics YOLOv8, 2023.
- [10] TensorFlow. Tensorflow lite: On-device machine learning. Documentation, 2024.
- [11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library, 2019.
- [12] Ross Wightman. PyTorch image models. <https://github.com/rwightman/pytorch-image-models>, 2019.
- [13] ONNX Community. ONNX: Open neural network exchange. <https://github.com/onnx/onnx>, 2023.
- [14] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, et al. TensorFlow: Large-scale machine learning on heterogeneous systems. <https://www.tensorflow.org>, 2015.
- [15] Natalie Cheng. The ingredients of design: Diving into the ui of the zest cooking app. Medium, 2024.
- [16] IBM. Monitoring recipe preparation using instructive device. US Patent US8992225B2, 2015.
- [17] Y. Zhang et al. Mirecipe: A dataset for stage-aware ingredient recognition. In *ACM Multimedia Asia*, 2020.
- [18] Lin et al. Automatic judgment of food color and cooking conditions using ai. *Processes*, 2021.

- [19] R. Qiao et al. Food image quality evaluation using deep learning. *IEEE Access*, 2020.
- [20] Roboflow. Roboflow: Computer vision tools for developers. <https://roboflow.com>, 2023.
- [21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.
- [22] Katsuya Hyodo. onnx2tf: Convert ONNX to TensorFlow / TFLite / EdgeTPU / TensorRT. <https://github.com/PINTO0309/onnx2tf>, 2023.