



UNIVERSIDAD DE BURGOS  
ESCUELA POLITÉCNICA SUPERIOR  
Grado en Ingeniería Informática



TFG del Grado en Ingeniería  
Informática

Detección de defectos en  
piezas metálicas usando  
radiografías y *Deep Learning*  
-  
Documentación Técnica



Presentado por Fco. Javier Yagüe Izquierdo  
en Universidad de Burgos — 9 de junio de  
2021

Tutor: José Francisco Díez Pastor y Pedro  
Latorre Carmona



---

# Índice general

---

<b>Índice general</b>	<b>I</b>
<b>Índice de figuras</b>	<b>III</b>
<b>Índice de tablas</b>	<b>v</b>
<b>Apéndice A Plan de Proyecto Software</b>	<b>1</b>
A.1. Introducción . . . . .	1
A.2. Planificación temporal . . . . .	1
A.3. Estudio de viabilidad . . . . .	6
<b>Apéndice B Especificación de Requisitos</b>	<b>11</b>
B.1. Introducción . . . . .	11
B.2. Objetivos generales . . . . .	11
B.3. Catalogo de requisitos . . . . .	11
B.4. Especificación de requisitos . . . . .	13
<b>Apéndice C Especificación de diseño</b>	<b>19</b>
C.1. Introducción . . . . .	19
C.2. Diseño de datos . . . . .	19
C.3. Diseño procedimental . . . . .	22
C.4. Diseño arquitectónico . . . . .	23
<b>Apéndice D Documentación técnica de programación</b>	<b>25</b>
D.1. Introducción . . . . .	25
D.2. Estructura de directorios . . . . .	25
D.3. Manual del programador . . . . .	27

D.4. Compilación, instalación y ejecución del proyecto . . . . .	29
D.5. Pruebas del sistema . . . . .	34
<b>Apéndice E Documentación de usuario</b>	<b>37</b>
E.1. Introducción . . . . .	37
E.2. Requisitos de usuarios . . . . .	37
E.3. Instalación . . . . .	38
E.4. Manual del usuario . . . . .	45
<b>Bibliografía</b>	<b>57</b>

---

# Índice de figuras

---

B.1. Diagrama de casos de uso . . . . .	13
C.1. Clase Detector . . . . .	20
C.2. Diagrama de secuencia de la aplicación . . . . .	22
D.1. Estructura del Proyecto . . . . .	26
D.2. Integración de Google Drive . . . . .	30
D.3. Recorrido de directorios . . . . .	30
D.4. Instalación de dependencias . . . . .	30
D.5. Instalación de Detectron2 . . . . .	31
D.6. Importación de bibliotecas básicas . . . . .	31
D.7. Importación de bibliotecas básicas . . . . .	32
D.8. Importación de bibliotecas básicas . . . . .	32
E.1. Icono de ejecución Docker . . . . .	39
E.2. Directorio tras clonación o copia . . . . .	39
E.3. Opción de apertura directa PowerShell . . . . .	40
E.4. Comando build de imagen . . . . .	40
E.5. Descarga de modelo . . . . .	41
E.6. Sección Images en Docker . . . . .	41
E.7. Creación de contenedor desde imagen . . . . .	41
E.8. Configuración del contenedor . . . . .	42
E.9. Contenedor correctamente creado . . . . .	42
E.10. Directorio correcto . . . . .	43
E.11. Limpieza de la cache de instalación . . . . .	44
E.12. Lanzamiento de la aplicación . . . . .	46
E.13. Página de inicio . . . . .	46
E.14. Navbar superior . . . . .	46

E.15.Fichero requerido . . . . .	47
E.16.Formatos requeridos . . . . .	47
E.17.Previsualización de imagen a detectar . . . . .	48
E.18.Loader de detección . . . . .	48
E.19.Muestra de resultados . . . . .	49
E.20.Muestra de métricas . . . . .	49
E.21.Controles Plotly . . . . .	50
E.22.Función de zoom . . . . .	50
E.23.Ocultando detecciones . . . . .	51
E.24.Botones de descarga . . . . .	51
E.25.Máscara binaria descargada . . . . .	52
E.26.Composición descargada . . . . .	52
E.27.Ejecución sin detecciones . . . . .	53
E.28.Gráfico de histórico . . . . .	54
E.29.Actualización del modelo . . . . .	55
E.30.Loader de actualización . . . . .	55
E.31.Actualización correcta . . . . .	55
E.32.Modelo local actualizado . . . . .	56

---

# Índice de tablas

---

A.1. Costes de personal . . . . .	7
A.2. Costes Finales . . . . .	8
A.3. Bibliotecas de <i>Python</i> utilizadas y sus licencias . . . . .	9
B.1. C0: Cargar imagen a la aplicación . . . . .	14
B.2. C1: Ejecutar detección . . . . .	15
B.3. C2: Obtener información de los defectos . . . . .	16
B.4. C4: Consultar el histórico . . . . .	17
D.1. Pruebas de la aplicación . . . . .	34
D.2. Pruebas de la aplicación . . . . .	35
D.3. Pruebas de la aplicación . . . . .	36





## Apéndice A

---

# Plan de Proyecto Software

---

### A.1. Introducción

En este apartado se recoge la planificación temporal que se ha seguido durante el desarrollo del proyecto, analizando los pasos seguidos en cada una de las fases y el estudio de la viabilidad tanto económica como legal que tendría el desarrollo del proyecto.

### A.2. Planificación temporal

En este proyecto se ha aplicado la metodología *Scrum*, por lo que se han ido estableciendo unos objetivos conforme se avanzaba el desarrollo y se han ido dividiendo en *sprints*. Estos normalmente compuestos de *issues* o puntos relevantes a abordar hasta el siguiente *sprint*.

En determinadas ocasiones y en caso de que las tareas llevasen más o menos tiempo, era posible adelantar o retrasar las reuniones semanales para optimizar el tiempo. El objetivo no era hacer muchas tareas simultáneamente sino tener siempre ciertas tareas en proceso para que el desarrollo fuese constante.

Para la gestión del proyecto se utilizó la plataforma de desarrollo colaborativo *Github*. El repositorio puede encontrarse en <https://github.com/fyi0000/TFG-GII-20.04>.

Paralelamente y aunque se hizo de forma personal y se ha ido modificando, también se llevó un diario de tareas posibles o conceptos en la plataforma *Trello*. Esta plataforma permite la organización temporal según se personalice

en diferentes tarjetas o *boxes* que pueden eliminarse, editarse o asignarse a otras tareas según se considere.

## Sprint 0: Introducción y revisión

Durante este primer *sprint* se concertó una reunión y se estableció un plan de fechas en *Microsoft Teams* para las posteriores semanas. A su vez se introdujo el trabajo anterior de la compañera *Noelia Ubierna Fernández* realizado el año pasado.

Se comentaron las mejoras y principales diferencias respecto al anterior. Marcando como objetivo que el funcionamiento se basase en un conjunto de imágenes propias en lugar de un repositorio de terceros [2].

Por ello primero se establecieron los puntos:

- Inicializar el repositorio y la toma de documentación
- Descarga de imágenes propias
- Descarga y ejecución con las imágenes objetivo actuales en la herramienta del año pasado

## Sprint 1: Conclusiones y decisiones

Tras las pruebas iniciales en las imágenes que se marcaron como objetivo se observó que el comportamiento no era el más adecuado, si bien esto se esperaba al estar la red neuronal entrenada sobre un conjunto de imágenes distinto al de las imágenes segmentadas por los propios tutores.

Se contemplaron alternativas y modelos parecidos a *Mask R-CNN* que utilizaba la antigua herramienta.

La principal tarea de este *sprint* se estableció en estudiar la viabilidad de utilizar la herramienta *Detectron2*.

## Sprint 2: Registro de imágenes y pruebas

Finalmente se decidió que el proyecto comenzaría desde 0 utilizando *Detectron2*.

El primer paso para su uso era el estudio del formato *COCO* (*Common Objects in Context*) para el registro del conjunto y posterior entrenamiento. Este formato se ha detallado en la memoria del proyecto, consiste en la

generación de un fichero *JSON* que recoge las imágenes y sus correspondientes nombres de fichero asignándoles un identificador, clases presentes en el conjunto y las anotaciones. Este último punto especifica como es la forma de un defecto y que espacio ocupa dentro de la imagen además de relacionar cada uno con las distintas imágenes que forman el conjunto.

Por ello los tutores facilitaron el acceso a la computadora de la Universidad *Alpha*.

Los objetivos fueron:

- Introducirse y documentarse al registro de imágenes en formato *COCO*
- Primeras pruebas con las imágenes del repositorio de *Max Ferguson* [2]

### **Sprint 3: Cambio de entorno y primeros resultados**

Durante el desarrollo del anterior *sprint* se “descubrió” el entorno *Google Colaboratory* y la facilidad de uso del mismo. Si bien ya se conocía la herramienta *Azure* de *Microsoft*.

La documentación oficial de *Detectron2* y numerosos usuarios utilizaban este entorno así que en lugar de realizar el desarrollo y pruebas en la máquina *Alpha* de la Universidad, se realizaría finalmente en *notebooks* alojados en la plataforma de *Google*. Esto facilitaría el uso repetido ya que no es necesario el uso de *VPN* para su acceso.

Los primeros resultados del registro fueron positivos y se consiguió un *notebook* que:

- Recorría las imágenes y estandarizaba los nombres
- Generaba el fichero *JSON* con cada una de las secciones
- Recorría los directorios y emparejaba las máscaras individualizadas de cada defecto con la imagen correspondiente

### **Sprint 4: Primer Entrenamiento**

Durante este *sprint*:

- Se completó el registro de todas las imágenes propias

- Se comprobó la correcta integración del conjunto con *Detectron2*
- Se realizó el primer entrenamiento provisional

Los resultados fueron positivos y se planificó continuar las pruebas de entrenamiento, variando valores y distribución de los conjuntos de entrenamiento y test.

## Sprint 5: Entrenamientos y estudio de gráficas

- Se ajustaron los valores de entrenamiento para mejorar los resultados
- Con la herramienta integrada de *Detectron2* llamada *TensorBoard* se observaron las gráficas de *loss* durante entrenamiento y test. Este parámetro representa la eficiencia del modelo sobre el conjunto de entrenamiento y test respectivamente además de servir como referencia para determinar fenómenos como el sobreentrenamiento.
- Se observaron las primeras anomalías de entrenamiento

En este paso ya se comenzaron a comprobar efectos de sobreentrenamiento en el conjunto.

También se implementaron las métricas *Precision*, *Recall* y F1 para evaluar el rendimiento del modelo sobre las máscaras reales.

## Sprint 6: Refinamiento y despliegue web

Una vez obtenido un modelo eficiente y que cumpliera los mínimos establecidos, se decidió que la ejecución de la herramienta se realizaría mediante el *microframework Flask* que utiliza el lenguaje *Python*.

Es en este punto cuando se observa el desarrollo en *sprints* marcados como hitos en el repositorio [3]

A partir de este punto se empiezan a registrar los avances en el repositorio, ya que se hizo uno de prueba y la multitud de cambios realizados en simplemente dos *notebooks* no representaban demasiados avances.

A su vez, para estudiar el comportamiento del modelo dependiendo de la forma del registro de los modelos, 1 imagen por defecto ó 1 imagen conteniendo todos los defectos, se elaboró otro *notebook* basado en el anterior que dividía nuestras máscaras binarias. El proceso consistía en reconocer

las secciones conexas de la imagen, que presentaban defectos y generar una imagen propia conteniendo dicha región. Además los nombres se generaban de forma que fuese sencillo de relacionar cada imagen original con las múltiples asociadas. No significó diferencia en los resultados.

## Sprint 7: Avances en *Flask*

Los sucesivos avances marcados como los dos primeros *milestones* en el repositorio consistieron en:

- Generar una sencilla web en *Flask* que a su vez pudiese contenerse en una imagen *Docker*
- Probar el funcionamiento de los *Dockerfile*
- Añadir gráficos *Plotly* tanto para la presentación de resultados como para el histórico
- Añadir *slider* que permite establecer el mínimo de *score* o confianza para que se muestre un defecto
- Estudiar el funcionamiento de *Bootstrap*

Se generó también una clase propia que manipulase *Detectron2* y permitiese su interacción con la aplicación web.

## Sprint 8: Finalización y últimos ajustes

En este *sprint* se finalizó la aplicación web y se ajustó *Bootstrap* para la mejora de la apariencia general de la misma. Se comenzaron a depurar errores que surgieron en la aplicación meramente estéticos con la introducción de *Bootstrap*

Se realizaron comprobaciones finales y se comenzó con la documentación al tener un contenido consistente.

### A.3. Estudio de viabilidad

A continuación se estudiará la viabilidad económica del proyecto contemplando los diferentes costes de personal y equipo y la viabilidad legal que engloba el uso de licencias gratuitas o la compra de las mismas.

#### Viabilidad económica

Esta sección se dividirá por un lado el coste relacionado con las personas implicadas y por otro, el equipo junto con software y hardware utilizado.

##### Coste de personal

Para el desarrollo del proyecto se han observado dos fases relevantes,

- Entrenamiento y *Deep Learning*
- Desarrollo Web

Sin embargo, a pesar de ser dos fases casi independientes, la dependencia la una de la otra es tal que es perfectamente viable el desarrollo completo por parte de un único trabajador. No se tiene en cuenta posteriores mejoras como la segmentación de más imágenes para las que podría ser necesario personal más especializado.

##### Salario de personal

Para la consulta del salario de un desarrollador que fuese capaz de hacer ambas tareas se ha consultado la web *PayScale* que analiza perfiles y ofrece estadísticas salariales según puesto, experiencia o rama que se ocupa en un campo. [https://www.payscale.com/research/ES/Job=Web\\_Developer/Salary](https://www.payscale.com/research/ES/Job=Web_Developer/Salary)

A pesar de recoger distintos perfiles, se tendrá en cuenta la rama general de *Web Developer* en España con 1-2 años de experiencia. Esto significa un sueldo al mes de  $18000/12 = 1500\text{€}$ .

Concepto	Valor
Salario anual bruto	18.000
Cotización a la Seguridad Social	-1.143
IRPF 10,89 %	-1.888
<b>Sueldo Neto Anual</b>	<b>14.968</b>

Tabla A.1: Costes de personal

Por ello si el proyecto se alargase unos 6 meses el coste total en personal sería de la mitad del anual, 7.484 €.

### Coste de equipo

En cuanto al equipo se debe remarcar que *Flask* es un *miniframework* que no está especialmente diseñado para la respuesta a múltiples peticiones y suele utilizarse en un ámbito más local, como es el presente caso. Si se quisiese desplegar la aplicación en una escala mayor, se debería contemplar la migración a otros *frameworks* o por ejemplo su alojamiento en *Google Colab* que incluso utilizando la versión Pro, el uso prolongado estaría limitado a 24h.

Un servicio de *hosting* limitado y no demasiado tráfico disponible, podría rondar al año en torno a los 100€, que no se contabilizarán en el total ya que en un uso industrial, el despliegue a nivel local puede ser suficiente según necesidades.

## Hardware

Inicialmente y se se hubiese utilizado la máquina *Alpha* el precio a contemplar sería mucho más alto. Pero como finalmente se ha hecho uso de *Google Colaboratory* y que la versión Pro, de pago, no parece a priori necesaria, se sumarán 500 euros al total. Esto teniendo en cuenta un equipo sencillo que permita un desarrollo web mínimo y acceso a la plataforma de *Google*. Luego:

Concepto	Valor
Salario de personal	7.484
Equipo mínimo	500
<b>Coste Total</b>	<b>7.984</b>

Tabla A.2: Costes Finales

## Viabilidad legal

En cuanto a la viabilidad legal, tanto las bibliotecas de *Python* como el uso de *Google Colab* permiten el uso gratuito y no presentan problemas para el uso comercial.

Un punto importante es el uso de imágenes, normalmente privadas, que tendría que facilitar bajo permiso una empresa interesada para su uso en el proyecto. Sin dichas imágenes solo se podría recurrir a imágenes de dominio público.

Esto podría representar un serio problema ya que se requiere de una cantidad mínima de imágenes con las características necesarias para que el modelo esté balanceado. Además es común que este tipo de imágenes correspondan a diferentes piezas y harían falta diferentes enfoques de la misma para que el entrenamiento fuese fiable.

Es uno de los puntos más importantes ya que depende de él la eficiencia del producto final.



**Bibliotecas utilizadas**

<b>Biblioteca</b>	<b>Licencia</b>
OpenCV	MIT License (MIT)
Requests	Apache Software License (Apache 2.0)
Numpy	BSD License (BSD)
Pandas	BSD License (BSD)
Plotly	MIT License (MIT)
Matplotlib	BSD License (BSD) Compatible
PIL	Historical Permission Notice and Disclaimer (HPND) (HPND)
scikit-image	BSD License (Modified BSD)
Werkzeug	BSD License (Modified BSD)
gdown	MIT License (MIT)
urllib	MIT License (MIT)
wget	Public Domain (Public Domain)

Tabla A.3: Bibliotecas de *Python* utilizadas y sus licencias

Y por último, tanto *Detectron2* como *Docker*, en su versión *Community* tienen licencia *Apache License 2.0*, que permite el uso comercial con los debidos créditos. Repositorio oficial de *Detectron2*: <https://github.com/facebookresearch/detectron2>



## *Apéndice B*

---

# Especificación de Requisitos

---

### B.1. Introducción

A continuación se exponen los objetivos generales del proyecto y del producto final así como los diferentes requisitos y casos de uso.

### B.2. Objetivos generales

- Obtención de un modelo que se comporte correctamente con las imágenes propias
- Integración en una aplicación web que permita la correcta ejecución del modelo de forma sencilla para el usuario
- Mostrar los resultados de una forma amigable y permitir la conservación de los mismos

### B.3. Catalogo de requisitos

Se detallan ahora los requisitos funcionales de la aplicación.

- **RF 1** Permitir la ejecución del modelo sobre una radiografía mostrando las detecciones.
  - **RF 1.1** Poder subir la imagen objetivo del usuario a la aplicación.
  - **RF 1.2** Mostrar previsualización de la imagen.

- **RF 1.3** Muestra de los resultados de forma interactiva.
- **RF 1.4** Personalizar el valor límite o confianza de los defectos.
- **RF 2** Permitir la descarga de resultados.
  - **RF 2.1** Descargar la máscara binaria generada.
  - **RF 2.2** Descargar el resultado interactivo de la figura *Plotly*.
  - **RF 2.3** Descargar los resultados en forma de composición.
- **RF 3** Mostrar información de las detecciones.
  - **RF 3.1** Clasificar los defectos por tamaño y etiquetarlos en la figura.
  - **RF 3.2** Mostrar el área del defecto.
- **RF 4** Generar un histórico de detecciones.
  - **RF 4.1** Registrar en un fichero las detecciones por fecha y tamaño de defectos.
  - **RF 4.2** Generar un gráfico de áreas a partir del histórico.

## B.4. Especificación de requisitos

### Diagrama de casos de uso

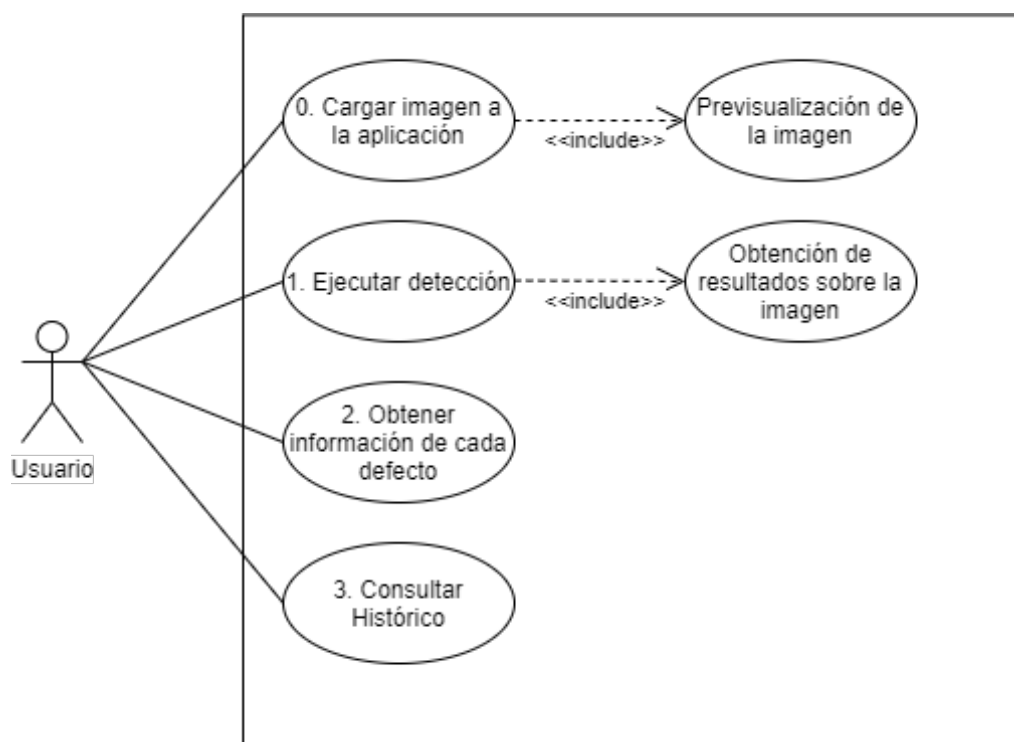


Figura B.1: Diagrama de casos de uso

C0: Cargar imagen a la aplicación.	
Descripción	Permite cargar una imagen a la aplicación para su uso.
Requisitos	RF 1.1
	RF 1.2
Precondiciones	Aplicación inicializada
Secuencia normal	Paso    Acción
	1        Se selecciona una imagen.
	2        Se pulsa el botón subir y se muestra la profesionalización.
	3        La imagen se guarda en el directorio de la aplicación.
Postcondiciones	La imagen se guarda en el directorio correspondiente y se muestra la previsualización.
Excepciones	No se ha seleccionado ningún fichero. El formato de fichero no es ni .png, .jpg o .jpeg

Tabla B.1: C0: Cargar imagen a la aplicación

C1: Ejecutar detección.		
Descripción	Se ejecuta el modelo sobre la imagen y se muestran los resultados.	
Requisitos	RF 1	
	RF 1.4	
Precondiciones	Imagen correctamente cargada en la aplicación.	
Secuencia normal	Paso	Acción
	1	Se selecciona en el <i>slider</i> la confianza deseada para la detección.
	2	Se pulsa el botón Aceptar y se procede a la detección.
	3	Tras la carga, se muestran los resultados en forma de gráfico <i>Plotly</i> .
Postcondiciones	Se genera el gráfico <i>Plotly</i> y se guarda correctamente.	
Excepciones	Hay un fallo en la carga de la imagen.	

Tabla B.2: C1: Ejecutar detección

C2: Obtener información de los defectos.	
Descripción	Gracias al gráfico <i>Plotly</i> el usuario puede colocar el cursor sobre los diferentes defectos y observar la información correspondiente. También puede pulsar sobre la leyenda para mostrar/ocultar cada uno.
Requisitos	RF 1.3
	RF 2
	RF 2.1
	RF 2.2
	RF 2.3
	RF 3.1
	RF 3.2
Precondiciones	Detección ejecutada.
Secuencia normal	Paso    Acción
	1        Mostrando el gráfico, colocar el cursor sobre cada defecto coloreado.
	2        Se muestra el ID del defecto, Área y tipo clasificado por tamaño, ya sea Grande, Pequeño o Mediano.
	3        Se puede pulsar y mantener pulsado para hacer zoom sobre la imagen o defectos.
	4        Se muestran 3 opciones de descarga de resultados: Gráfico interactivo, máscara binaria ó composición original-máscara.
Postcondiciones	El usuario ha podido comprobar la información de cada defecto detectado.
Excepciones	No hay detecciones y no se muestran defectos pero si el gráfico <i>Plotly</i>

Tabla B.3: C2: Obtener información de los defectos



C4: Consultar el histórico.		
Descripción	Permite cargar una imagen a la aplicación para su uso.	
Requisitos	RF 4	
	RF 4.1	
	RF 4.2	
Precondiciones	Aplicación inicializada	
Secuencia normal	Paso	Acción
	1	En la <i>Navbar</i> superior, se selecciona la opción Histórico.
	2	Se muestra el gráfico <i>Plotly</i> y es posible hacer zoom, alejar la vista y colocar el cursor sobre los puntos para obtener la información de cada día. Defectos de un determinado tamaño detectados ese día y número total de imágenes procesadas.
Postcondiciones	El usuario comprueba como es el histórico de detecciones en la aplicación.	
Excepciones	El fichero histórico está ausente o defectuoso.	

Tabla B.4: C4: Consultar el histórico



## *Apéndice C*

---

# Especificación de diseño

---

### C.1. Introducción

En esta sección se expone cómo se manipulan los datos en la aplicación, diseño procedimental y el diseño arquitectónico.

### C.2. Diseño de datos

A continuación se detalla cómo se organizan los datos en la aplicación. Las imágenes que se han usado para el entrenamiento y test, cómo está estructurada la aplicación con sus ficheros y por último cómo se estructura el fichero que proporciona cierta persistencia entre las ejecuciones.

#### Imágenes

Las imágenes utilizadas han sido segmentadas por los tutores de este proyecto. Son un total de 21 imágenes facilitadas en el repositorio. Contienen diferentes piezas metálicas que presentan determinados defectos, a su vez se acompañan de 21 máscaras binarias correspondientes a cada una de las anteriores y que presentan una imagen de mismas dimensiones que la original. El contenido es una imagen en 2 colores (binaria) correspondiendo el negro a la pieza y las regiones blancas a los defectos etiquetados.

Asimismo se facilitan unas imágenes con ningún o apenas defectos para la comprobación del comportamiento en estos casos.

## Aplicación y clases

La aplicación utiliza los ficheros y repositorio de *Detectron2*, por lo que en el *container* de *Docker* deberá estar en el mismo directorio para su carga.

El proyecto consta de por un lado 2 ficheros *Python*, 5 ficheros *HTML* y un último fichero de estilo *CSS*.

### Ficheros *Python*

En esta sección no se entrará en detalle en el funcionamiento interno de *Detectron2*, pero se ha de destacar que se hace uso de la clase *DefaultPredictor* para instanciar el objeto que devolverá los resultados de la imagen.

### Clase *Detector*

Clase muy sencilla que se inicializa con la configuración por defecto para ejecutar la detección. Para ello recibe el fichero pesos del modelo a ejecutar y un parámetro confianza que marcará el mínimo de *score* o puntuación que requiere un defecto para ser considerado como tal en los resultados. Generada la configuración, instancia un *DefaultPredictor* de *Detectron2* con la configuración indicada en el método *inference()*, carga la imagen, ejecuta la detección de la imagen y devuelve los resultados.

Los métodos *getOutputMask()* y *getConfidence()* son auxiliares y devuelven por un lado la máscara binaria generada de la detección y el vector de *scores* o confianza de cada defecto detectado en caso de detectarse alguno. Por defecto el umbral es de un 70 %.

Detector
+ <code>__init__(self, modelo, confianza=0.7)</code>
+ <code>inference(self, fichero):</code>
+ <code>getOutputMask(self, salida):</code>
+ <code>getConfidence(self, salida):</code>

Figura C.1: Clase *Detector*

### Fichero `app.py`

Este fichero contiene la aplicación en *Flask* como tal y es la que conecta el objeto *Detector* de la clase anterior con el usuario. Gestiona los ficheros *HTML* y la visualización de la aplicación web. Está dividido en `@app.routes` que indican en qué sección de la aplicación se ejecuta cada método y si responde a peticiones *POST*, por ejemplo.

Es una clase muy extensa y a pesar de estar relativamente modularizada con los métodos, se derivó más de los mismos a esta misma desde la clase *Detector* al presentar algunos problemas como la generación de la propia máscara completa en la clase que podría no detectar el mismo orden al cargarse en la aplicación.

El objeto *Detector* se instancia en cada ejecución. Esto se planteó como un problema ya que gracias a por ejemplo el patrón *Singleton* se podría reutilizar el objeto ya instanciado y optimizar el uso. La principal razón de la estructura actual es que si se quiere cargar la configuración del usuario en el *DefaultPredicor* debe de ser al instanciar de nuevo el objeto.

### Ficheros *HTML*

Son la parte *Frontend* de la aplicación y contienen el correspondiente código para la generación de cada una de las partes de la web. Se ha optado por añadir *Bootstrap* para una mejor apariencia y unas secciones simples de *JavaScript* con peticiones *Ajax* que hacen más interactiva la web en lugar de tener que recargar cada fichero *HTML* cada vez que el usuario realiza una acción.

### Fichero *CSS*

Contiene el estilo de los ficheros *HTML* para controlar aspectos como la posición y apariencia de los mismos.

### Fichero registro

Fichero *.csv* simple que almacena un identificador de fila, número de imágenes procesadas en un mismo día y 3 columnas que contabilizan cuantos defectos de cada tamaño se han detectado. Se va actualizando conforme avanza la ejecución.

### C.3. Diseño procedimental

A continuación se muestra el diagrama de secuencia que detalla la iteración del usuario con la aplicación y cada una de sus secciones.

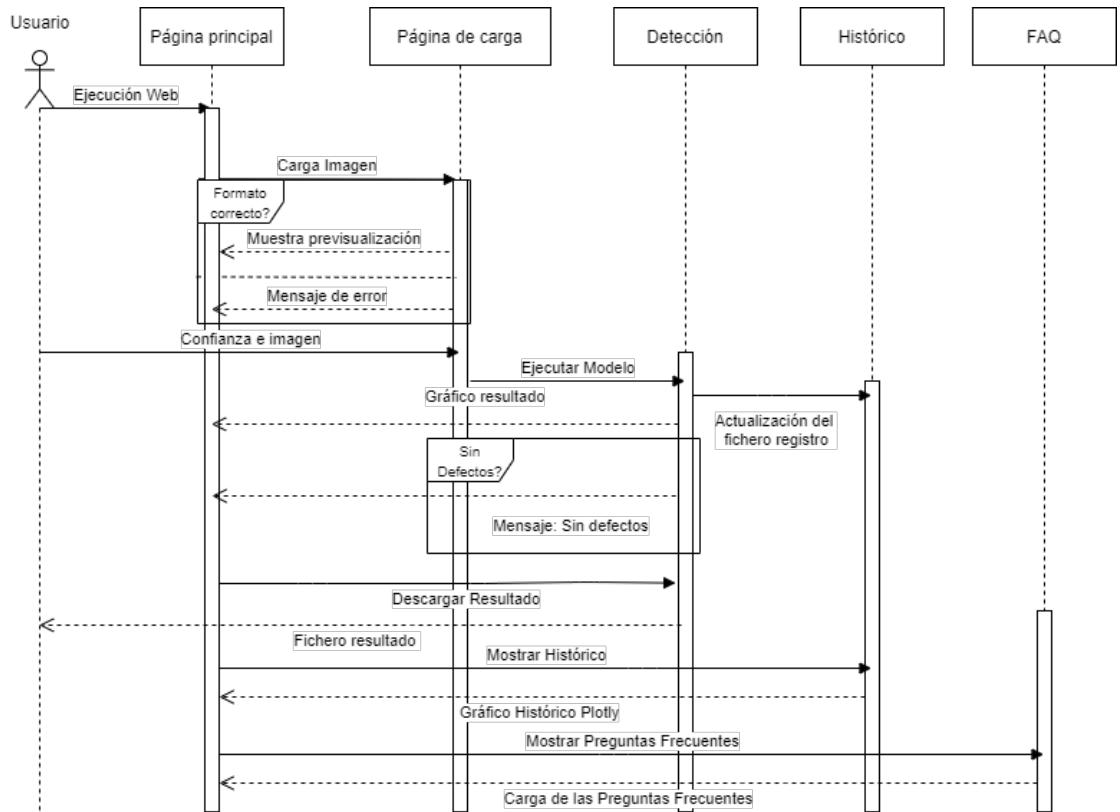


Figura C.2: Diagrama de secuencia de la aplicación

## C.4. Diseño arquitectónico

La estructura de la aplicación se divide en la parte de *Backend* ó código *Python* y *Frontend* que es el contenido de la web y sus ficheros.

### Backend

Los ficheros de la aplicación deben de estar inmediatamente situados en el directorio donde se sitúa la carpeta *detectron2* de la cual se cargará la herramienta. Tanto el fichero *app.py* como *detector.py* deben de estar en el mismo directorio. El fichero *registro.csv* también deberá estar en el mismo directorio o la aplicación creará uno de nuevo.

### Frontend

Junto al fichero *app.py* que representa la aplicación *Flask* se encuentran los siguientes directorios:

- static
  - uploads: Directorio donde se almacenarán las imágenes resultado una vez ejecutado el modelo.
  - css: Contiene el fichero *style.css* de estilo web
- templates: Contiene los ficheros *HTML* que representan cada sección de la aplicación

Esta estructura se detallará en el siguiente apartado, mostrando la estructura de directorios.





## Apéndice *D*

---

# Documentación técnica de programación

---

### D.1. Introducción

En este apartado se expone la estructura del proyecto, manual para el programador, una guía de instalación y correcta ejecución del proyecto, además de pruebas realizadas sobre su funcionamiento.

### D.2. Estructura de directorios

La estructura se contempla una vez ejecutado el fichero *Dockerfile* que construye la imagen que contiene, por un lado el entorno original facilitado por *Detectron2* en su *Dockerfile* oficial que se ha modificado para este proyecto y además el repositorio de *Github* descargado y extraído.

La carpeta **proyecto** contiene el proyecto directamente clonado, pero el fichero *Dockerfile* dispone todo según se corresponde automáticamente, no obstante el contenido de *proyecto/src* está tanto en dicho directorio como en el directorio padre.

Además es importante remarcar la presencia de los siguientes ficheros:

- **app.py** y **detector.py** deberán estar a la misma altura de directorios que la carpeta **detectron2**
- Si no está presente **registro.csv** en dicho directorio se creará uno nuevo perdiendo la información guardada hasta ese momento

```
/
├── detectron2
├── configs
├── imagenes .2 setup.py
├── proyecto
│   ├── src
│   └── ...
├── descargaModelo.py
├── requirements.txt
├── modelos.json
├── app.py
├── descargaModelo.py
├── detector.py
├── registro.csv
├── templates
│   ├── base.html
│   ├── faq.html
│   ├── deteccion.html
│   ├── index.html
│   └── historico.html
├── static
│   ├── css
│   │   └── style.css
│   └── uploads
│       └── graficoHistorico.html
└── modelo-0.1.pth
```

Figura D.1: Estructura del Proyecto

- La descarga del modelo es bastante pesada, y aunque cabía la posibilidad de lanzar el fichero **descargaModelo.py** si no se encuentra el modelo para así asegurar la ejecución, el proceso de descarga se puede demorar por lo que se obvia la presencia del fichero *modelo-0.1.pth*

- Para poder mostrar un histórico y en caso de que no haya valores suficientes, el fichero *graficoHistórico.html* debe estar presente desde el primer momento

## D.3. Manual del programador

En este apartado se detallará cómo está organizado el proyecto y qué ficheros se han utilizado en el desarrollo para que en un futuro pueda ser comprendido correctamente y ayude en el proceso de edición o mejora.

### Ficheros y directorios

- **src/app.py:** Fichero principal de la aplicación que contiene el código *Flask*, está estructurado en *routes* que identifican cada método o función con una sección de la web y las peticiones a las que responde. A nivel de configuración y más allá del contenido de los propios métodos, se puede personalizar la dirección IP en la que se servirá la aplicación web y el modo *debug* en la parte inferior de la aplicación.
- **src/detector.py:** Fichero que contiene la clase del detector creado para este proyecto. Utiliza el objeto *DefaultPredictor* de *Detectron2* para instanciarlo, darle una configuración o *cfg* y ejecutar la detección sobre el fichero que recibe.
- **src/static/css/style.css:** Fichero de estilos para los archivos *HTML* en *templates*. Si se modifica la ruta deberá de modificarse también la referencia en dichos ficheros.
- **src/static/uploads/graficoHistórico.html:** Fichero generado por *Plotly* y que por si mismo contiene todo el código para ser interactivo. Puede abrirse en el navegador de forma independiente.
- **src/templates/base.html:** Fichero *HTML* que se utiliza principalmente para organizar la *navbar* de *Bootstrap* y las secciones que comparten todas las páginas, de esa forma el resto de ficheros *HTML* extienden de este mismo y puede modificarse a la vez todos las secciones comunes desde este fichero.
- **Dockerfile:** Fichero para la importación y ejecución creado a partir del fichero oficial de *Detectron2* disponible en su repositorio<sup>1</sup>. Partiendo

---

<sup>1</sup><https://github.com/facebookresearch/detectron2>

de una imagen de *NVIDIA* se establece un usuario, un directorio padre y se clona los ficheros de *Detectron2* instalando las dependencias.

Posteriormente desde *Github* se importa este proyecto y se estructuran los ficheros más importantes. Por último con el fichero *descargaModelo.py* se descarga desde *Google Drive* el fichero pesos debido a que *Github* no permite alojarlo debido a su tamaño. Una vez finalizado el proceso la imagen está lista para generar contenedores.

Se ha probado a generar imágenes de menor tamaño usando por ejemplo la imagen oficial de *Docker Hub* de *Debian* pero puede mostrar incompatibilidades con *Detectron2* y por ello no se ha utilizado finalmente.

- **registro.csv:** Fichero que contiene el registro de detecciones por tamaño y fecha.
- **modelo.pth:** Fichero de pesos generado por la red neuronal.
- **detectron2/:** Directorio oficial de *Detectron2*
- **Conversor y Registro COCO.ipynb:** *Notebook* creado para el recorrido, procesado y generación del fichero *JSON* para el registro del conjunto de imágenes en el proceso de entrenamiento y test.
- **Separador de Mascaras.ipynb:** *Notebook* que se creó con la idea de observar la diferencia de utilizar máscaras únicas o individuales. Partiendo de un fichero de máscara binaria, genera tantos como defectos contenga de forma individual.
- **requirements.txt:** Fichero de texto que contiene las bibliotecas requeridas para este proyecto, el fichero *Dockerfile* se encarga de recorrerlo e instalarlo automáticamente en el despliegue para que el usuario no tenga que hacerlo.
- **modelos.json:** Fichero de diccionarios que contiene los modelos registrados con la estructura : *Nombre:URL* de forma que la aplicación al iniciarse descarga el fichero y lo sobrescribe, lo recorre y comprueba el número de versión para descargar la última versión. Dichos modelos también son alojados en *Google Drive* debido a su tamaño.
- **updateDocker.bat:** Fichero informal que se creó a la hora de trabajar con *Pycharm* y *Visual Studio Code*. Ya que en Windows no es posible la ejecución y pruebas del proyecto, contiene un par de líneas

que permiten copiar el contenido completo del proyecto al contenedor *Docker* ejecutándose y actualizando el contenido para su prueba inmediata. Los comandos siguen la estructura:

```
docker cp directorioHostOrigen nombreContenedor:directorioDestinoContenedor
```

## D.4. Compilación, instalación y ejecución del proyecto

En esta sección se describirá el proceso para la instalación y uso del proyecto de cara al desarrollo y modificación por parte de un programador. Para la instalación y uso por parte del usuario hay más detalles en la sección **Instalación**

Hay 2 alternativas para la instalación del proyecto con intención de modificarlo:

- Instalar mediante el *Dockerfile* la imagen que se facilita y que contendrá el proyecto, generar un contenedor y mantenerlo en ejecución mientras se hacen los cambios normalmente desde el *host* con un editor más amigable, ya que la interfaz de los contenedores es algo tosca.
- Inicializar un directorio en una cuenta personal en *Google Colab* que finalmente se almacenará en la unidad de *Google Drive* de la cuenta asociada

Debido a que el primer punto se cubrirá de forma más detallada en la sección de usuario, se procederá ahora a explicar cómo se podría preparar un directorio personal en *Google Drive* y clonar el proyecto para editar y ejecutarlo desde *Google Colab*.

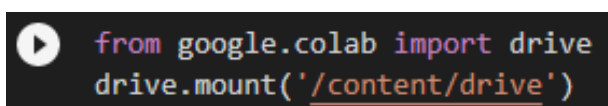
Sin embargo este método tiene un problema y al alojar nuestros ficheros en *Google Drive*, no podemos hacer uso de un editor de texto o *IDE* que nos facilite la edición de código. Una solución puede ser la sincronización de *Visual Studio Code* con *Google Drive*, pero ya que en este proyecto se ha trabajado con el proyecto local, no se abordará esta situación.

1. **Creación del directorio:** Primero se accede mediante una cuenta personal *Gmail* a la unidad de *Google Drive* donde según se considere se creará un directorio nuevo para una mejor organización.

2. **Google Colab y creación del *Notebooks*:** Con este método se trabajará de forma similar a la de *Jupyter* de *Anaconda*, que es posible que el lector esté familiarizado.

Se accede a *Google Colab*<sup>2</sup> y seguramente se presente un *Notebook* de introducción. Se crea uno nuevo con un nombre a elegir libremente pero manteniendo la extensión *.ipynb*.

3. **Disposición e instalación:** El primer paso es integrar la unidad de *Google Drive* con el *Notebook*.



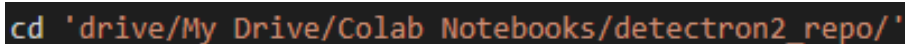
```
from google.colab import drive
drive.mount('/content/drive')
```

Figura D.2: Integración de Google Drive

Las diferentes celdas se ejecutan con un icono a la izquierda aunque desde el menú *Entorno de ejecución* es posible ejecutarlas todas.

Es posible que el código de la figura superior nos pregunte por un código generado por seguridad en un enlace, teniendo iniciada la sesión de *Google* basta con visitar la dirección y copiar y pegar el código que se facilita. Hecho esto se montará la unidad de *Google Drive*.

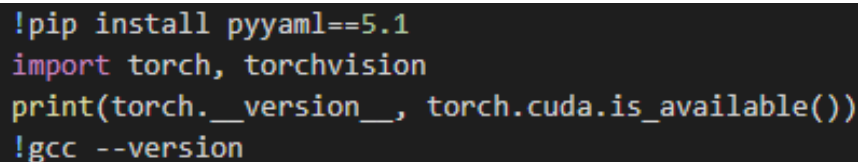
A partir de aquí es posible recorrer los directorios con *cd* ó *ls* para ver el contenido.



```
cd 'drive/My Drive/Colab Notebooks/detectron2_repo/'
```

Figura D.3: Recorrido de directorios

A continuación instalamos e importamos *pyyaml* y *torch*



```
!pip install pyyaml==5.1
import torch, torchvision
print(torch.__version__, torch.cuda.is_available())
!gcc --version
```

Figura D.4: Instalación de dependencias

Posteriormente se clona e instala el repositorio oficial de *Detectron2*

---

<sup>2</sup><https://colab.research.google.com/notebooks/intro.ipynb>

```
!pip install -U torch torchvision
!pip install git+https://github.com/facebookresearch/fvcore.git
!git clone https://github.com/facebookresearch/detectron2 detectron2_repo
!pip install -e detectron2_repo
```

Figura D.5: Instalación de Detectron2

Este proceso se puede demorar y hay que tener en cuenta de que al no estar en un entorno virtual, puede haber dependencias no instaladas, por lo que es posible que al ejecutar determinadas celdas se nos solicite la instalación de alguna biblioteca mediante *pip install*. Dichos comandos pueden ejecutarse normalmente de forma individual por celda pero si se ejecutan varios deben de ir precedidos por el símbolo *!* de cierre de exclamación.

Por último se importan bibliotecas y clases básicas de *Detectron2* como se recomienda en la documentación oficial.

```
# Some basic setup:
# Setup detectron2 logger
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import os, json, cv2, random
from google.colab.patches import cv2_imshow

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
from detectron2.evaluation import COCOEvaluator
```

Figura D.6: Importación de bibliotecas básicas

4. **Clonación del proyecto:** Posteriormente se clona el proyecto con un comando de *git* asegurándonos de estar dentro del directorio de *Detectron2*

```
! git clone https://github.com/fyi0000/TFG-GII-20.04
```

5. **Organización del proyecto:** Una vez clonado el proyecto, se deben de disponer los ficheros de forma que *app.py*, *detector.py*, *descargar-Modelo.py*, *static* y *templates* estén a la misma altura que el directorio *detectron2*.

Si hay algún problema de importación simplemente usamos *cd* ó *ls* para asegurarnos que la disposición es correcta.

6. **Uso de ngrok:** *ngrok* nos permite ejecutar de forma local en *Colab* una aplicación *Flask* y a la vez acceder como si se estuviese ejecutando en nuestra máquina, para ello primero instalamos la biblioteca con:

*!pip install flask-ngrok*

Y posteriormente editamos el fichero *app.py* añadiendo estas líneas en la cabecera

```
from flask_ngrok import run_with_ngrok
run_with_ngrok(app)
```

Figura D.7: Importación de bibliotecas básicas

Ahora cuando se ejecute la aplicación, además de la traza normal de *Flask* nos facilitará una *URL* temporal acabada en *ngrok.io* que nos llevará a la aplicación.

```
!python app.py
```

Figura D.8: Importación de bibliotecas básicas

Conviene recordar además que la ejecución continuada en *Google Colab* está limitada y podría dejar de ejecutarse nuestra aplicación si se mantiene en ejecución durante demasiado tiempo. El límite es de unos 60 minutos.



## Funcionamiento concurrente de la aplicación

A nivel de programador es necesario tener presente que *Flask* solo admite por defecto una petición de forma síncrona, por lo que como se ha observado en las pruebas que se comentan en la siguiente sección, las detecciones concurrentes no son una opción por defecto.

Sin embargo y a juicio del lector, es posible indicar en el fichero *app.py* a *Flask* que se admitan diferentes peticiones simultáneas, aunque se avisa de que el rendimiento se puede degradar notablemente.

La forma más común es añadir en la última fila del fichero *app.py*, en el método `__init__` donde tiene lugar el lanzamiento y se indica el puerto, la frase `threaded=True` de forma que el resultado sería:

```
app.run(host=0.0.0.0, threaded=True)
```

## D.5. Pruebas del sistema

Prueba Realizada	Resultado
Iniciar la aplicación mediante “sudo python app.py”	La aplicación funciona con normalidad.
Iniciar la aplicación mediante “python app.py”	La aplicación ha funcionado en ocasiones pero al no tener permisos de escritura lanza errores.
Acceder a la sección de <i>FAQ</i> (Preguntas frecuentes)	Se accede correctamente y los hiperenlaces son accesibles.
Detección de una imagen <i>test</i>	La aplicación funciona y muestra resultados. Las descargas funcionan correctamente.
Detección de una imagen <i>training</i>	La aplicación funciona con normalidad y los resultados son excesivamente buenos, como se esperaba.
Detección en una radiografía sin defectos	Se muestran los resultados y se indica correctamente que no hay detecciones. Algunas marcadas por los tutores aparentemente sin defectos parecen tener alguno pequeño.
Detección con imagen ajena al conjunto	La aplicación funciona y a veces detecta falsos positivos y a veces no detecta nada y lo indica correctamente.

Tabla D.1: Pruebas de la aplicación

Prueba Realizada	Resultado
Detección de imagen con extensión <i>PNG</i>	La detección es correcta.
Detección de imagen con extensión <i>JPG</i>	La detección es correcta.
Detección de imagen con extensión <i>JPEG</i>	La detección es correcta.
Detección de fichero <i>PDF</i>	La aplicación rechaza el fichero por extensión correctamente.
Acceso desde dos pestañas concurrentes	La aplicación funciona correctamente.
Detección de dos ficheros de forma concurrente	Como es de esperar, <i>Flask</i> solo admite una petición y muestra en ambas el resultado de la última imagen que le ha sido enviada.
Descarga de resultados	Se descargan correctamente, algo más de demora en la composición pero todo funciona según requisitos.
Borrado del fichero “modelo-0.1.pth” y detección	Se muestra un error web y en la consola se muestra la traza de la excepción y remarcado el mensaje de captura preguntando si existe dicho fichero.
Actualización del registro tras detección	Se actualiza la fila correspondiente o se añade una fecha nueva de forma correcta.

Tabla D.2: Pruebas de la aplicación

Prueba Realizada	Resultado
Actualización del modelo	La aplicación descarga el nuevo fichero y actualiza la variable sesión que indica que fichero se usará en la próxima instanciación del Detector.
Actualización del modelo con enlace erróneo	Al comprobarse de forma independiente tras el proceso la existencia del fichero mostrado en <i>modelos.json</i> , al no encontrarse se indica que no se ha podido actualizar.
Reinicio tras actualización	Se muestra la versión actual correcta y se añade que es la más actual.
Detección de una imagen de dimensiones reducidas	Al ser tan pequeño, <i>Plotly</i> genera un gráfico de unas dimensiones también muy pequeñas, pero de esta forma se ajusta mejor a imágenes de un tamaño normal.
Ejecución de la aplicación sin conexión a Internet	Los CDN no funcionan y la apariencia se degrada, la función de actualización y <i>loaders</i> pierden funcionalidad.

Tabla D.3: Pruebas de la aplicación

## Apéndice *E*

---

# Documentación de usuario

---

### E.1. Introducción

A continuación se explica cómo es el proceso de instalación completo del proyecto, desde la instalación de *Docker* hasta el uso del fichero *Dockerfile* que se recuerda está en el repositorio del proyecto<sup>1</sup>.

### E.2. Requisitos de usuarios

Es imprescindible que, al menos para la instalación y actualización del modelo se disponga de **conexión a Internet**.

Además se debe contemplar que tanto *jQuery* como *Bootstrap* se utilizan mediante un *CDN* (*Content Delivery Network*), que evita la descarga de los ficheros para que ambos componentes funcionen. Por ello de no haber conexión a internet determinadas peticiones no podrían resolverse y la aplicación perdería funcionalidades.

Como ya se ha expuesto anteriormente, oficialmente *Detectron2* no es compatible con Windows, por lo que, y según su web oficial<sup>[1]</sup> los requisitos son:

- *Linux* o *MacOs* con una versión de *Python* 3.6 o superior
- *Pytorch* y *torchvision* versión 1.6 o superior
- *OpenCV* es opcional pero se recomienda para la visualización

---

<sup>1</sup>Repositorio: <https://github.com/fyi0000/TFG-GII-20.04>

Además, por la forma como está elaborado el proyecto, se requiere tener instalado *Docker*. La instalación que se detalla no es para nada compleja pero es para *Windows* y puede variar en el entorno Linux o similares aunque los comandos de uso son los mismos.

Una vez generada la imagen ocupa un total de unos 7.5-8GB de espacio en disco. Se han hecho pruebas con la imagen sobre la que se genera la imagen, siendo esta de *Nvidia* y la indicada por *Detectron2*. A pesar de probar con imágenes como *Debian*, instalar las dependencias no es suficiente y puede generar errores. Por ello no ha sido posible minimizar más el espacio.

### E.3. Instalación

1. El primer paso es acceder a la web de *Docker* y en concreto a la sección de *Windows*:

<https://docs.docker.com/docker-for-windows/install/>

La instalación es prácticamente automática y solo hay que esperar a que se complete.

2. Una vez instalado, se aconseja utilizar *WSL 2 based engine* que es una alternativa a *Hyper V* para la virtualización.

**Nota importante:** En los equipos que se han utilizado ha sido además necesario la instalación de una actualización del kernel de *Linux* y posterior reinicio. Puede no ser el caso pero si *Docker* no inicia correctamente, se adjunta el instalador de la última versión en el soporte digital y si no acceder a a:

<https://docs.docker.com/docker-for-windows/wsl/>

ó

<https://docs.microsoft.com/es-es/windows/wsl/install-win10#step-4---download-the-linux-kernel-update-package>

Se ha de tener en cuenta que los comandos que se detallan a continuación y en general todas las funciones de *Docker* no funcionarán salvo que se esté ejecutando el *daemon*. Por ello confirmar que no hay una actualización en curso y que está funcionando correctamente si se observa el siguiente icono estático en la barra de tareas



Figura E.1: Icono de ejecución Docker

3. Una vez instalado y teniendo *Docker* en ejecución, se debería de clonar el proyecto desde el repositorio mediante un comando:

```
git clone https://github.com/fyi0000/TFG-GII-20.04
```

O bien desde las unidades digitales facilitadas que contienen los mismos ficheros más el modelo. Al final se tendría que observar un directorio similar al siguiente:

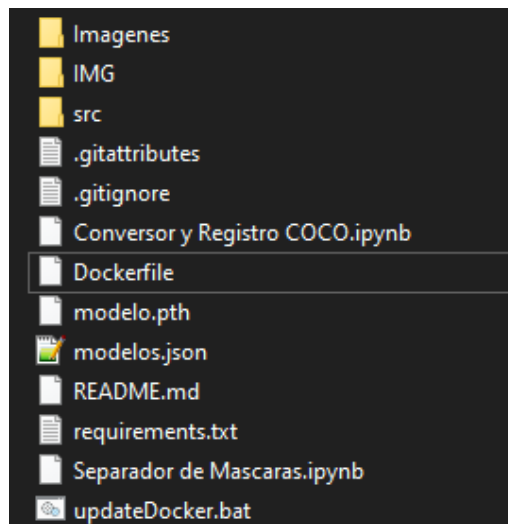


Figura E.2: Directorio tras clonación o copia

4. A partir de este punto se recomienda utilizar el *PowerShell* para llegar al directorio donde se encuentra el fichero *Dockerfile* sin extensión, aunque también es posible utilizar la Consola normal.

Una forma rápida es pulsar *Shift Izquierdo + Botón derecho* y se mostrará esta opción:

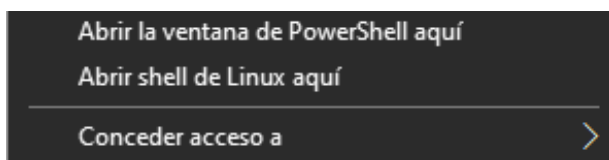


Figura E.3: Opción de apertura directa PowerShell

Que como se puede comprobar nos permite abrir la ventana en el propio directorio sin necesidad de utilizar *cd's*.

5. A continuación ejecutaremos el comando *Docker* que a partir del *Dockerfile* facilitado, construye la imagen a partir de la cual se pueden generar los contenedores. El comando es:

*docker build . -t nombreimagen*

El punto a continuación de *build* indica que el fichero está en el directorio actual, aunque como se puede suponer es posible ejecutar la consola en cualquier directorio y *apuntar* al fichero *Dockerfile* pasando su ruta como este primer directorio. A continuación de *-t* o *tag* se especifica un nombre de la imagen en minúscula.

```
\\TFG-GII-20.04> docker build . -t imagendt2
```

Figura E.4: Comando build de imagen

6. Debido al tamaño de la imagen y que además se instalan todas las dependencias contenidas en el fichero *requirements.txt*, el proceso puede superar los 10 minutos hasta que se complete la creación de la imagen. Aunque la traza del proceso simplemente indica lo que está haciendo, es importante por ejemplo, y si es posible, asegurarse de que el comando “RUN python descargaModelo.py” se demora un tiempo ya que nos



```
=> [18/20] RUN cp /lib/modules/5.15.0-76-generic/kernel/drivers/net/ethernet/mellanox/mlnx_core/libmlx5/libmlx5.ko
=> [19/20] RUN pip install --user -r requirements.txt
=> [20/20] RUN ["python", "descargaModelo.py"]
```

Figura E.5: Descarga de modelo

indica si ha podido haber algún problema con el modelo alojado en *Google Drive*.

7. Terminado el proceso, abrimos *Docker* y nos vamos a la sección *images* en la parte superior izquierda donde deberíamos ver una imagen de entorno a 8GB y con el nombre o *tag* que le hemos dado en el paso 5 de este apartado.

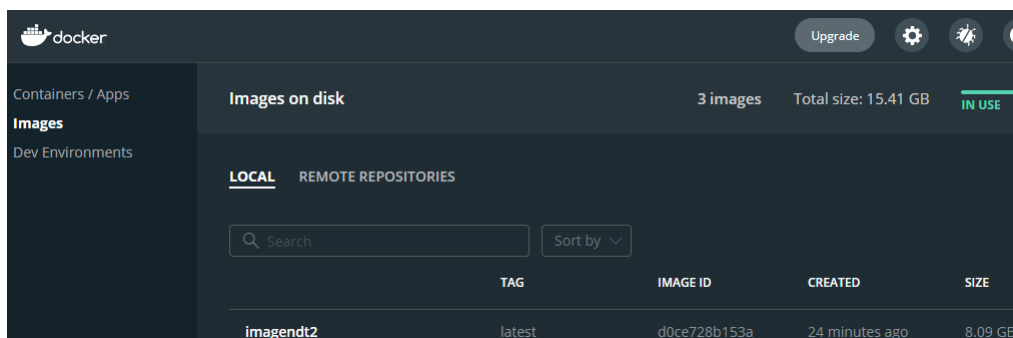


Figura E.6: Sección Images en Docker

8. Confirmado que la imagen se ha creado correctamente, pulsamos sobre el botón *RUN* que aparece al colocar el cursor sobre la imagen.

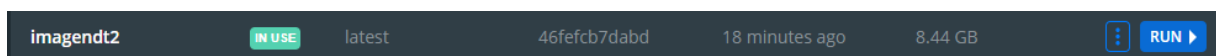


Figura E.7: Creación de contenedor desde imagen

Ahora se nos abrirá una pequeña ventana de creación directa. **No pulsamos Run** y desplegamos las *Optional Settings*. Ahora asignamos un nombre a nuestra elección al contenedor y en *Local Port* o Puerto Local escribimos **5000**. Hacemos esto ya que por defecto es el puerto expuesto en *Flask*.

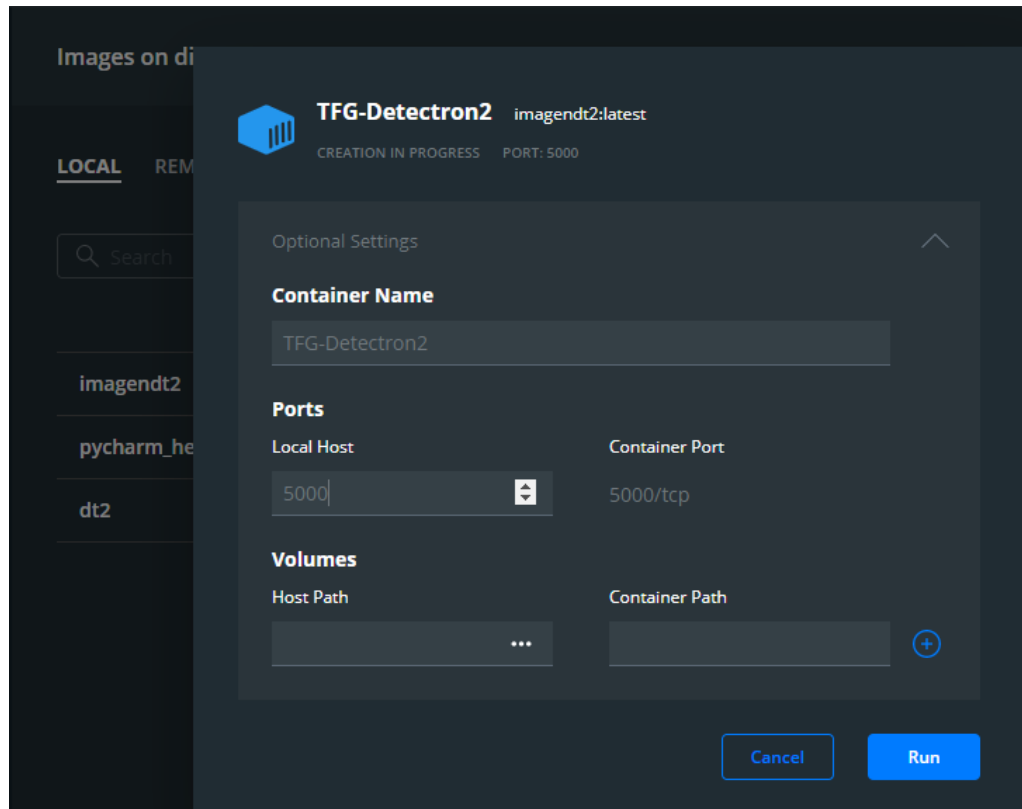


Figura E.8: Configuración del contenedor

9. Ahora nos vamos a la sección *Containers/Apps* de *Docker* y confirmamos que está el contenedor que acabamos de crear ejecutándose.



Figura E.9: Contenedor correctamente creado

Para comprobar que todo ha funcionado:

- El contenedor se mantiene en ejecución
- En texto azul y a la derecha del nombre del contenedor está nuestra imagen inicialmente creada
- El puerto que se indica debajo del nombre es efectivamente el 5000

A la derecha del nombre nos aparecen diferentes botones, los que se usarán son:

- 2º para lanzar la línea de comandos del contenedor
- 3º para iniciar o detener el contenedor
- 5º para borrar el contenedor, se debe tener en cuenta que no se puede borrar una imagen si hay contenedores dependientes existentes

Y por último podemos lanzar la línea de comandos, con el 2º botón anteriormente mencionado e iniciando el contenedor si no lo estaba, y hacer un `ls` para comprobar que todo está correctamente estructurado.

```
$ ls
GETTING_STARTED.md  __pycache__  demo          dev           proyecto      static
INSTALL.md           app.py        descargaModelo.py  docker       registro.csv  templates
LICENSE              build         detector.py      docs          requirements.txt  tests
MODEL_ZOO.md         configs       detectron2       modelo-0.1.pth  setup.cfg      tools
README.md            datasets     detectron2.egg-info  projects      setup.py
```

Figura E.10: Directorio correcto

Ficheros que deben estar presentes:

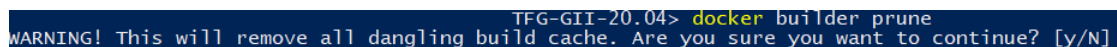
- **app.py**: Es la aplicación y debe estar presente en este punto.
- **detector.py**
- **descargaModelo.py**: Es el script auxiliar que descarga el modelo inicialmente.
- **detectron2**: Directorio de *Detectron2* necesario para la ejecución de la aplicación. Nos indica que se ha clonado correctamente e instalado.
- **modelo-0.1.pth**: Modelo inicial que debe estar presente siempre.

- **static y templates:** Directorios que contienen la estructura web luego han de estar localizados aquí.
- **registro.csv:** Fichero que inicializa el histórico.
- **static/uploads/graficoHistorico.html:** Histórico inicial

De no estar estos ficheros o estar en otra disposición puede dar lugar a errores.

Como punto final es posible que se quiera limpiar la *caché* del *builder* que aunque *Docker* la limpia, se puede hacer de forma inmediata con:

*docker builder prune*



```
TFG-GII-20.04> docker builder prune
WARNING! This will remove all dangling build cache. Are you sure you want to continue? [y/N]
```

Figura E.11: Limpieza de la cache de instalación

Confirmando que realmente se desea limpiar la caché de toda la instalación de esta sección.

Hasta este punto se considera la instalación de la herramienta, el uso y manual correspondiente se detallará en el apartado que viene a continuación.

## E.4. Manual del usuario

En esta sección se detalla el uso como usuario de la aplicación en cada una de sus partes.

### Ejecución y acceso

Suponiendo que el proceso de la sección anterior haya funcionado correctamente, ya se está en disposición de iniciar la aplicación.

Primero se iniciará la aplicación *Docker* se encontraba ya en ejecución y de igual manera el contenedor anteriormente creado.

Ahora se accede mediante el 2º botón del contenedor a la línea de comandos. Para iniciar la aplicación y asegurarse de que la ejecución tiene los permisos, se hará con la palabra *sudo*, ya que se ha probado a ajustar permisos con *chmod* en el propio *Dockerfile* pero han surgido problemas de igual manera.

```
sudo python app.py
```

**Nota:** Es posible configurar el contenedor para el lanzamiento inmediato de nuestra aplicación, para ello basta con añadir al *Dockerfile* la línea

```
#CMD ["python", "app.py"]
```

que se encuentra comentada en el *Dockerfile* facilitado. El problema que presenta es que la ejecución no siempre es como *sudo* y de haber un fallo, el contenedor se detendrá y tendremos que reconstruir la imagen de nuevo desde el *Dockerfile*. Por ello se ha optado por dejar la ejecución de forma manual.

*Flask* nos indica ahora que es un *microframework* y por lo tanto el mensaje que no recomienda su uso en producción es normal, dado que no se recomienda someterlo a múltiples peticiones. Para este proyecto su rendimiento es perfectamente suficiente luego no hay problema.

En la parte inferior nos indica la dirección donde se ejecuta nuestra aplicación web seguido del puerto que se expuso cuando se configuró el contenedor. Accedemos desde el navegador escribiendo la dirección o bien:

```
http://localhost:5000/
```

```
$ sudo python app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on all addresses.
  WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://172.17.0.2:5000/ (Press CTRL+C to quit)
```

Figura E.12: Lanzamiento de la aplicación

Una vez accedemos se nos muestra el inicio con una pequeña sección de información

## Inicio

- Detector de defectos metálicos basado en Detectron 2.
- Autor: Fco Javier Yagüe Izquierdo.
- Año: 2021

Detectar >

Figura E.13: Página de inicio

A su vez en la parte superior izquierda tenemos una *navbar* con la que podemos navegar de forma rápida por todas las secciones

Inicio   Detección   Histórico   FAQ

Figura E.14: Navbar superior

## Detección

Si queremos empezar con la detección, podemos acceder desde el botón *Detectar* que se nos muestra el inicio o desde la correspondiente opción de la *navbar*.

En esta sección se nos muestra un *input* para ficheros que no permite continuar a no ser que se haya facilitado un fichero.

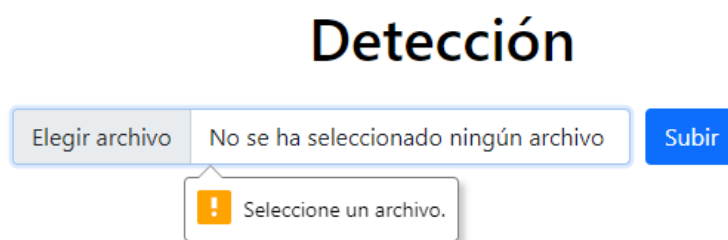


Figura E.15: Fichero requerido

A su vez las restricciones de imagen debido a los formatos que se ha comprobado que funcionan en *Detectron2* son *PNG*, *JPG* y *JPEG*, rechazando cualquier otro fichero.

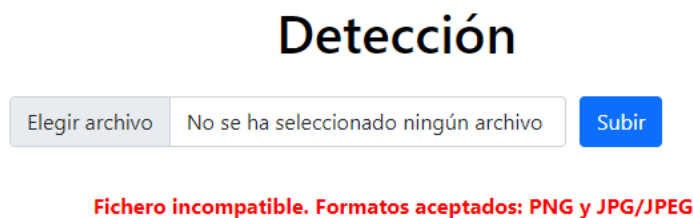


Figura E.16: Formatos requeridos

Si la imagen que se facilita es correcta se obtiene una previsualización de la misma y un *slider* para marcar la confianza o seguridad mínima que debe de tener un defecto detectado para ser mostrado en los resultados.

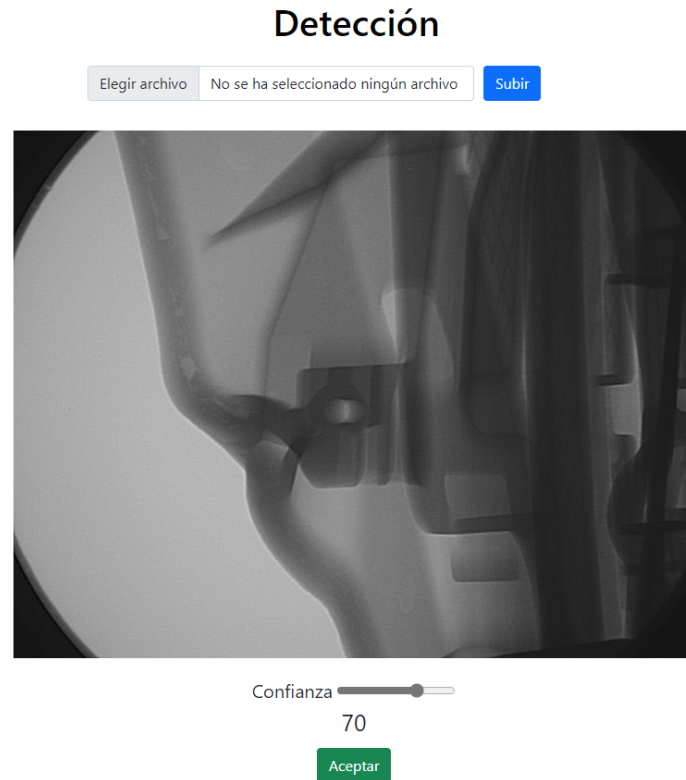


Figura E.17: Previsualización de imagen a detectar

Si todo es correcto se pulsa *Aceptar* y comienza el proceso mostrando un *loader*.

### Detección



Figura E.18: Loader de detección

Una vez finaliza el proceso, se muestra un gráfico *Plotly* mostrando los resultados y detecciones.





Figura E.19: Muestra de resultados

Si se coloca el cursor sobre algún defecto, se obtiene además información sobre el mismo como el área que ocupa, tamaño que se le otorga según el área, confianza de la detección y en el borde gris el identificador del defecto.



Figura E.20: Muestra de métricas

El gráfico *Plotly* tiene multitud de opciones y al colocar el cursor sobre él apareciendo diferentes controles



Figura E.21: Controles Plotly

Aunque son descriptivos por si mismos, algunas opciones son:

- Descargar como .png la perspectiva actual
- Aumentar el zoom o seleccionar herramienta mover
- Manipular la leyenda y restaurar el zoom inicial
- Enlace a la web de *Plotly*

Además es posible hacer zoom arrastrando y soltando el cursor sobre una zona específica

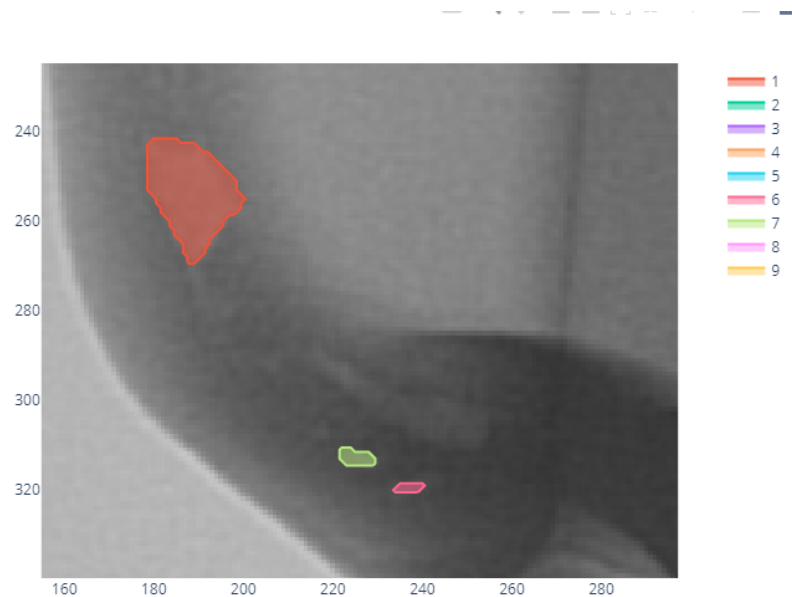


Figura E.22: Función de zoom

Pudiendo ahora aplicar otra herramienta interesante, si se hace click sobre la leyenda y en concreto el identificador de un defecto que queremos ocultar temporalmente.

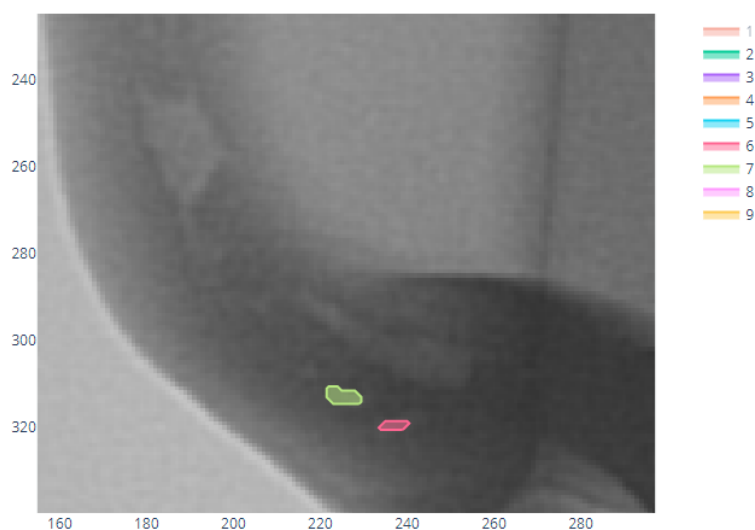


Figura E.23: Ocultando detecciones

Basta con volverá hacer click en dicha leyenda para que se vuelva a mostrar.

A su vez se muestran opciones de descarga de resultados, Máscara binaria por un lado y por otro tanto el propio gráfico *Plotly* que puede abrirse independientemente en el navegador como una composición imagen original - máscara binaria.

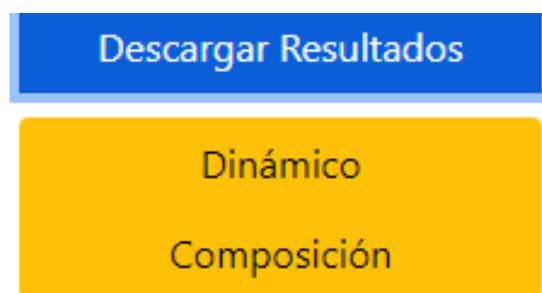


Figura E.24: Botones de descarga

La máscara binaria:



Figura E.25: Máscara binaria descargada

Composición:

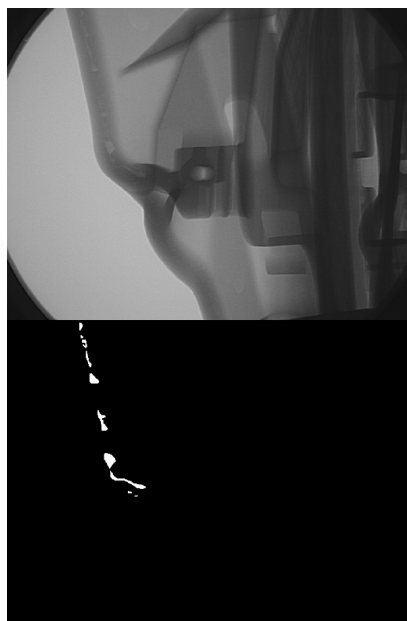


Figura E.26: Composición descargada

En caso de no haber detecciones se permitirá igualmente la observación con *Plotly* y la descarga del gráfico interactivo, indicando al lado del botón que los resultados están vacíos y por lo tanto no se han detectado defectos.

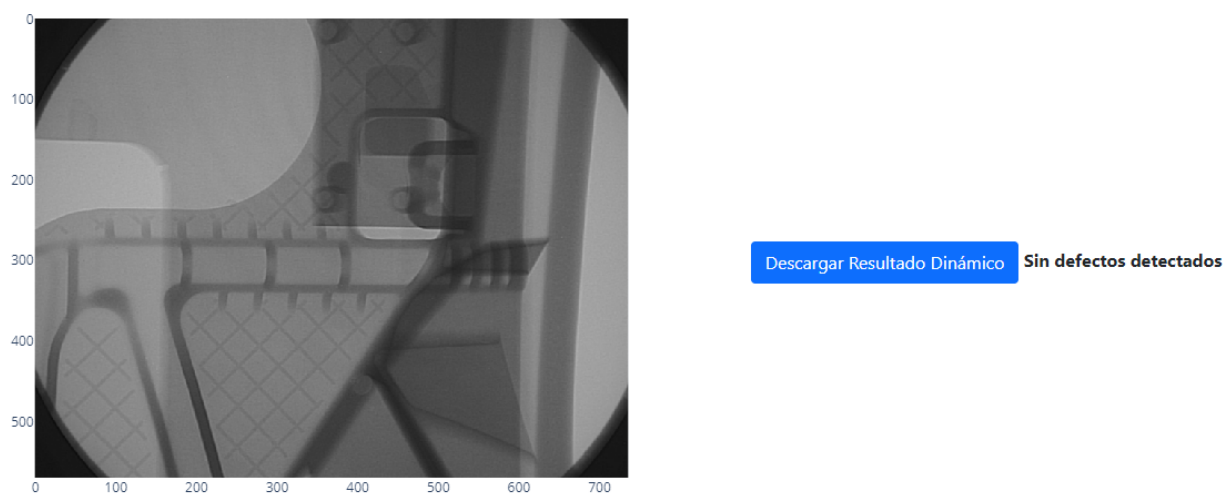


Figura E.27: Ejecución sin detecciones

## Histórico

Además si se accede a la sección Histórico se mostrará otro gráfico *Plotly* que muestra la sucesión de detecciones por fecha, dividiendo por el tamaño el número de defectos que se han detectado ese día y mostrando además el número de imágenes procesadas ese día para tener mayor perspectiva.

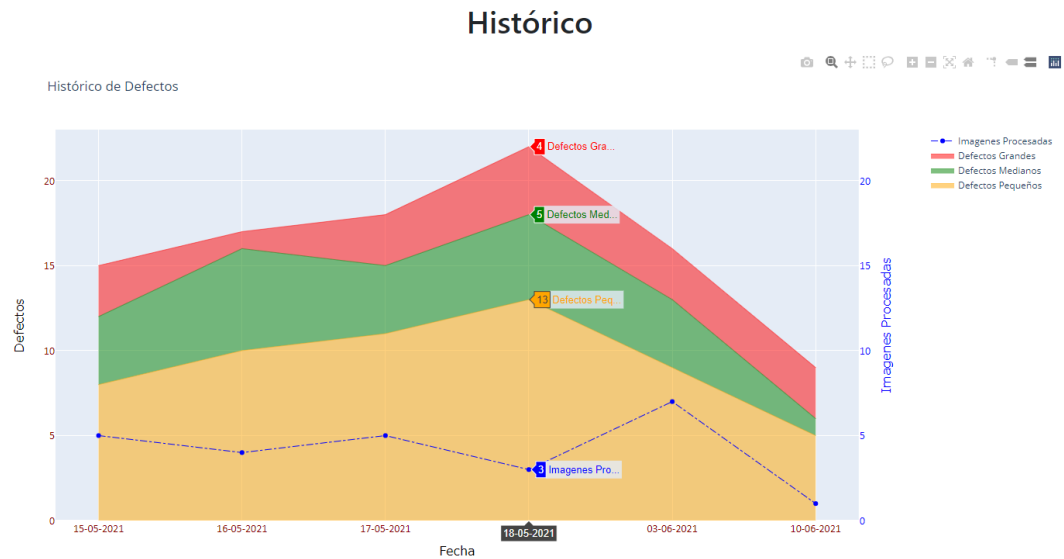


Figura E.28: Gráfico de histórico

Igualmente si se pasa el cursor por el gráfico además de las mismas opciones del gráfico resultados, se muestran los valores de la fecha sobre la que se sitúa el cursor.

### Actualización de modelo

Debido a que en un hipotético caso podrían incorporarse más imágenes al conjunto, sería posible reentrenar la red y obtener un mayor rendimiento. Por ello la aplicación comprueba cada vez que se lanza, el fichero *modelos.json* del repositorio donde se podría colocar en forma de diccionario, un nuevo modelo, indicando la versión y enlace de *Google Drive* para la detección y descarga.

Por defecto y para demostrar el funcionamiento, el *Dockerfile* descarga la versión 0.1 estando disponible la 0.2, por lo que se mostrará este mensaje.



Figura E.29: Actualización del modelo

Si se pulsa el botón *Actualizar Modelo* se iniciará la actualización y se mostrará un pequeño *loader*



Figura E.30: Loader de actualización

Si todo funciona aparecerá el siguiente mensaje indicando a qué versión se ha actualizado, si no se indicará que ha habido un problema en la actualización.

**Modelo actualizado a la version 0.2**

Figura E.31: Actualización correcta

Y en sucesivas ocasiones que se lance la aplicación, se mostrará un mensaje similar asegurando que la comprobación se ha realizado y el modelo local es el último disponible.

**El modelo esta actualizado v0.2**

Figura E.32: Modelo local actualizado

Debido a que la presencia de al menos un modelo es vital para que la aplicación cumpla su función, no se elimina el modelo anterior, lo que conviene tener presente a pesar de que no ocupan demasiado espacio por sí mismos.

## En caso de no poder descargar el modelo inicial

En los soportes digitales se facilita el modelo inicial además del script para su descarga. Si por algún motivo no se pudiese descargar y hacer funcionar la aplicación, los pasos serían los siguientes:

1. Iniciar el contenedor creado como se ha detallado anteriormente
2. Mediante el comando *pwd* es posible confirmar que la ruta es

*home/appuser/detectron2\_repo*

si no es así navegar con *cd* hasta encontrar *app.py*.

3. A continuación y en el directorio del anfitrión donde se encuentre el fichero *modelo-0.1.pth*, abrir o navegar mediante una consola común o *PowerShell* y ejecutar:

*docker cp modelo-0.1.pth*  
*nombreContenedor:home/appuser/detectron2\_repo*

Donde *nombreContenedor* es el nombre que se le haya dado al contenedor creado.

Prestar atención a los espacios en las rutas ya que podría ser necesario el uso de comillas en el argumento del comando



---

## Bibliografía

---

- [1] Detectron2. Installation. <https://detectron2.readthedocs.io/en/latest/tutorials/install.html>, 2020.
- [2] Max Ferguson. Detection and segmentation of manufacturing defects with convolutional neural networks and transfer learning. <https://github.com/maxkferg/metal-defect-detection>, 2016.
- [3] Fco Javier Yagüe Izquierdo. Tfg gii-20.04 detección de defectos metálicos mediante deep learning. <https://github.com/fyi0000/TFG-GII-20.04/>, 2021.