20-3-2020

FALLOS EN SUPERFICIES METÁLICAS

Documento 2: DISEÑO DE LA SOLUCIÓN

PATRICIA FERNÁNDEZ FERNÁNDEZ
GARAZI ALFARO GARCÍA

Contenido

1. INTRODUCCIÓN....1
1.1. INTRODUCCIÓN AL DESARROLLO REALIZADO....1
1.2. DESCRIPCIÓN DEL FUNCIONAMIENTO DE LOS TEST UNITARIOS....1
2. DISEÑO DE LA SOLUCIÓN....4
2.1. INTRODUCCIÓN....4
2.2. DIAGRAMA UML ESTÁTICO DE CLASES....4
2.3. DIAGRAMA DE SECUENCIA....8
2.4. DESCRIPCIÓN DE LOS PRINCIPALES ALGORITMOS DESARROLLADOS....10
3. REFERENCIAS BIBLIOGRÁFICAS....14

1. INTRODUCCIÓN

1.1. INTRODUCCIÓN AL DESARROLLO REALIZADO

En el siguiente documento se podrá visualizar el desarrollo realizado para detectar fallos en imágenes de superficies metálicas. Tras ser detectados se deberán clasificar y localizar dichos fallos.

Para este objetivo se decide utilizar un algoritmo basado en un tipo de red neuronal convolucional muy conocida como "YOLO: Real-Time Object Detection". Se utiliza este tipo de redes porque con un único algoritmo es capaz de realizar la clasificación y detección de los defectos en un tiempo bastante corto, por eso se dice que es en "tiempo real". Hacer la detección en tiempo real era otro de los requisitos más importantes de este proyecto, ya que formará parte de una cadena de producción en una fábrica. La versión 3 de YOLO es una de las más rápidas, y será la que se utilizará el este trabajo.

Por lo tanto, se decide poner en marcha un desarrollo Python en el que se incluirán todas las clases y todos los métodos necesarios para completar un algoritmo completo que, dada una imagen, de la base de datos disponible, detecte todos los defectos que en ella se encuentren clasificándolos, y al mismo tiempo localizándolos, en tiempo real.

1.2. DESCRIPCIÓN DEL FUNCIONAMIENTO DE LOS TEST UNITARIOS

Tomando como ejemplo, uno de los posibles casos más importantes para el cliente, el defecto "crazing", se desarrolla un test unitario para comprobar que el resultado del detector es correcto. En la *Figura 1,* se puede ver el código correspondiente, que utiliza el atributo "assertEquals" de *self* para comparar el valor de los parámetros de la salida obtenida con los valores reales. Pero antes de esto, se realiza un parseo del archivo.xml que nos da como resultado el detector. En la misma figura, a la derecha, se muestran los comandos necesarios para poner en marcha el test unitario.

testunitario.py > ...

You, 4 hours ago | 1 author (You)

```python
class TestImage (unittest.TestCase):
    def test_crazing(self):
        '''
            Test detection a crazing.
        '''
        #detector = DefectDetector()
        #result = detector.detector(imagen)
        tree = ElementTree.parse('./NEU-DET/ANNOTATIONS/crazing_1.xml')
        # tree = ElementTree.parse('./NEU-DET/ANNOTATIONS/inclusion_1.xml')
        root = tree.getroot()
        folder = root.find('folder')
        self.assertEquals(folder.text,'cr')
        w = root.find('size').find('width')
        h = root.find('size').find('height')
        d = root.find('size').find('depth')
        self.assertEqual(int(w.text),200)
        self.assertEqual(int(h.text),200)
        self.assertEqual(int(d.text),1)
        for obj in root.findall('.//object'):
            name = obj.find('name').text
            self.assertEqual(name,'crazing')
            pose = obj.find('pose').text
            self.assertEqual(pose,'Unspecified')
```

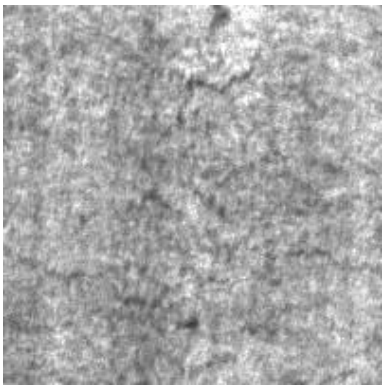
```python
from testunitario import TestImage as test

if __name__ == '__main__':
    tester = test()

    test_cr = tester.test_crazing()
```

Figura 1

Además, en la *Figura 2*, se puede ver el contenido del archivo.xml obtenido junto con la imagen que lo genera. Si dicho resultado es correcto la salida producida por el test unitario será “None” y dejará que el resto del desarrollo se continúe ejecutando. Mientras que, si el resultado es erróneo, la salida dará un “Assertion Error” identificando qué parámetro no es correcto.

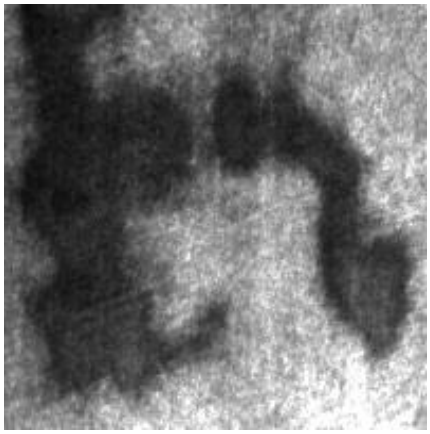


DET > ANNOTATIONS > crazing_13.xml

```
<annotation>
    <folder>cr</folder>
    <filename>crazing_13.jpg</filename>
    <source>
        <database>NEU-DET</database>
    </source>
    <size>
        <width>200</width>
        <height>200</height>
        <depth>1</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>crazing</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>71</xmin>
            <ymin>21</ymin>
            <xmax>140</xmax>
            <ymax>114</ymax>
        </bndbox>
    </object>
    <object>
        <name>crazing</name>
        <pose>Unspecified</pose>
        <truncated>1</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>1</xmin>
            <ymin>126</ymin>
            <xmax>89</xmax>
            <ymax>176</ymax>
        </bndbox>
    </object>
    <object>
        <name>crazing</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>124</xmin>
            <ymin>117</ymin>
            <xmax>199</xmax>
            <ymax>173</ymax>
        </bndbox>
    </object>
</annotation>
```

Figura 2

De igual manera que con el ejemplo anterior, cuando se tiene una superficie metálica en la que se detecten manchas (patches), se deberá obtener cómo resultado el código mostrado en la figura 3 junto a la imagen seleccionada.



NEU-DET > ANNOTATIONS > patches_124.xml

```xml
<annotation>
    <folder>pa</folder>
    <filename>patches_124.jpg</filename>
    <source>
        <database>NEU-DET</database>
    </source>
    <size>
        <width>200</width>
        <height>200</height>
        <depth>1</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>patches</name>
        <pose>Unspecified</pose>
        <truncated>1</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>3</xmin>
            <ymin>1</ymin>
            <xmax>86</xmax>
            <ymax>178</ymax>
        </bndbox>
    </object>
    <object>
        <name>patches</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>96</xmin>
            <ymin>28</ymin>
            <xmax>194</xmax>
            <ymax>167</ymax>
        </bndbox>
    </object>
</annotation>
```

Figura 3

Siguiendo el código mostrado en la figura 4, si ocurre uno de los siguientes casos:

- su campo “folder” no es “pa”
- sus dimensiones no son (200,200,1)
- si el “name”objeto detectado no es “patches”

El terminal de ejecución mostrará un “Assertion Error” en el que se especificará el elemento incorrecto, y el que detendrá la ejecución del algoritmo de detección.

```python
def test_patches(self):
    '''
        Test detection a patches.
    '''
    #detector = DefectDetector()
    #result = detector.detector(imagen)
    tree = ElementTree.parse('./NEU-DET/ANNOTATIONS/patches_1.xml')
    # tree = ElementTree.parse('./NEU-DET/ANNOTATIONS/crazing_1.xml')
    root = tree.getroot()
    folder = root.find('folder')
    self.assertEquals(folder.text,'pa')
    w = root.find('size').find('width')
    h = root.find('size').find('height')
    d = root.find('size').find('depth')
    self.assertEqual(int(w.text),200)
    self.assertEqual(int(h.text),200)
    self.assertEqual(int(d.text),1)
    for obj in root.findall('.//object'):
        name = obj.find('name').text
        self.assertEqual(name,'patches')
        pose = obj.find('pose').text
        self.assertEqual(pose,'Unspecified')
```

Figura 4

2. DISEÑO DE LA SOLUCIÓN

2.1. INTRODUCCIÓN

Esta sección está orientada a ofrecer una visión global de la etapa del diseño, una segunda fase sustancial en el desarrollo del proyecto.

Se hará referencia a aspectos como el funcionamiento del sistema informático que se reflejará con un diagrama de secuencia, una descripción de las diferentes clases que forma el software a través de un diagrama UML estático de clases o un diagrama de actividad detallado a fin de comprender la operatividad de los algoritmos principales desarrollados.

2.2. DIAGRAMA UML ESTÁTICO DE CLASES

A través de este apartado, se especificará las diferentes clases utilizadas en el software, sus atributos, operaciones y las relaciones entre estas. A fin de visualizar la interacción de clases de una manera más representativa, se puede observar el Diagrama 1.

En total se emplean 4 clases que se describirán en *Tabla 1,* a continuación:

Tabla 1: Descripción de las clases del sistema.

CLASE	DESCRIPCIÓN	ATRIBUTOS	OPERACIONES	RELACIONES ENTRE CLASES
Metal_detector_defect	Es la clase principal del sistema, se encarga de agrupar todas las demás clases para su correcto funcionamiento.	*Pathdb: str* Es la dirección de la base de datos de las imágenes. *pathoutput: str* Es la dirección en donde se va a guardar el fichero XML de salida.	*Detection()* Se le introduce la imagen de la base de datos y como resultado obtenemos el fichero XML.	-
Initialize	Toma cada imagen de la base de datos y le aplica su respectivo preprocesado antes de su introducción a la red.	*pathdb: str* Es la dirección de la base de datos de las imágenes. *dirsImage: img*	*loadimage()* Carga la imagen de la base de datos. *preprocesing(file)* Le aplica técnicas de tratamiento digital a la imagen.	**Initialize** está asociada a la clase **Metal_detector_defect.**
Defect	Obtiene la imagen preprocesada de la clase "Initialize" y procede a introducirla en la red neuronal Yolo. Como resultado se dispone un fichero con formato XML, en el cual indica la clase del defecto, la ubicación del bbox además de la fiabilidad.	*tag: str* La etiqueta del defecto, en total habrá 6. *fidelity: float* La fidelidad de la predicción de la red, de 0 a 1. *location: matrix*	*defectdetector(image)* Introduce la imagen a la red neuronal Yolo.	**Defect** está asociada a la clase **Metal_detector_defect.**

		Ubicación del bbox del defecto en la imagen, corresponde a 4 variables. *model: object* Activa el modelo de la red neuronal Yolo.		
TestImage	Con el objetivo de que todas las operaciones del sistema devuelvan resultados esperados y lógicos, se ha creado la clase "TestImage". Por el momento, verifica que la red neuronal devuelve los datos coherentes tras su ejecución.	-	*test_patches()* Verifica que el resultado del fichero XML de la clase predicha "Patches" es coherente. *test_inclusion()* Verifica que el resultado del fichero XML de la clase predicha "Inclusion" es coherente. *test_scratches()* Verifica que el resultado del fichero XML de la clase predicha "Scratches" es coherente. *test_crazing()* Verifica que el resultado del fichero XML de la clase predicha "Crazing" es coherente. *test_pitted_surface()* Verifica que el resultado del fichero XML de la clase predicha "The pitted surface" es coherente. *test_rolled_in_scale()* Verifica que el resultado del fichero XML de la clase predicha "Rolled in scale" es coherente.	**TestImage** usa la clase **Metal_detector_defect.**

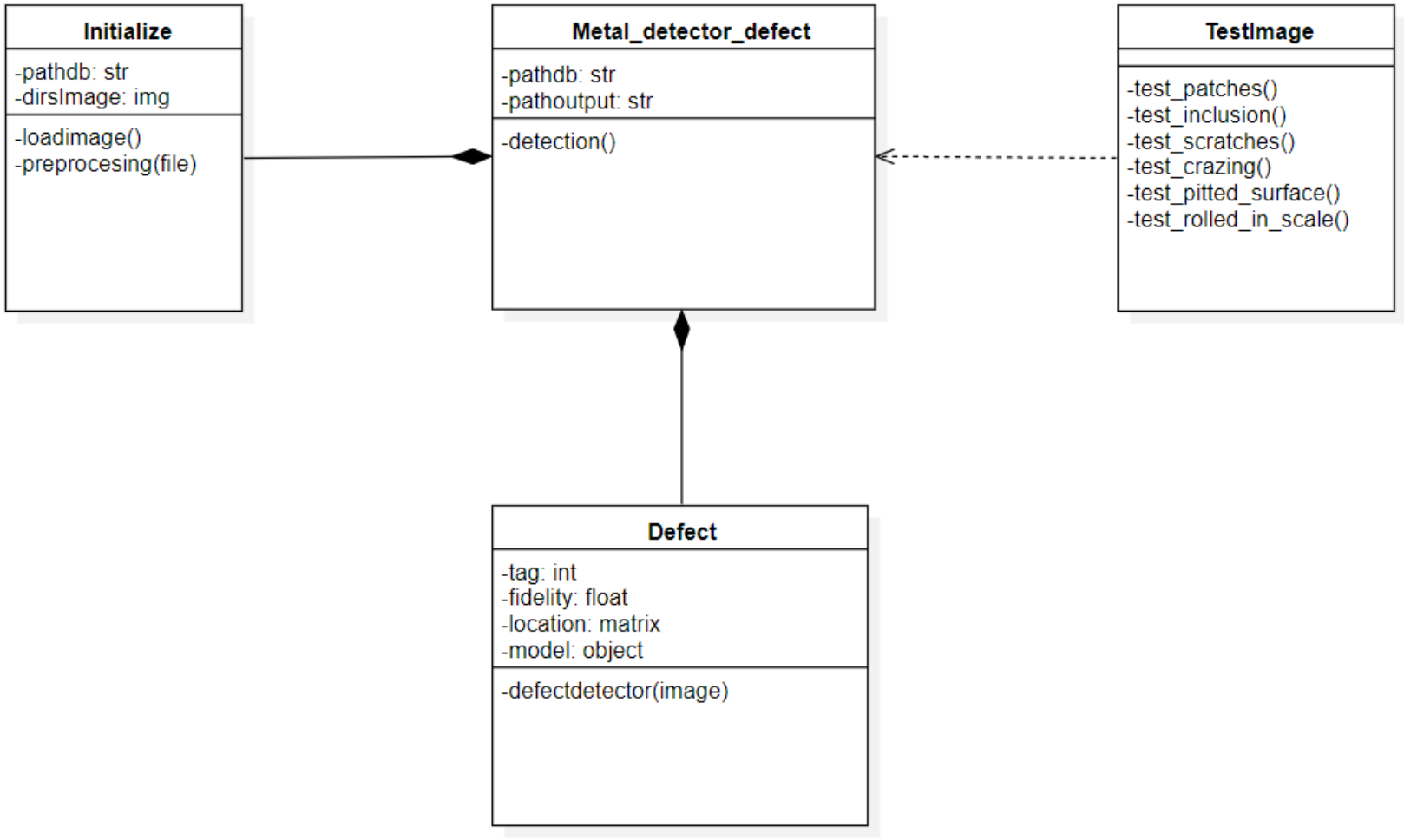


Diagrama 1: Diagrama de clases.

2.3. DIAGRAMA DE SECUENCIA

El diagrama de secuencia muestra la interacción de un grupo de objetos cuando funcionan en conjunto o en el momento que completan un proceso, Diagrama 2.

Se ha formado el diagrama de secuencia con 4 objetos:

- **Directory:** Corresponde a la base de datos almacenada en el ordenador.
- **Defect:** Es una clase que se ocupa de detectar defectos en las imágenes de las piezas metálicas.
- **Yolo Network:** Este objeto corresponde a una red de detección que se le adjunta el nombre de Yolo, predice la clase de imperfección a la que forma parte una región y el *bbox*[1] del objeto completo al que pertenece.
- **Image Pre-processing:** Se relaciona con el preprocesado oportuno que se realiza en la imagen antes de introducirla a la red.

En cuanto al funcionamiento del sistema, en primer lugar, se carga el modelo que en nuestro caso se implementa como una red neuronal convolucional nombrada como Yolo.

En seguida el sistema irá accediendo a la base de datos a por imágenes, se ira introduciendo de una en una, estas soportaran técnicas de tratamiento digital de imágenes como preprocesado.

A continuación, la imagen ya preprocesada se dirigirá a la clase "Defect", donde hará uso de la red neuronal Yolo, con el propósito de detectar las regiones con imperfecciones, identificar su clase correspondiente, la confianza en dicha predicción y la integración en un *bbox*. Como resultado la red retorna un archivo XML con la clase del defecto, ubicación del *bbox* y la fiabilidad.

[1] *Bbox: Booking box en inglés, se define como un rectángulo hueco que sirve para indicar una región específica en una imagen.*

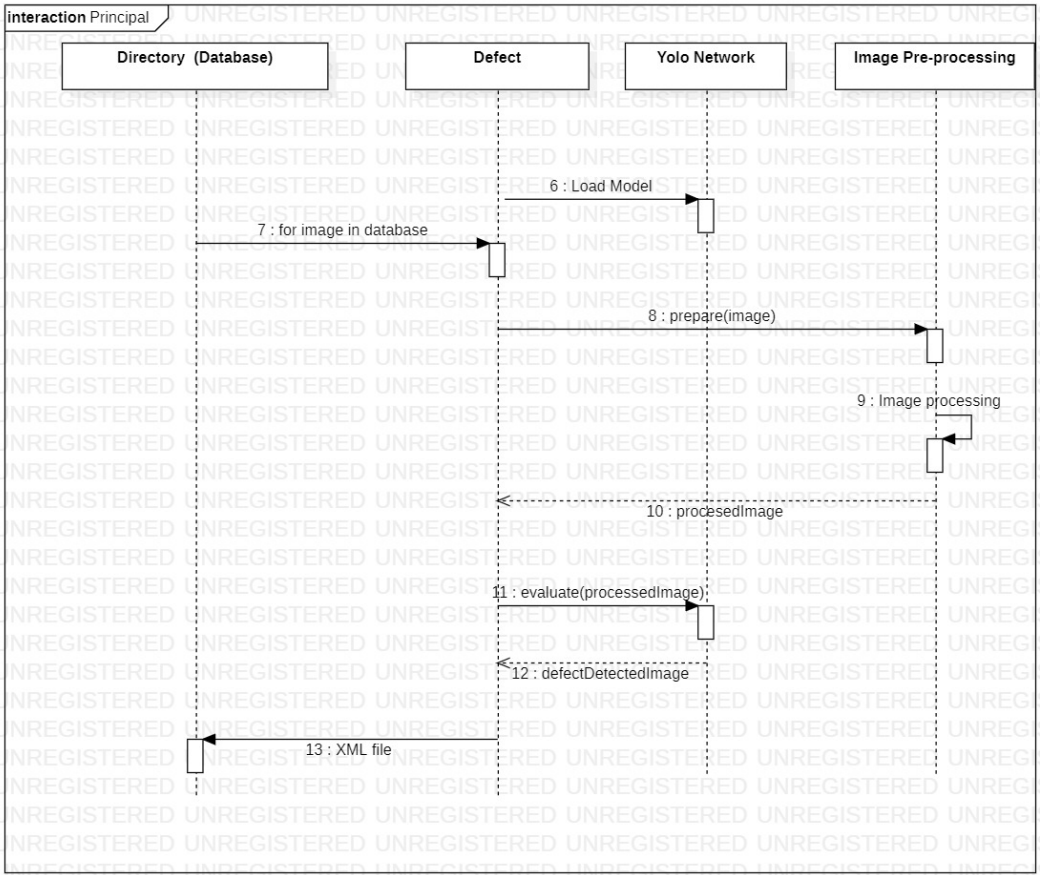


Diagrama 2: Diagrama secuencial. Nota: por temas de credenciales no se ha podido guardar la imagen sin "Unregistered" de fondo.

2.4. DESCRIPCIÓN DE LOS PRINCIPALES ALGORITMOS DESARROLLADOS

Desde una visión global al *software* del sistema, es conveniente realizar un diagrama de actividad completo, Diagrama 3.

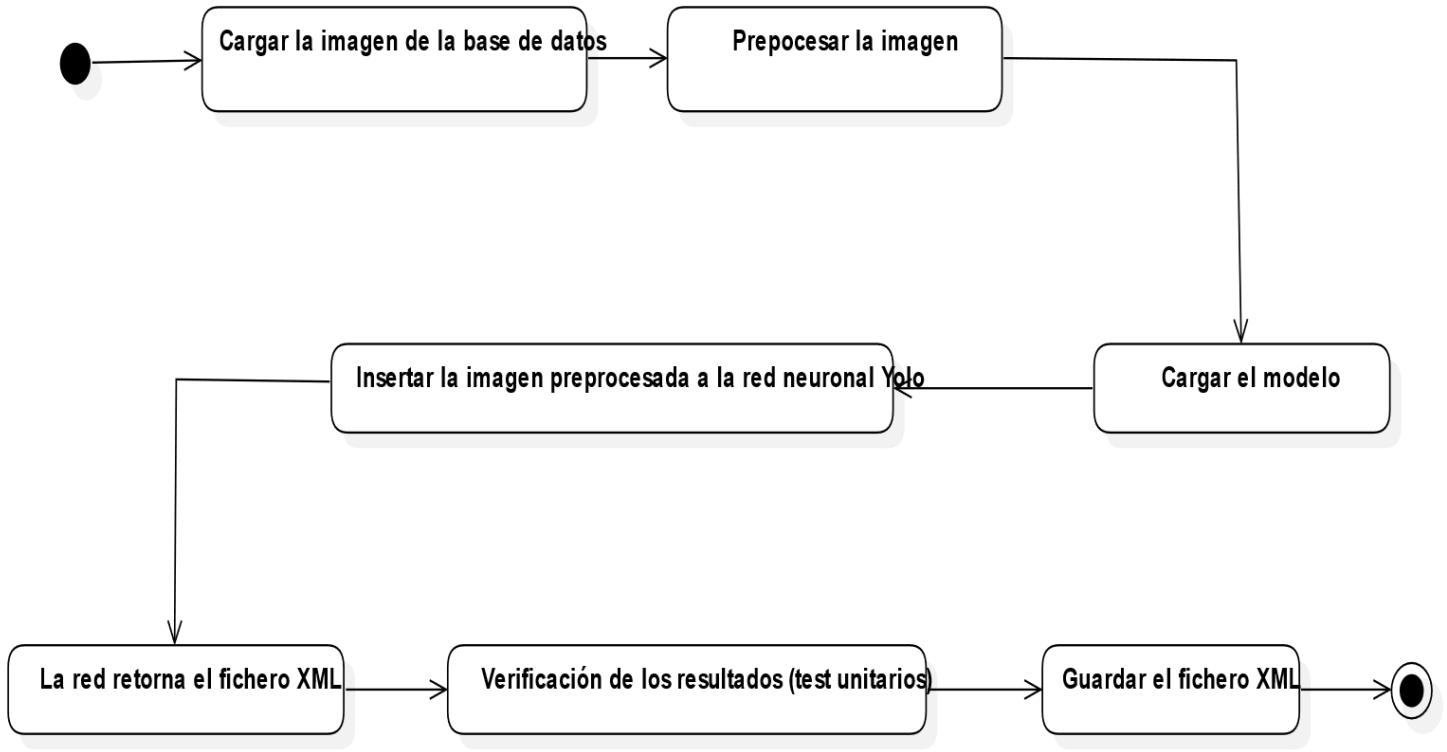


Diagrama 3: Diagrama de actividad global.

Como *imput* se extraen las imágenes de las piezas metálicas de la base de datos, se puede observar algunos ejemplos de imágenes de la base de datos como la *Figura 5* y *Figura 6*, a continuación, se lleva a cabo un preprocesado que se especificara con más detalle en un diagrama de actividad, Diagrama 4.

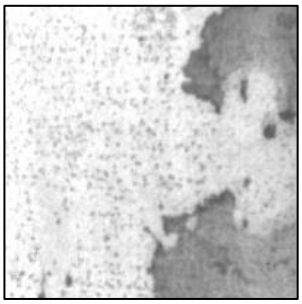


Figura 6: Pieza metálica con la imperfección "Pitted surface".

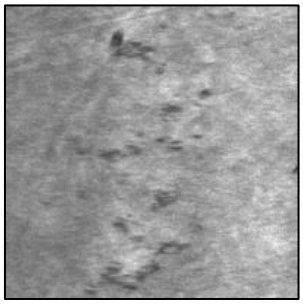


Figura 5: Pieza metálica con la imperfección "Rolled in scale".

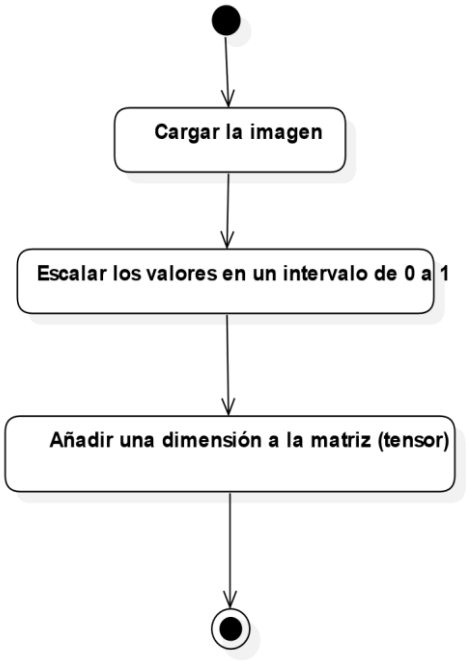


Diagrama 4: Diagrama de actividad del preprocesado.

Seguidamente, tras prepocesar la imagen se introduce a la red neuronal Yolo, la cual devuelve un fichero XML con la clase, fidelidad y la ubicación del *bbox* en la imagen. Se verifica que el resultado es coherente y su estructura es la correcta a través de los test unitarios. Por último, se guarda el archivo en la dirección *output* asignada.

Para la detección de defectos en las piezas metálicas se ha apostado por un modelo de red neuronal convolucional llamada Yolo (*You Only Look Once*). Para este proyecto es interesante su uso debido a que realiza tareas de detección en tiempo real, ajustándose perfectamente a las necesidades del sistema.

Lo primero que realiza Yolo es dividir la imagen de entrada en celdas de 7x7. Para cada celda se aplican tareas de regresión para predecir la ubicación del *bbox* dentro de la imagen (*Figura 8),* de igual manera, la red produce una distribución de masa de probabilidad sobre todas las clases posibles, asignando la clase con mayor probabilidad (*Figura 9*). Dado que cada celda produce un vector, tomando toda la imagen al completo se genera un tensor de salida (*Figura 10),* la cual contendría las diferentes propuestas junto la confianza de cada predicción además de la distribución de clases de cada recorte.

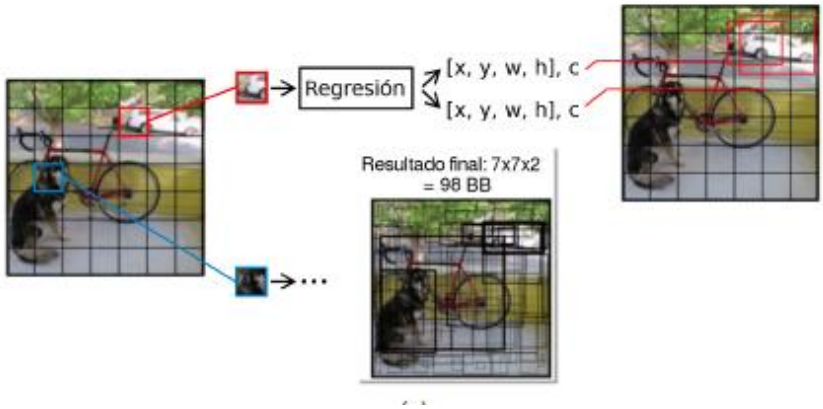


Figura 7: Red de regresión Yolo.

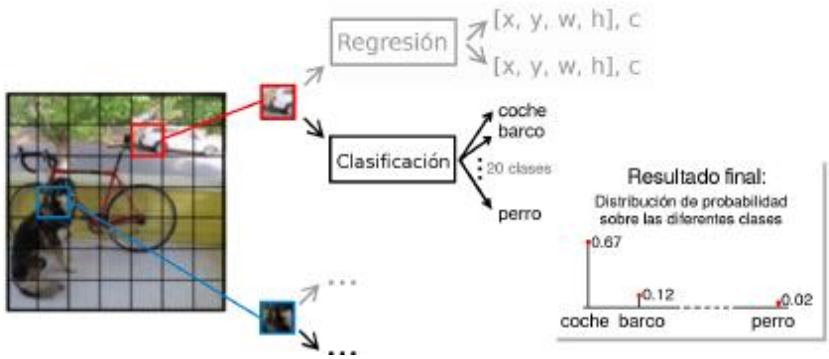


Figura 9: Red de clasificación Yolo.

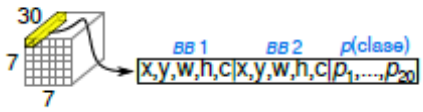


Figura 10: Tensor de salida Yolo.

Como resultado final de la red profunda de detección Yolo, dispondríamos de una imagen con los bbox dibujados a cada clase con mayor probabilidad asignada (*Figura 11*).

Figura11: Resultado de Yolo.

Por último, a parte de la imagen se obtendrá un fichero XML con los resultados de la red neuronal adjuntos, tal y como se aprecia en la *Figura 12*. En “name” indica la clase del defecto, “size” las dimensiones de la imagen y “bndbox” la ubicación del bbox.

```
<annotation>
        <folder>rs</folder>
        <filename>rolled-in_scale_29.jpg</filename>
        <source>
                 <database>NEU-DET</database>
        </source>
        <size>
                <width>200</width>
                <height>200</height>
                <depth>1</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>rolled-in_scale</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>36</xmin>
                        <ymin>74</ymin>
                        <xmax>72</xmax>
                        <ymax>123</ymax>
                </bndbox>
        </object>
        <object>
                <name>rolled-in_scale</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>90</xmin>
                        <ymin>134</ymin>
                        <xmax>131</xmax>
                        <ymax>163</ymax>
                </bndbox>
        </object>
        <object>
                <name>rolled-in_scale</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
```

Figura 12: Estructura del fichero XML.

3. REFERENCIAS BIBLIOGRÁFICAS

- https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/
- https://machinelearningmastery.com/how-to-train-an-object-detection-model-with-keras/
- https://machinelearningmastery.com/how-to-load-large-datasets-from-directories-for-deep-learning-with-keras/
- Apuntes de la asignatura “Reconocimiento de Patrones”, Alfredo Cuesta Infante.