GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Procesamiento automático de Origami de ADN usando técnicas de inteligencia artificial

Autora: Delia Clar Ramírez
Tutor 1: David Contreras Bárcena
Tutor 2: Mario Castro Ponce

Madrid

UNIVERSIDAD PONTIFICIA
ICAI ICADE
COMILLAS
M A D R I D

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

ADN ORIGAMI

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2019/20 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Delia Clar Ramírez Fecha: 11/06/2020

Autorizada la entrega del proyecto

CONTRERAS BARCENA DAVID - 53007209Y

Firmado digitalmente por CONTRERAS BARCENA DAVID - 53007209Y
Fecha: 2020.07.11 09:58:21 +02'00'

Fdo.: David Contreras Bárcena Fecha: 11/07/2020

Mario Castro Ponce

UNIVERSIDAD PONTIFICIA
ICAI ICADE
COMILLAS
M A D R I D

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Procesamiento automático de Origami de ADN usando técnicas de inteligencia artificial

Autora: Delia Clar Ramírez
Tutor 1: David Contreras Bárcena
Tutor 2: Mario Castro Ponce

Madrid

UNIVERSIDAD PONTIFICIA
ICAI ICADE
COMILLAS
M A D R I D

PROCESAMIENTO AUTOMÁTICO DE ORIGAMI DE ADN USANDO TÉCNICAS DE INTELIGENCIA ARTIFICIAL

Autor: Clar Ramírez, Delia
Directores: Contreras Bárcena, David y Castro Ponce, Mario.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

En el presente proyecto se desarrollarán modelos de Machine Learning basados en algoritmos de detección de objetos. El objetivo será, dada una imagen que contenga moléculas de ADN transformadas ("ADN Origami"), lograr detectar las posiciones dentro de ésta. Se compararán los resultados y se seleccionará aquel que ofrezca la mejor solución.

Palabras clave: Machine learning, ADN origami, detección de objetos

1. Introducción

El ADN origami es una tecnología química capaz de plegar las moléculas de ADN para convertirlas en multitud de estructuras diferentes. Se propondrá la creación de un software capaz de detectar en una imagen las posiciones en las que se encuentran estas figuras de ADN. En este caso tomarán la forma de triángulos.

2. Definición del Proyecto

Para dar solución al problema es necesaria la creación de una red neuronal convolucional, conocida como el motor del Machine Learning. Su funcionalidad será ser capaz de aprender por sí misma, por lo que será necesario aplicar una fase de entrenamiento.

Para que una red neuronal sea capaz de aprender, es necesario obtener unos datos de entrada, (las imágenes), junto con sus respectivos resultados, (las posiciones de los triángulos). Una red neuronal aplica diferentes técnicas a los datos para transformarlas en la solución esperada. En el proceso de entrenamiento se comparan las soluciones obtenidas con las reales. Así, se modificarán las distintas transformaciones hasta minimizar la diferencia entre ambas.

Se desarrollarán distintas técnicas con el objetivo fundamental es encontrar el modelo que detecte el mayor número posible de moléculas, dentro de cada imagen.

3. Descripción del modelo

3.1. Modelo de Clasificación

En primer lugar, se desarrollará una red neuronal de clasificación. Un modelo capaz de detectar si en una imagen existe o no un objeto definido. Sin embargo, esta red no detecta la posición en la que se encuentra.

A través de los algoritmos de propuestas de regiones, Selective Search [1], o una ventana deslizante que recorra la imagen, se propondrán posibles posiciones dentro de la figura, las cuales se procesarán por el modelo. Si el resultado, en porcentaje, supera un determinado umbral (0.9) se almacenarán las posiciones.

Una vez implementado por este algoritmo se procederá a utilizar el algoritmo de Non-Maximum Supression [2], utilizado para que un mismo objeto no esté rodeado por más de una región. El proceso de esta técnica se muestra en la siguiente ilustración 1.

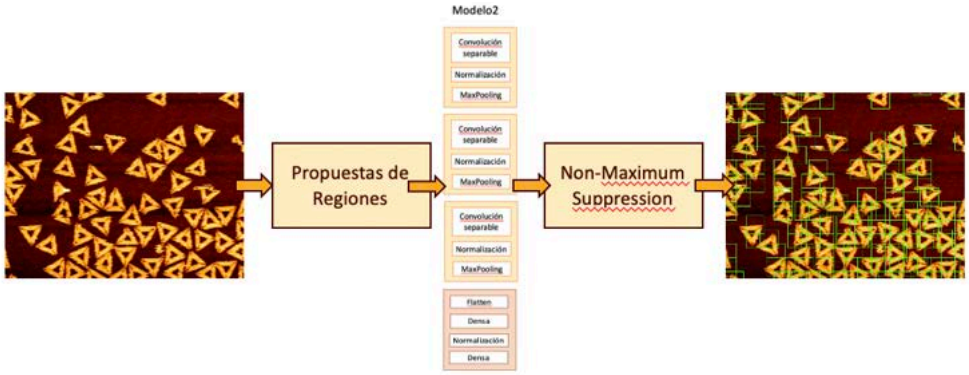


Ilustración 1. Proceso de detección de triángulos

3.2. Reentrenamiento de Faster R-CNN

Otra técnica desarrollada es el reentrenamiento de un modelo ya creado, Faster R-CNN [3]. Para adaptarlo al problema será necesaria la modificación de ciertos parámetros de la configuración del modelo.

La diferencia de este modelo con respecto al anterior es que durante el proceso de entrenamiento lleva a cabo la predicción de las coordenadas de las regiones que rodean al objeto. Es decir, no es necesaria la implementación de otros algoritmos. La arquitectura de este modelo junto con un ejemplo de salida se muestra en la siguiente ilustración 2.

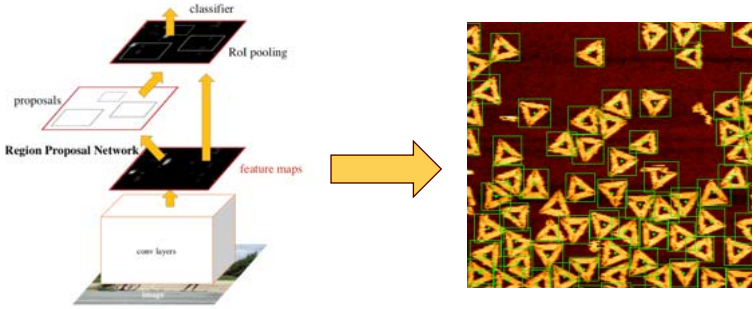


Ilustración 2. Arquitectura e implementación del modelo Faster R-CNN

3.3. Faster R-CNN + Modelo de Clasificación

Los resultados del modelo de Faster R-CNN son las coordenadas de las cajas delimitadoras con posibles moléculas de ADN (en forma de triángulo), en su interior, junto con las probabilidades de que realmente lo sean.

Otra manera de dar solución al problema es utilizar las regiones dadas por el modelo reentrenado y utilizar el clasificador implementado en el primer apartado, tal y como se muestra en la siguiente ilustración 3.

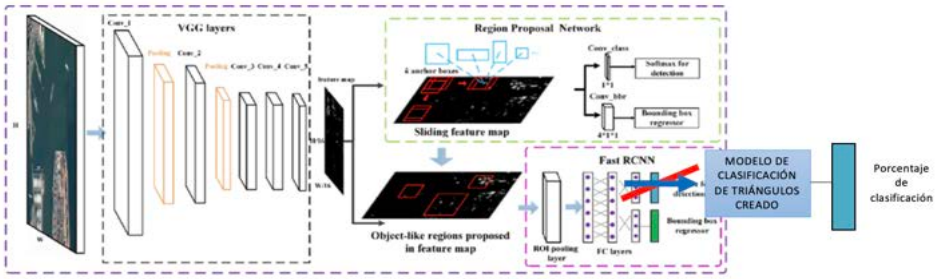


Ilustración 3. Arquitectura del conjunto de modelos

4. Resultados

Para calcular los resultados de dichos modelos se hace uso de lo que se denomina Precisión media. En primer lugar, se calcula la precisión, que mide sobre todos los resultados reales cuántos se han acertado. Seguidamente, el recall, que mide sobre todos los resultados predichos cuantos se han acertado.

Finalmente, la precisión media (AP) es el área de la curva que relaciona la precisión y el recall. Dichos resultados se muestran, para cada modelo, en la tabla 1.

Modelo	**AP (%)**	**Recall (%)**
Clasificación + Algoritmos	10.5	25
Reentrenamiento Faster R-CNN	85.3	89
Faster R-CNN + Clasificación	70.3	74.4

Tabla 1. Resultados obtenidos de todos los modelos

En siguiente ilustración 4 se muestra un ejemplo de la aplicación de cada modelo:

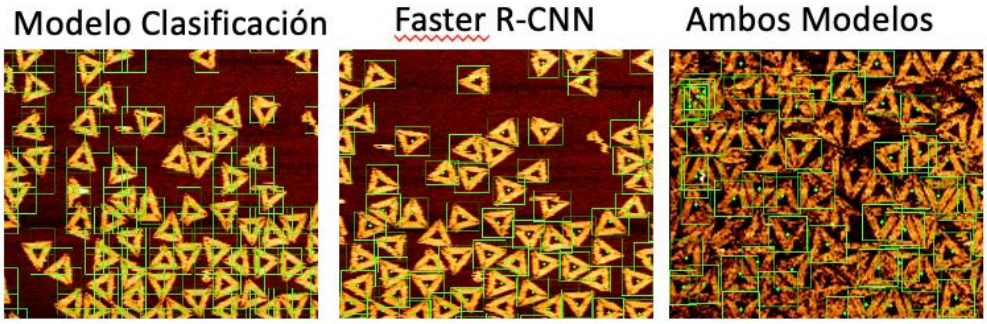


Ilustración 4. Aplicación de cada modelo a una imagen

5. Conclusiones

Tal y como se ha podido comprobar, Faster R-CNN es el mejor detector de objetos. Esto es debido a que este modelo ya ha sido entrenado y probado para distintos tipos de clase.

Además, se observa como los fallos producidos por el modelo surgen de la primera fase, en la cual se proponen posibles regiones con moléculas en su interior. Error que arrastra el conjunto de ambos modelos.

Otro error producido por el modelo es la superposición de regiones, en la cual se muestra como distintos cuadrados rodean al mismo triángulo.

6. Referencias

[1] J.R.R.R. Uijlings, K.E.A. van de Sande, T. Gevers and A.W.M. Smeulders - Selective Search. [Available: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf]

[2] J. Hosang, R.Benenson,-NMS. [Available: https://arxiv.org/pdf/1705.02950.pdf]

[3] S. Ren, K. He, R. Girshick , J. Sun Faster R-CNN [https://arxiv.org/pdf/1506.01497.pdf]

AUTOMATIC DNA ORIGAMI PROCESSING USING ARTIFICIAL INTELLIGENCE TECHNIQUES

Author: Clar Ramírez, Delia
Supervisors: Contreras Bárcena, David y Castro Ponce, Mario.
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

In this present Project, Machine Learning models based on object detection algorithms will be developed. The objective will be, given an image containing transformed DNA molecules ("Origami DNA"), to be able to detect the positions within it. The results will be compared and the one that offers the best solution will be selected.

Key words: Machine Learning, Object detection, Origami DNA

1. Introduction

Origami DNA is a chemical technology capable of folding DNA molecules into a multitude of different structures. In this project it will be proposed the creation of software capable of detecting in an image the positions in which these DNA figures, in this case they will take the form of triangles.

2. Proyect Definition

The solution to the problem requires the creation of a convolutional neural network, known as the engine of Machine Learning. Its functionality will be to be able to learn by itself, so it will be necessary to apply a training phase.

For a neuronal network to be able to learn, it is necessary to obtain some input data, (the images), along with their respective results, (the positions of the triangles). A neural network applies different techniques to the data to transform them into the expected solution. In the training process, the solutions obtained are compared with the real ones. Thus, the different transformations will be modified until the difference between both is minimized.

Different techniques will be developed with the fundamental objective of finding the model that detects the greatest possible number of molecules within each image.

3. Model Description

3.1. Classification Model

To begin with, a neural network of classification will be developed. A model capable of detecting whether or not a defined object exists in an image. However, this network does not detect the position in which it is located.

Using the region proposal algorithms, Selective Search [1], or a sliding window that runs through the image, possible positions within the figure will be proposed and processed by the model. If the result, in percentage, exceeds a certain threshold (0.9), the positions will be stored.

Once implemented, the Non-Maximum Suppression algorithm [2] will be applied, used so that the same object is not surrounded by more than one region. The process of this technique is shown in the following illustration 1.

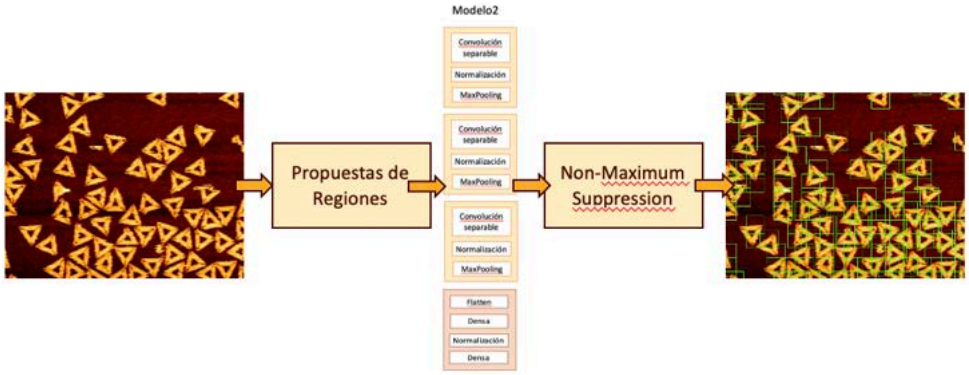
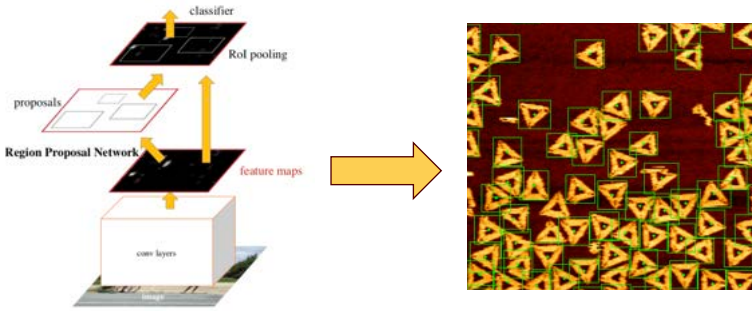


Illustration 1. Triangle detection process

3.2. Faster R-CNN Retraining

Another technique developed is the retraining of an already created model, Faster R-CNN [3]. To adapt it to the problem it will be necessary to modify certain parameters of the model configuration.

The difference of this model with the previous one is that during the training process it carries out the prediction of the coordinates of the regions surrounding the object. That is, it is not necessary to implement other algorithms. The architecture of this model along with an example of output is shown in the following illustration 2.

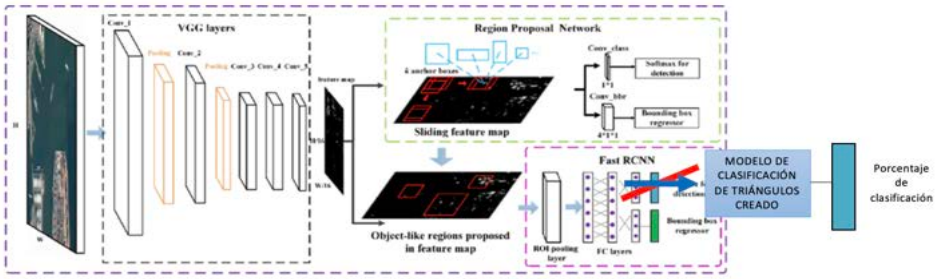


4. **Illustration 2**. Architecture and implementation of the Faster R-CNN model

4.1. Faster R-CNN + Classification Model

The results of the Faster R-CNN model are the coordinates of the boundary boxes with possible DNA molecules (in the form of a triangle), inside them, together with the probabilities that they really are.

Another way to solve the problem is to use the regions given by the retrained model and used the classifier implemented in the first section, as shown in Figure 3 below



5. Illustration 3. Model set architecture

6. Results

The results of these models are calculated using what is called the Average Precision (AP). First, the accuracy is calculated, which measures over all the true results how many have been successful. Then, the recall, which measures how many of the predicted results are correct.

Finally, the AP is the area of the curve that relates precision and recall. These results are shown, for each model, in table 1.

Model	**AP (%)**	**Recall (%)**
Classification + Algorithms	10.5	25
Faster R-CNN Retraining	85.3	89
Faster R-CNN + Classification	70.3	74.4

Table 1. Results obtained from all models

An example of the application of each model is shown in Figure 4 below:

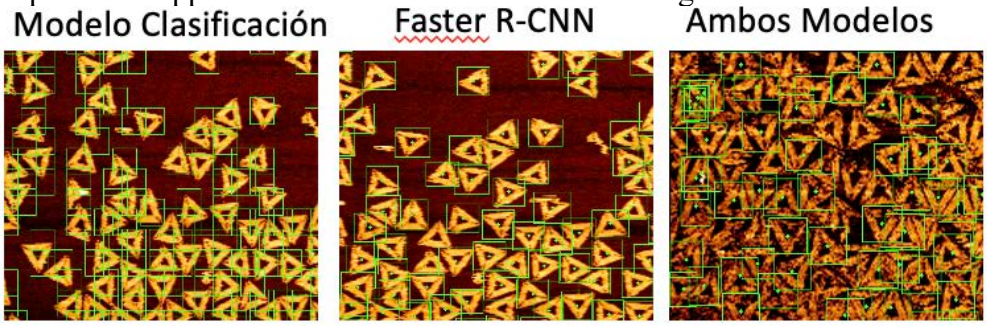


Ilustración 4. Aplicación de cada modelo a una imagen

7. Conclusions

As it has been proven, Faster R-CNN is the best object detector. This is because this model has already been trained and tested for different types of class.

In addition, it can be seen how the failures produced by the model arise from the first phase, in which it proposes regions with molecules inside them. This error drags the whole of both models.

Another error produced by the model is the superposition of regions, in which it is shown how different squares surround the same triangle

8. References

[1] J.R.R.R. Uijlings, K.E.A. van de Sande, T. Gevers and A.W.M. Smeulders - Selective Search. [Available: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf]

[2] J. Hosang, R.Benenson, B. Schiele, M. Plank- Non-Maximum Suppression. [Available: https://arxiv.org/pdf/1705.02950.pdf]

[3] S. Ren, K. He, R. Girshick , J. Sun Faster R-CNN [https://arxiv.org/pdf/1506.01497.pdf]

Índice de la memoria

Capítulo 1. Introducción 6

1.1 Motivación del proyecto 7

1.2 Planteamiento del proyecto 7

Capítulo 2. Descripción de las Tecnologías 9

2.1 RED NEURONAL 9

2.2 Red Neuronal Convolucional CNN 12

2.3 Entrenamiento 15

2.3.1 Sobreajuste y subajuste (Overfitting / Underfitting) 18

2.4 Tipos de Problemas 19

2.5 Herramientas a utilizar 21

Capítulo 3. Estado de la Cuestión 22

3.1 Técnicas para la creación de Redes Neuronales 22

3.1.1 Clasificador de triángulos 25

3.2 Aplicación Red Neuronal Convolucional 29

3.3 Modelos De Detección De Objetos 31

3.3.1 Propuesta de Región (Region proposal) 32

3.3.2 Predicción 32

3.3.3 Non-Maximum Suppression (NMS) 33

3.3.4 Métrica 34

3.4 Uso de OpencV 38

Capítulo 4. Modelo Desarrollado 40

4.1 Etiquetado de Imágenes 40

4.2 Creación de un Modelo De Clasificación 41

4.2.1 Primer modelo 44

4.2.2 Segundo modelo 46

4.2.3 Modelos Restantes 55

4.3 Aplicación de Algoritmos de Detección de Regiones 56

4.4 Reentrenamiento de un Modelo 58

4.5 Faster R-CNN 59

4.5.1 Arquitectura 60

4.5.2 Predicción del Cuadrado Delimitador....63
4.5.3 Función de Pérdida....64
4.5.4 Reentrenamiento....65

Capítulo 5. Análisis de Resultados....69

5.1. Resultados del Clasificador....69
5.2. Resultados. modelo reentrenado Faster R-CNN....71
5.3 Resultados del conjunto de Ambos Modelos....88

Capítulo 6. Conclusiones y Trabajos Futuros....93

6.1 Conclusiones....93
6.2 Trabajos Futuros....95

Capítulo 7. Bibliografía....97

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS....101

ANEXO II: CÓDIGO....103

Índice de tablas

Tabla 5.1. *Resultados del Clasificador* *69*
Tabla 5.2*. Resultados del reentrenamiento de Faster R-CNN* *75*
Tabla 5.3 *Comparación de resultados del modelo Faster R-CNN* *77*
Tabla 5.4 *Porcentaje de los resultados por imagen* *85*
Tabla 5.5 *Resultados del conjunto de ambos modelos* *88*
Tabla 5.6 *Resultados para el modelo de Faster R-CNN junto con el modelo 4* *89*
Tabla 5.7 Resultados del modelo de Faster R-CNN junto con VGG-16 ..... *91*
Tabla 6.1 *Resultados de los modelos desarrollados* *93*
Tabla 6.2*. Comparación de distintos modelos de detección de objetos* *94*
Tabla 6.3*. Comparación de las clases identificadas por el modelo Faster R-CNN* *95*

Índice de figuras

***Figura 1.1**. Imagen ADN origami* 6
***Figura 1.2**. Machine Learning. Nueva manera de Programar [2]* 7
***Figura 2.1.** Esquema de una red neuronal [4]* 9
***Figura 2.2.** Esquema del perceptrón y una neurona [5]* 10
***Figura 2.3.** Función de Activación. Sigmoide [7]* 11
***Figura 2.4** Esquema matemático de una red neuronal [8]* 11
***Figura 2.5** Representación de las muestras de imágenes que entran como datos de entrada a una CNN [9].* 12
***Figura 2.6** Arquitectura de una red convolucional [10]* 13
***Figura 2.7**. Resultado de una imagen tras la aplicación de diversas convoluciones* 14
***Figura 2.8.** Operación Max-pooling [11]* 14
***Figura 2.9** Arquitectura red convolucional [12]* 15
***Figura 2.10**. Función sencilla de pérdida [14]* 16
***Figura 2.11**. Proceso de entrenamiento de la red [15]* 17
***Figura 2.12**. Error sobre los datos de entrenamiento y validación según las iteraciones [16]* 19
***Figura 2.13** Problema clasificación de imagen [17]* 19
***Figura 2.14** Detección de objetos [18]* 20
***Figura 2.15**. Segmentación semántica [19]* 20
***Figura 3.1** Aumento de datos* 23
***Figura 3.2** Proceso de extracción de características [21]* 24
***Figura 3.3** Arquitectura "Fine-tuning" [23]* 25
***Figura 3.4** Ejemplo del conjunto de datos a entrenar* 26
***Figura 3.5** Comparación de distintas arquitecturas de redes neuronales* 27
***Figura 3.6** Gráficas de los distintos modelos* 28
***Figura 3.7** Dos ejemplos de aplicación del algoritmo Selective Search [25]* 30
***Figura 3.8** Representación de la posición de un objeto* 31
***Figura 3.9** Región de interés* 32
***Figura 3.10** Ejemplo de regiones de interés que rodean a la misma imagen* 33
***Figura 3.11** Ejemplo de aplicación de Non-Maximum Suppression* 34
***Figura 3.12** Cálculo de IOU* 35
***Figura 3.13** Cálculo de FP y TP* 36
***Figura 3.14** Ejemplo Precision-Recall Curve [26]* 37
***Figura 3.15** Representación del dominio HSV [28]* 39
***Figura 4.1** Ejemplo de ejecución del programa LabelImg* 41
***Figura 4.2** Datos utilizados para la creación de un modelo de clasificación* 42
***Figura 4.3** Arquitecturas de los distintos modelos* 43
***Figura 4.4** Gráfica Función Sigmoide* 44
***Figura 4.5** Ejemplo de aplicación de Dropout al 50% [16]* 45
***Figura 4.6** Matriz de confusión de los resultados del modelo 1* 45
***Figura 4.7** Convolución separable en profundidad [30]* 46
***Figura 4.8** Histogramas de los pesos del segundo bloque del modelo* 47
***Figura 4.9** Histogramas de los pesos del primer bloque del modelo* 47
***Figura 4.10** Histogramas de los pesos del tercer bloque del modelo* 48
***Figura 4.11** Histogramas de los pesos del cuarto bloque del modelo* 49
***Figura 4.12** Resultados obtenidos tras aplicar el segundo modelo* 51
***Figura 4.13** Función de pérdida en el proceso de entrenamiento* 51
***Figura 4.14** Función de precisión en el proceso de entrenamiento* 52
***Figura 4.15** Patrones de la imagen pasada por los bloques convolucionales* 54

Figura 4.16 *Imágenes pasadas al modelo con los resultados obtenidos 54*
Figura 4.17 *Matrices de confusión de los modelos creados aplicados a imágenes nunca vistas por el modelo 55*
Figura 4.18 *Matrices de confusión de los modelos creados aplicados a todas las imágenes 55*
Figura 4.19 *Aplicación de recorrido del clasificador a través de la imagen 56*
Figura 4.20 *Aplicación del clasificador a través de la imagen más uso de Non-Maximum Suppression 57*
Figura 4.21 *Aplicación del clasificador sobre regiones detectada por el algoritmo Selective Search 58*
Figura 4.22 *Proceso de Faster R-CNN [33] 60*
Figura 4.23 *Region Proposal Network [33] 61*
Figura 4.24 *Arquitectura Faster R-CNN [38] 62*
Figura 4.25 *Gráfica L1 smooth [39] 65*
Figura 4.26 *Ficheros de la configuración del modelo 67*
Figura 4.27*. Resultado de reentrenamiento de Faster R-CNN 68*
Figura 5.1 *Aplicación del modelo de clasificación 70*
Figura 5.2 *Aplicación del modelo de clasificación 70*
Figura 5.3 *Aplicación del modelo de clasificación a una imagen con triángulos más pequeños 71*
Figura 5.4 *Pérdida del clasificador durante el proceso de reentrenamiento de Faster R-CNN 72*
Figura 5.5 *Pérdida del componente RPN durante el proceso de reentrenamiento de Faster R-CNN 72*
Figura 5.6 *Pérdida total durante el proceso de reentrenamiento de Faster R-CNN 73*
Figura 5.7 *Histograma de los pesos del modelo reentrenado Faster R-CNN 74*
Figura 5.8*. Aplicación del modelo de Faster R-CNN 75*
Figura 5.9 *Aplicación del modelo Faster R-CNN 76*
Figura 5.10 *Aplicación del modelo Faster R-CNN 76*
Figura 5.11 *Resultados del entrenamiento de Faster R-CNN original aplicado a los datos PASCAL VOC [33] 77*
Figura 5.12 *Resultados del entrenamiento de Faster R-CNN original aplicado a los datos MS COCO [33] 77*
Figura 5.13. *Resultados reales y predichos por el modelo Faster R-CNN 78*
Figura 5.14 *Resultados reales y predichos por el modelo Faster R-CNN 78*
Figura 5.15 *Resultados reales y predichos por el modelo Faster R-CNN 79*
Figura 5.16 *Resultados reales y predichos por el modelo Faster R-CNN 79*
Figura 5.17 *Resultados reales y predichos por el modelo Faster R-CNN 80*
Figura 5.18 *Resultados reales y predichos por el modelo Faster R-CNN 80*
Figura 5.19 *Resultados reales y predichos por el modelo Faster R-CNN 81*
Figura 5.20 *Resultados reales y predichos por el modelo Faster R-CNN 81*
Figura 5.21 *Resultados reales y predichos por el modelo Faster R-CNN 82*
Figura 5.22 *Resultados reales y predichos por el modelo Faster R-CNN 82*
Figura 5.23 *Resultados reales y predichos por el modelo Faster R-CNN 83*
Figura 5.24 *Resultados reales y predichos por el modelo Faster R-CNN 83*
Figura 5.25 *Resultados reales y predichos por el modelo Faster R-CNN 84*
Figura 5.26 *Resultados reales y predichos por el modelo Faster R-CNN 84*
Figura 5.27 *Resultados reales y predichos por el modelo Faster R-CNN 85*
Figura 5.28 *Faster R-CNN como detector de regiones 86*
Figura 5.29 *Falsos Negativos 87*
Figura 5.30 *Falsos Positivos 87*
Figura 5.31 *Arquitectura de Faster R-CNN junto con un clasificador de triángulos 88*
Figura 5.32 *Aplicación de ambos modelos 89*
Figura 5.33 *Aplicación del modelo 4 junto con Faster R-CNN 90*
Figura 5.34 *Aplicación del modelo 3 junto con Faster R-CNN 90*
Figura 5.35 *Aplicación del modelo 5 junto con Faster R-CNN 91*
Figura 5.36 Aplicación del modelo de Faster R-CNN junto con VGG-16 92

Capítulo 1. INTRODUCCIÓN

Antes de detallar en que se basa el proyecto, se introducirá qué es el ADN origami. El arte del origami es la técnica en la cual, a través de ir plegando un papel, se consigue crear formas. Además, existe una tecnología química para el ADN que da la posibilidad de modificarla y sintetizarla con facilidad. El ADN origami se refiere a la posibilidad de conseguir formas concretas al doblar sus moléculas.

Es decir, el ADN origami es encontrar la forma de plegar las moléculas de esta para originar multitud de estructuras diferentes. Se encontrarán, entonces, distintas imágenes, tal y como se observa en la siguiente figura 1.1, hechas por un microscopio a 500nm. En el cual se observa está técnica en la que convierte las moléculas en triángulos.

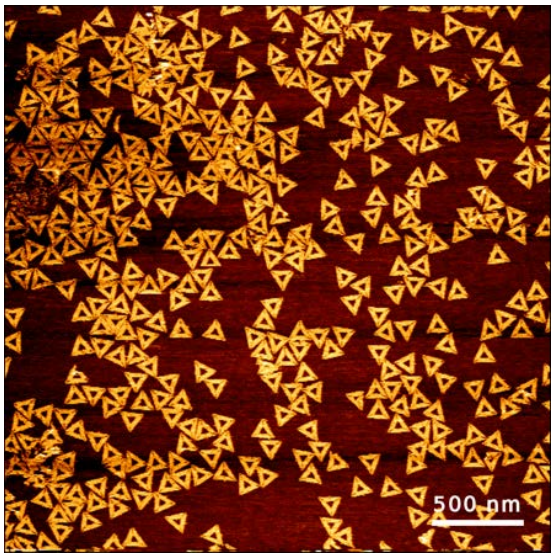


***Figura 1.1**. Imagen ADN origami*

La finalidad del proyecto es detectar estas figuras en dichas imágenes. Para ello se hará uso de lo que se denomina red neuronal. Este término se explicará más adelante. Primero, para entender de donde surge, es necesario definir el término de *Machine Learning*.

Este concepto aparece de la pregunta ¿Puede un ordenador aprender por su cuenta y desarrollar una tarea específica? En lugar de ser el desarrollador el que le indique a la máquina lo que ha de implementar a través de un código ¿Es capaz de aprender automáticamente a través de observar los datos? [1]. En la siguiente figura 1.2, se observan las arquitecturas de la programación clásica y la programación basada en machine learning.

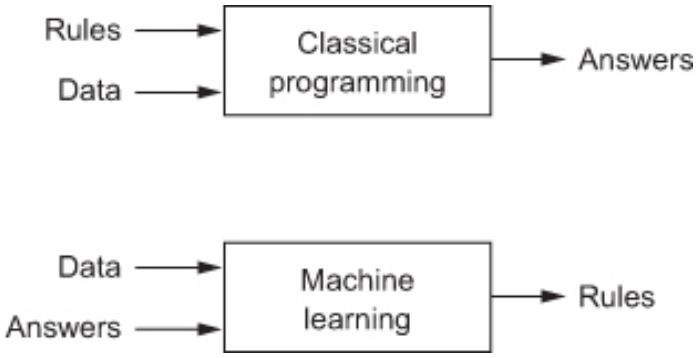


***Figura 1.2**. Machine Learning. Nueva manera de Programar [2]*

1.1 MOTIVACIÓN DEL PROYECTO

Los seres humanos nacen con la habilidad de aprendizaje gracias a las conexiones neuronales, las cuales nos dan la capacidad de razonar y generar conocimientos que nos ayudan a entender y resolver problemas de la vida cotidiana. Las redes neuronales son las responsables que permite al ser humano desarrollar la inteligencia.

La inteligencia artificial aparece como una motivación a imitar el funcionamiento humano en las máquinas, es decir, se busca la forma de darle "inteligencia". Para ello, se hace uso de las redes neuronales, las cuales tratan de reproducir el funcionamiento del cerebro humano. Dicha definición es una manera abstracta de comprender su finalidad. Sin embargo, esto no implica que realicen el comportamiento exacto de las neuronas.[3]

La investigación y el desarrollo de la inteligencia artificial ha progresado de manera exponencial durante los últimos años. Es un concepto de gran aplicación que ha supuesto un gran avance en la tecnología, empezando por el desarrollo de un asistente de voz, como Siri de Apple o Alexa para Amazon, hasta la creación de un coche capaz de funcionar por sí solo.

Para que un coche sea capaz de conducir por si mismo, es necesaria la visión computacional. No solo deberá observar la carretera y detectar las señales. Deberá ser capaz de determinar si hay una persona y la ubicación de esta, ya que el comportamiento que ha de seguir un coche al ver a una persona debe ser distinta si esta está cruzando por la carretera o si está situado en la acera.

1.2 PLANTEAMIENTO DEL PROYECTO

Para comprender mejor la función del proyecto se define el término ADN, el cual, es definido como la molécula donde se encuentra la información genética de cada ser vivo perteneciente a cada uno de ellos. Ésta, tiene una forma de doble hélice, la primera formada por la secuencia de letras químicas ACG [3], donde A pertenece a adenina, C a citosina y G a

guanina, emparejada con la otra hélice la cual contiene la secuencia TGC, donde T pertenece a timina.

Debido a este código las estructuras se unen solas. Existe, además, una tecnología química de la misma que da la posibilidad de modificarla y sintetizarla con facilidad. El ADN origami se refiere a la posibilidad de conseguir formas concretas al doblar las moléculas de ADN.

Para este proyecto será necesario implementar un código en Python que detecte las coordenadas de los triángulos en la imagen. Para ello, se definirán los objetivos que harán posible el desarrollo:

1. Etiquetar los triángulos, es decir, saber en que coordenadas de la imagen se encuentran. Las posiciones de cada figura serán las soluciones del modelo desarrollado

2. Redimensionar todas las imágenes al mismo tamaño. Todos los datos de entrada al modelo desarrollado deberán contener el mismo tamaño.

3. Manipular las imágenes para poder reducir la complejidad de la red. Se buscará la manera de reducir la cantidad de variables de entrada a través de técnicas estadísticos de correlaciones. Además, una posible técnica es pasar del sistema de color de tres dimensiones RGB a una única dimensión.

4. Creación o uso de una red neuronal ya creada, término que se explicará en los próximos apartados. En caso de reutilizar un modelo se podrán hacer modificaciones para adaptarla a las características propias que se buscan.

5. Probar las diferentes imágenes, y comprobar el error. Se deberá minimizar lo máximo posible.

Capítulo 2. DESCRIPCIÓN DE LAS TECNOLOGÍAS

En este apartado se explicarán los términos importantes y las tecnologías que se usarán en el desarrollo del proyecto. Para empezar, para la detección del adn origami se hará uso de una red neuronal convolucional. Para ello explicaremos qué son las redes neuronales y su funcionamiento.

2.1 RED NEURONAL

Una red neuronal es, como ya se ha comentado previamente, el motor del concepto de machine learning. Están inspiradas en el funcionamiento de una neurona. Éstas, reciben un conjunto de valores, lo que se denomina datos de entrada, los cuales atraviesan distintas capas, y se irán modificando hasta llegar a la salida, es decir, a la solución del problema planteado. Esta estructura se puede observar en la figura 2.1.

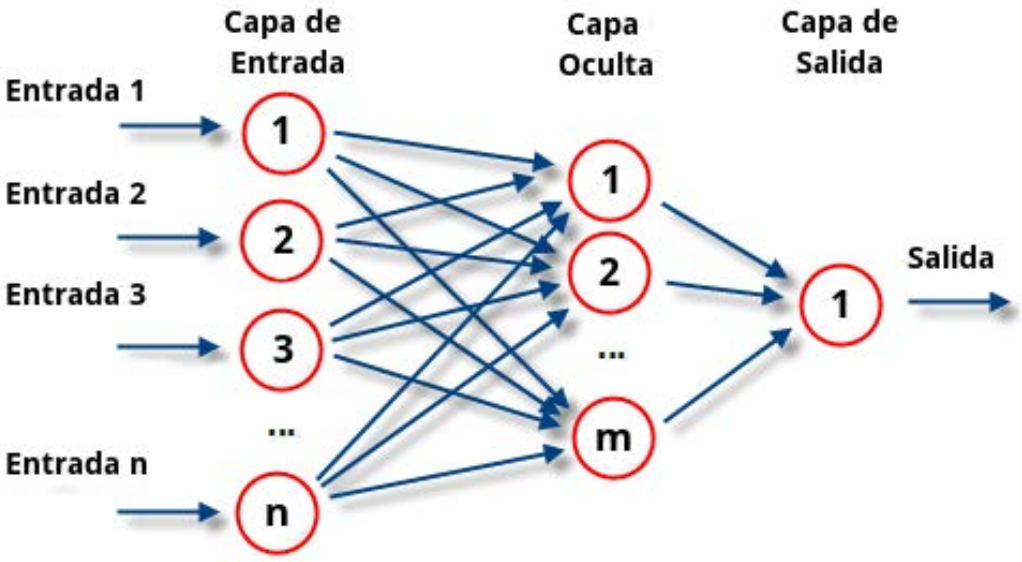


Figura 2.1. *Esquema de una red neuronal [4]*

Como ya se ha explicado, la funcionalidad de este método es que el programa encuentre unas reglas que den solución a un problema. Para ello, no solo es necesario unos datos de entrada, si no que deberá recibir las soluciones de dichos datos.

La red neuronal comparará entonces los resultados obtenidos por las modificaciones con los resultados reales. Con el fin de mejorar sus resultados irá modificándose a sí misma para que estas transformaciones de los datos de entrada lleven a la solución real. Es decir que no haya una gran diferencia entre la predicción de la red y el dato real. Para ello se hará uso de lo que se denomina función de pérdida. Este concepto se explicará más adelante.

Para entender mejor el desarrollo de las redes neuronales, primero se procederá a explicar el tipo de red más sencillo, el cual es conocido como perceptrón. Se observa el esquema en la figura 2.2.

La red neuronal, como ya se ha mencionado, imita le funcionamiento de las neuronas del cerebro. Es por ello por lo que el funcionamiento del perceptrón es similar a la de una única neurona, conforme a la siguiente imagen. (Figura 2.2).

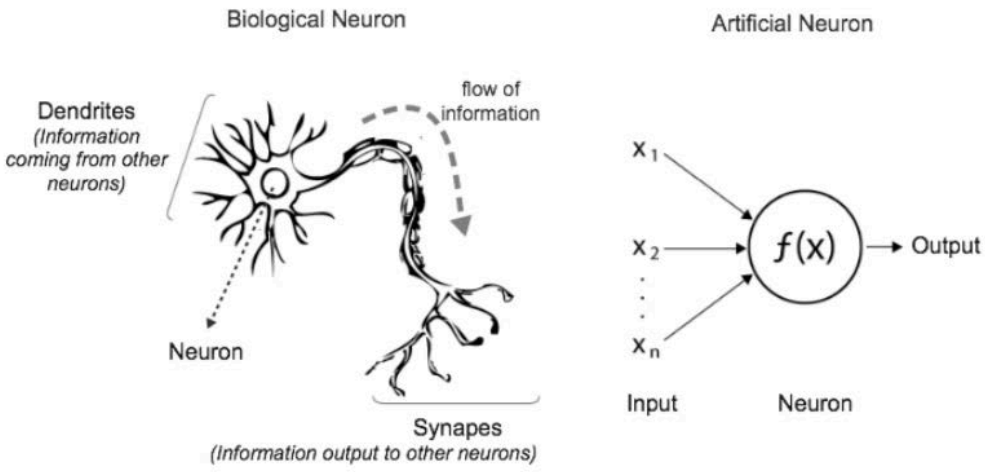


Figura 2.2. *Esquema del perceptrón y una neurona [5]*

La neurona recibe señales eléctricas de sus dendritas, las cuales están en contacto con otras neuronas. Éstas modulan las señales en distintas cantidades, y disparan la señal de salida a través de sus botones sinápticos a otras neuronas, únicamente si la intensidad de la señal de entrada supera un cierto umbral. [6]

Una red neuronal artificial lleva a cabo este funcionamiento a través de dos funciones. La primera, calcula la suma de las entradas con sus correspondientes pesos, término fundamental de la red, (esto representará la intensidad de la señal de entrada).

La segunda función, conocida como función de activación, determinará el umbral. Es decir, dará como resultado un 1 si la señal excede este umbral o un 0 si no. Para comprender este paso, podemos observar en la figura 2.3 en la cual se observa una de las funciones más usadas en esta etapa.

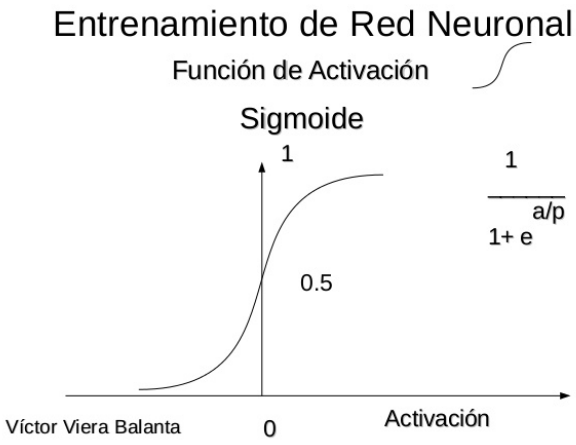


Figura 2.3. *Función de Activación. Sigmoide [7]*

Para este problema es necesario el uso de redes neuronales convolucionales, ya que este tipo específico de red son utilizadas, ya que funcionan mejor, en la manipulación de imágenes. La diferencia con respecto a las ya explicadas anteriormente, es que se filtrarán las imágenes previamente antes de entrar en el conocimiento profundo.

El esquema matemático real de una neurona artificial, tal y como se observa en la siguiente figura 2.4, está compuesto, para empezar, por la suma de unos datos de entrada, lo cuales serán multiplicados por unos pesos. Esto es debido a que los datos de entrada no tienen la misma influencia en la salida. Es decir, hay datos que influyen más que otros. Los pesos serán los valores que ha de determinar la red.

Seguidamente, nos encontramos con la función de activación, la cual nos permite realizar redes complejas, eliminando la linealidad de las funciones.

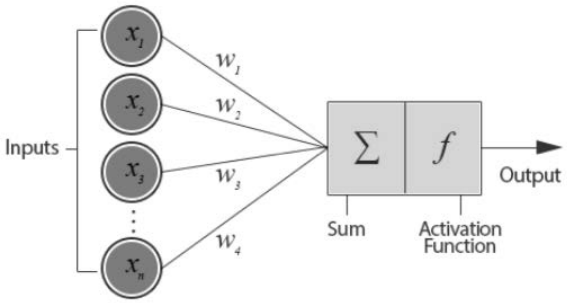


Figura 2.4 *Esquema matemático de una red neuronal [8]*

La complejidad de las redes neuronales se consigue al juntarlas unas con otras. Como se mostraba en la figura 2.2. En dicha imagen, se observa como la arquitectura de la misma está compuesta por tres partes.

La primera, la capa de entrada, la cual contiene las características de los datos de entrada. La segunda, las capas ocultas, las cuales son un conjunto de capas internas de la red que contienen y determinarán los pesos que darán la solución del problema. Finalmente, la capa de salida, que supondrá el valor, o los valores de los resultados.

Un ejemplo de una red neuronal puede ser tan sencillo como la predicción del tiempo. En base a unos datos de entrada y, yéndose al pasado, unos resultados, las temperaturas. se enseñará a la red para que sea capaz de predecir la temperatura futura según unos datos de entrada.

2.2 RED NEURONAL CONVOLUCIONAL CNN

Las redes neuronales convolucionales son consideradas una evolución de las ya estudiadas anteriormente para el tratamiento de imágenes.

Para este apartado es importante definir como un programa entiende las imágenes. Una imagen digital está compuesta por tres matrices de tamaño altura x anchura. En cada casilla de la matriz representan los valores de los pixeles.

Cada matriz define un color, los cuales son rojo, verde y azul, esto se define como canales de colores. El valor de un píxel de una matriz define la cantidad de color, al que pertenezca dicha matriz (rojo, verde o azul), que posee la imagen en dicha posición. Una red que reciba imágenes como datos de entrada recibirá estos valores tal y como observamos en la figura 2.5.

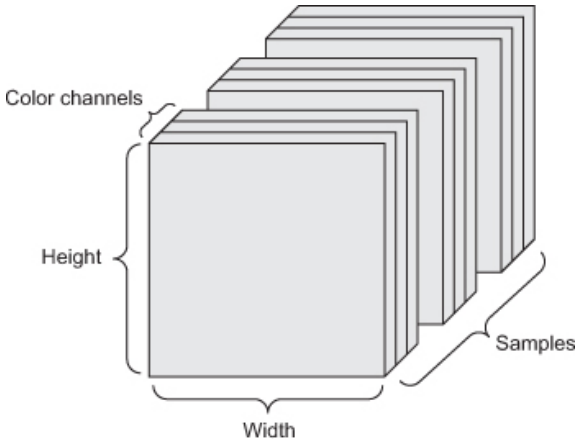


Figura 2.5 *Representación de las muestras de imágenes que entran como datos de entrada a una CNN [9].*

La arquitectura de este tipo de redes sigue el mismo patrón que las ya mencionadas. Distintas capas, unas encimas de otras, las cuales reciben unos datos de entrada a los que se le aplica unos pesos y una la función de activación para formar los datos de salida. La diferencia es que estas hacen uso de lo que se denomina convolución.

Las capas convolucionales aprenden patrones del objeto al tratar las distintas imágenes. Al pasar la primera convolución aprende distintas características de la imagen, y, a medida que avanza a lo largo de las distintas capas de la red aprende patrones más genéricos.

Empezando, por ejemplo, tras la primera capa de la convolución, por el aprendizaje de patrones específicos de una cara, tras el aprendizaje de las características de esta, hasta el reconocimiento de una cara completa, tal y como se aprecia en la siguiente figura 2.6

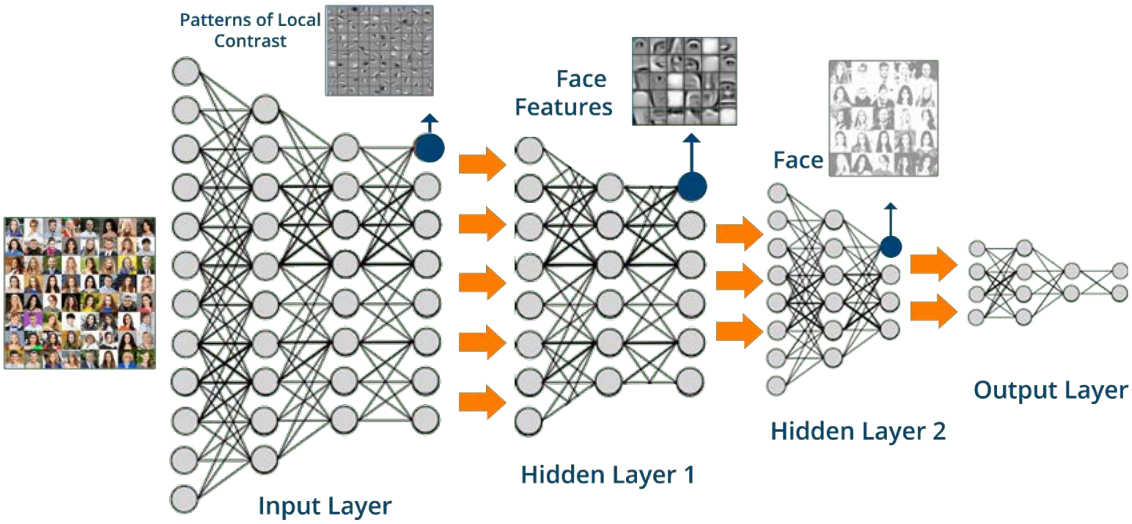


Figura 2.6 *Arquitectura de una red convolucional [10]*

Las capas convolucionadas son entendidas como filtros capaces de detectar patrones específicos de un objeto. Las matemáticas que se encuentran detrás de esta operación es únicamente la multiplicación una matriz que se desliza a través de la imagen. Según la matriz que se multiplique, pueden obtenerse hasta los contornos de los objetos en una imagen. Para visualizar este comportamiento se muestra, a continuación, el resultado de aplicar estas operaciones en la imagen a tratar, tal y como muestra la siguiente figura 2.7

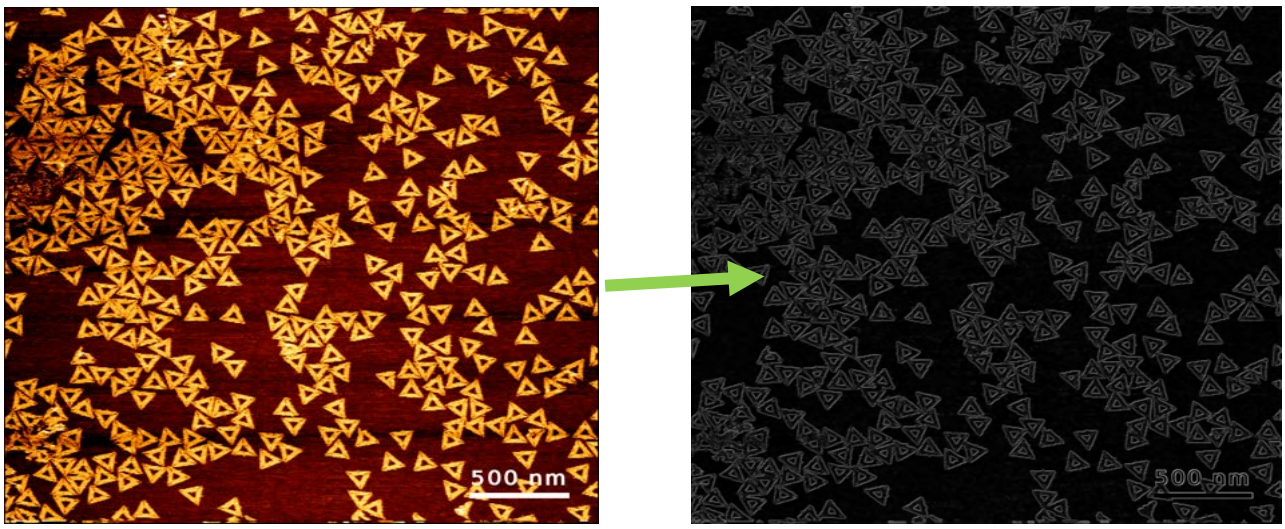


***Figura 2.7**. Resultado de una imagen tras la aplicación de diversas convoluciones*

Además de la convolución en este tipo de redes neuronales se encuentra con la operación de max-pooling. Esta función se utiliza para reducir las muestras de los patrones de características que salen de las convoluciones, es decir, reduce la información.

La operación de max-pooling consiste en dividir la matriz de salida de la convolución en pequeñas ventanas. Una vez divididas se mantiene únicamente el valor máximo de cada una, debido a ser la que almacena más información del pequeño conjunto de valores. Para entender mejor este concepto véase la figura 2.8.

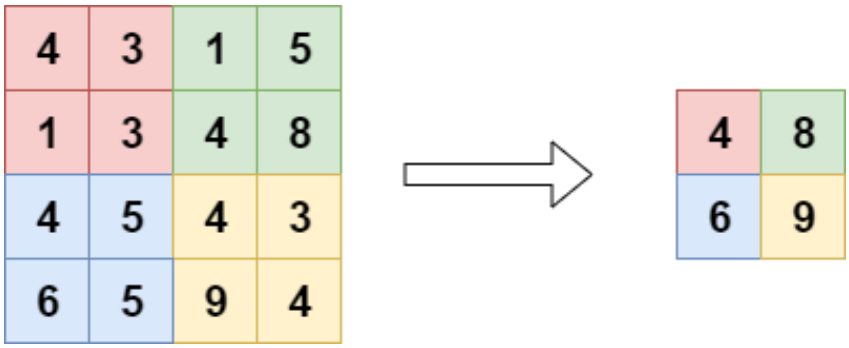


***Figura 2.8.** Operación Max-pooling [11]*

Tras estas operaciones, los datos, finalmente, se reducen a una única dimensión y entran en una red neuronal artificial como la explicada ya anteriormente. Es por ello por lo que se hace uso de la operación de max-pooling.

Este concepto trata de reducir el tamaño del vector de entrada a estas capas, debido a que para entrar en este tipo de redes los datos que entran han de ser de una única dimensión. Es

decir, se han de aplanar las matrices junto con sus canales, las cuales dentro de una capa de red se conocen como profundidad, a un único vector de tamaño altura x anchura x profundidad.

Los pasos que sigue una determinada ventana de una imagen en una red convolucional se resumen en la figura 2.9.

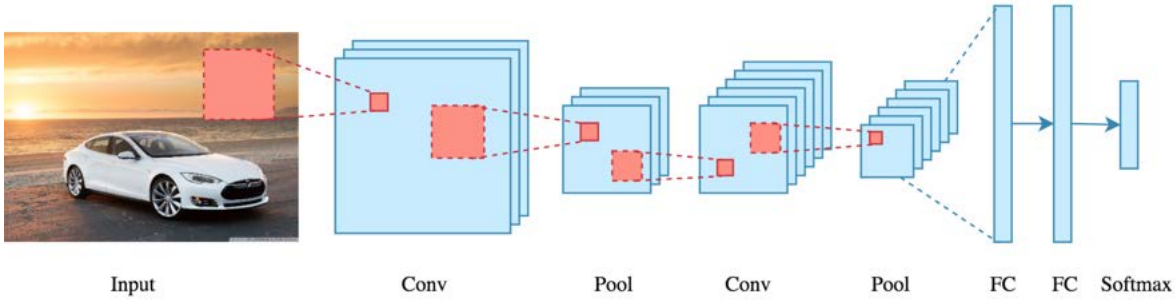


Figura 2.9 *Arquitectura red convolucional [12]*

En la figura 2.9, se puede observar como la profundidad aumenta a medida que pasa por las convoluciones. Cabe destacar que esto no implica que aumenten las matrices de colores, si no que las matrices ahora representan unos patrones, es decir, en el ejemplo del rostro, una matriz puede representar una nariz, otra los ojos, otra la boca y así consecutivamente hasta que el conjunto defina el patrón de la cara.

2.3 ENTRENAMIENTO

Una vez comprendido lo que es una red neuronal convolucional y los pasos que sigue una imagen a través de esta, es importante entender como es capaz de aprender.

El bloque de construcción de redes neuronales es la capa, por donde se filtran los datos. En una capa entran datos, se filtran y salen representaciones de los mismos. Para que una red neuronal aprenda, es necesario definir tres conceptos

Para empezar, la función de pérdidas, función que va a ser capaz de medir el comportamiento de los datos de entrenamiento y que va a ser capaz de mejorar su rendimiento dirigiendo el valor de los pesos en la dirección correcta.

También, el optimizador, mecanismo por el cual la red se actualiza a si misma basándose en los datos y la función de pérdida. Y, finalmente, la métrica que monitora durante el entrenamiento y el testeo. Ésta se preocupa de la precisión. Es decir, valora lo que ha predicho correctamente midiendo la diferencia entre lo que se ha predicho y el resultado real.

Como ya se ha mencionado, el aprendizaje de las redes neuronales se basa en definir los valores óptimos de los pesos. Recuérdese que los pesos son los valores que multiplican a los datos de entrada de la red con el fin de dar mayor o menor influencia a estos. Un concepto

más técnico es que las transformaciones implementadas por una capa son parametrizadas por sus pesos [13].

Para llevar a cabo este aprendizaje se hace uso del algoritmo *Backpropagation*. Este algoritmo asigna a los pesos unos valores aleatorios, los cuales, en la primera iteración, se aplican a los datos y se obtienen unos resultados, una predicción Y'.

Con el fin de mejorar la red neuronal es necesario conocer la distancia entre el resultado predicho (Y') y la solución real (Y). Se genera entonces una función que ejecuta esta distancia para todas las predicciones de los datos de entrada y sus resultados. Dicha función se considera como una señal de retroalimentación que servirá para ajustar los valores de los pesos.

Para ello, se hace uso de una serie de ecuaciones diferenciales con el fin de encontrar el valor de los pesos óptimos en el que el resultado de la función de pérdidas sea el valor mínimo.

Dada una función diferenciable, es teóricamente posible encontrar el mínimo. Se sabe que el mínimo de una función es el punto en el cual la derivada es 0. Únicamente se ha de encontrar la combinación de los puntos en los que la derivada sea 0 y, así, comprobar por cual de estos puntos da como resultado el menor valor.

Es decir, para las redes neuronales, estos puntos son los pesos y, para encontrar los mínimos se hace uso del gradiente de la función de pérdida, tal y como se observa en la figura 2.10.

Dicha función es definida por el creador de la red neuronal y varía según el problema. En capítulos posteriores se estudiará la función de pérdida utilizada para el problema a solucionar.

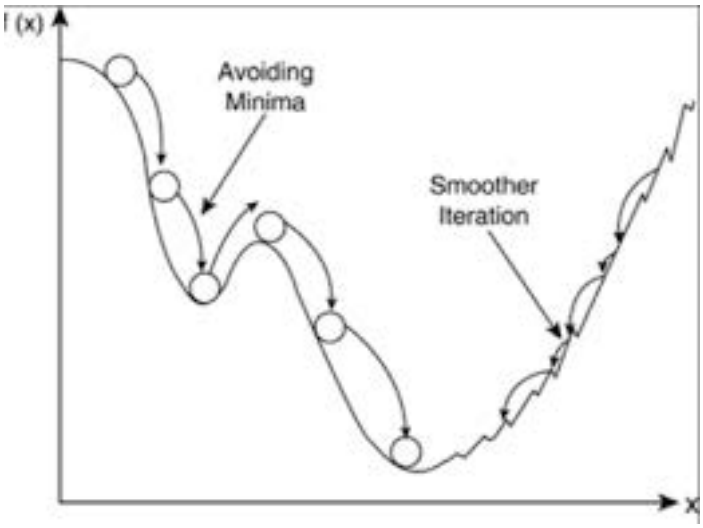


***Figura 2.10**. Función sencilla de pérdida [14]*

Una vez calculado dicho error y la derivada de esta, se desplaza a través de la función de pérdida en busca del mínimo. Es decir, los pesos, de los cuales depende esta función, irán variando su valor. Dicha variación es definida, además, por un valor denominado impulso, el cual define la velocidad a la que aprende el algoritmo. El desarrollo de lo explicado anteriormente se observa en la figura 2.11.

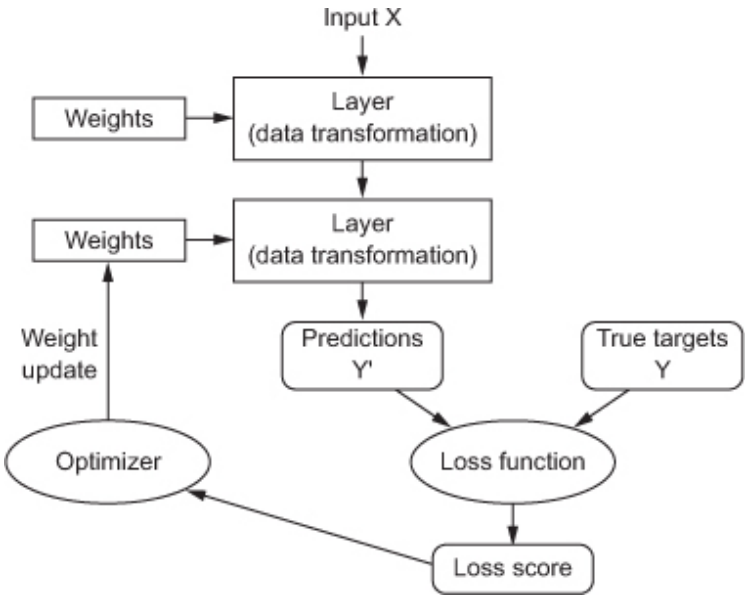


***Figura 2.11**. Proceso de entrenamiento de la red [15]*

En definitiva, el algoritmo sigue ciertos pasos que se repiten una y otra vez, los cuales se conocen como mini-batch stochastic gradient descent (mini-batch SGD) [16]. Estos pasos son los siguientes:

1. Escoge un conjunto de datos de entrenamiento (X) y los representa con sus correspondientes resultados (Y).
2. Ejecuta la red para los datos escogidos (X) y se obtienen los resultados predichos (Y')
3. Ejecuta la pérdida de la red, mediante la función de pérdida que mide la diferencia entre Y e Y'.
4. Genera el gradiente de la función junto con los pesos que han sido utilizados para la predicción
5. Varía el valor de estos pesos en la dirección opuesta al gradiente y haciendo uso del optimizador definido.

Existen múltiples variaciones en la actualización de los pesos según el problema. Estas variantes se denominan métodos de optimización, tal y como se observa en la figura 2.11

Otro factor importante en el proceso de aprendizaje de una red neuronal es la necesidad de dividir los datos en dos grupos. Éstos son, para empezar, los datos de entrenamiento, los cuales serán a los que se le aplique algoritmo de aprendizaje. Y, por otro lado, los datos de validación, los cuales tienen como función comprobar el funcionamiento de la red neuronal con datos nunca vistos.

El hecho de que una red funcione de manera óptima con los datos de entrada con los que se ha entrenado no significa que a la hora de pasarle otros nunca vistos funcione de la misma manera. El objetivo de una red neuronal es encontrar el equilibrio entre dos conceptos: la optimización y la generalización [16]

La optimización se refiere al proceso en el cual el modelo se ajusta para obtener el mejor rendimiento posible con los datos de entrenamiento. Por otro lado, la generalización se refiere a la capacidad de la red de ajustarse a datos nunca vistos.

Al principio del entrenamiento, ambos conceptos están correlacionados. A medida que mejora la optimización, la generalización avanza de la misma manera. Surgen, en este proceso dos conceptos muy importantes los cuales se definirán a continuación

2.3.1 SOBREAJUSTE Y SUBAJUSTE (OVERFITTING / UNDERFITTING)

En el apartado anterior se han definido los conceptos de optimización y generalización, como ya se ha comentado, al principio del entrenamiento ambos están correlacionados. Sin embargo, a lo largo del entrenamiento llega un punto en el cual la generalización empeora a medida que mejora la optimización. Esto es conocido como sobreajuste o “overfitting”.

El sobreajuste aparece cuando la red deja de aprender generalizaciones y comienza a aprenderse de memoria los datos, es decir, los conceptos específicos de esos datos. Aprende lo que se conoce como ruido. La solución a este tipo de problemas es el aumento de los datos.

El subajuste, conocido como “Underfitting”, hace referencia a cuando la red no ha aprendido lo suficiente y obtiene unos resultados con un error elevado tanto para los datos de entrenamiento como para los de validación.

El objetivo es encontrar el punto intermedio para no caer en ninguno de estos conceptos. Para ello, es necesario un mayor número de datos, ya que cuantos más datos haya más generalizaciones se encuentran. Sin embargo, a veces esto no es posible, a pesar de que existen distintas técnicas, las cuales se comentarán más adelante, para solucionar este tipo de problemas. Una manera de reducir el sobreajuste es disminuyendo el modelo creado. Así, únicamente será capaz de aprender conceptos más genéricos.

Un ejemplo de un problema genérico que muestra lo que se acaba explicar se puede observar en la siguiente gráfica (Figura 2.12). El eje x representa las iteraciones dada por la red y el eje y representa el error.

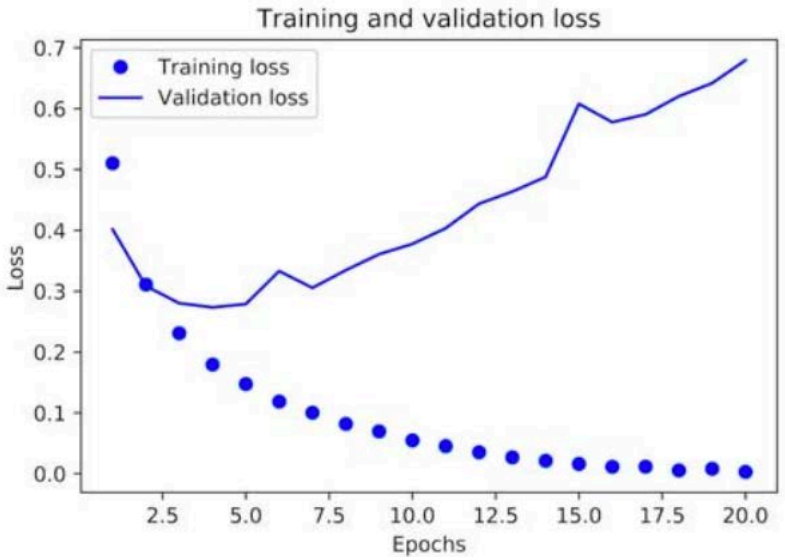


***Figura 2.12**. Error sobre los datos de entrenamiento y validación según las iteraciones [16]*

2.4 TIPOS DE PROBLEMAS

Una vez llegados a este punto, es necesario para el desarrollo del proyecto diferenciar entre los distintos tipos de problemas de aprendizaje profundo basado en la visión computacional.

Para empezar, el más sencillo y típico de todos, es la clasificación de imágenes. En este tipo de problemas es necesario definirlas con una etiqueta, la cual, será el resultado de los datos de entrada (las imágenes). Se puede observar el resultado de este tipo de problemas en la figura 2.13.

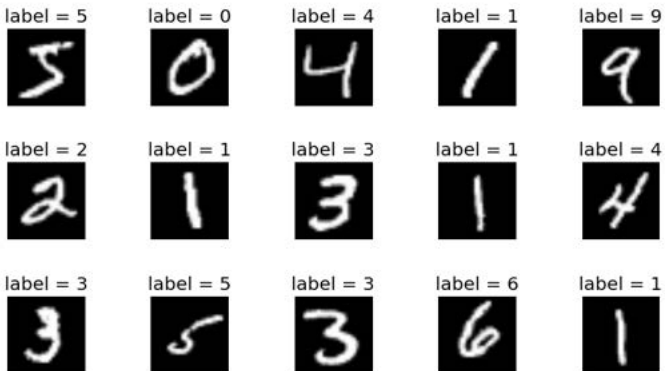


***Figura 2.13** Problema clasificación de imagen [17]*

Recibir una imagen y devolver el porcentaje de que en dicha imagen exista un determinado objeto puede ser de gran utilidad o no ser suficiente para el comportamiento esperado. Por ejemplo, para un coche capaz de conducir por si mismo, el observar que existe una persona en la imagen no le es suficiente, ha de ser capaz de determinar la posición exacta de la misma.

Es entonces cuando aparece el problema de detección de objetos, en el cual, además de definir que en una imagen se encuentra un objeto, ha de ser capaz de hallar la posición en la que se encuentra junto con su altura y anchura. Además, este tipo de problemas son capaces de encontrar diferentes objetos en la misma imagen, tal y como se puede observar en la figura 2.14.

Figura 2.14 *Detección de objetos [18]*

Otro tipo de problemas es denominado segmentación semántica, el cual trata de entender qué elementos se encuentran en una imagen. Le asigna a cada píxel una etiqueta, y aquellos que compartan la misma etiqueta pertenecen al mismo objeto. En la siguiente figura 2.15. se observa un ejemplo de éste.

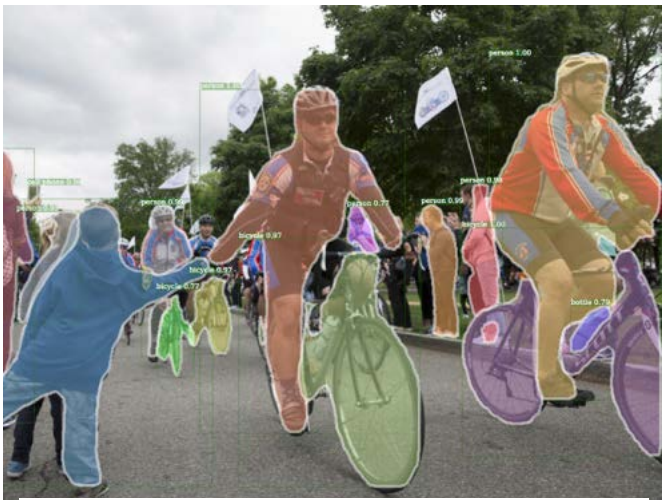


Figura 2.15*. Segmentación semántica [19]*

2.5 HERRAMIENTAS A UTILIZAR

Una vez entendido los conceptos básicos del machine learning se hablará sobre las herramientas necesarias para comenzar con el desarrollo del proyecto.

Para empezar, el lenguaje de programación usado para el desarrollo del proyecto será Python. Como ya se ha comentado anteriormente, el problema planteado es encontrar una red neuronal que encuentre la posición de los triángulos en una imagen. Para ello, es necesario comprender que, para que la red sea capaz de detectar las posiciones en futuras imágenes, es necesario definir las soluciones de los datos de entrada, es decir, se deberá etiquetar manualmente las imágenes.

Las herramientas que se usarán durante el desarrollo del proyecto son las siguientes:

1. **LabelImg** [20]. Este programa será el que se utilice para etiquetar las imágenes.

2. **Github**. Es un sistema por el cual, los desarrolladores gestionan sus proyectos de manera online. En esta página web se encuentran códigos ya creados por otros desarrolladores que se utilizarán para el desarrollo del problema.

3. **Sublime Text**. Editor de texto, en el se desarrolla parte del código del proyecto

4. **Jupyter Notebook.** Entorno de trabajo interactivo, útil para hacer pruebas de código y preparar las imágenes de manera visual

5. **Anaconda.** Gestor de entorno para el desarrollo en Pyhton, cuenta con una gran variedad de librerías y es de código abierto.

Capítulo 3. ESTADO DE LA CUESTIÓN

3.1 TÉCNICAS PARA LA CREACIÓN DE REDES NEURONALES

Como ya se ha comentado anteriormente, para el proyecto planteado, es necesario el uso de redes neuronales. Para ser capaces de comenzar el desarrollo de una, es necesario conocer las distintas técnicas realizadas para la aplicación de esta.

El problema fundamental de este tipo de técnicas es la cantidad datos necesarios para ser capaz de conseguir soluciones óptimas. Entrenar una red desde cero puede requerir una gran cantidad de datos. Cuanto mayor sea esta cantidad de información, mejor entrenada estará la red. El principal problema que conlleva este tipo de problemas es, como ya se ha comentado en el capitulo anterior, lo que se conoce como overfitting.

Como ya se comentó previamente, existen diversas técnicas para tratar la insuficiencia de datos. Este tipo de problemas surgen sobretodo para modelos que tratan con imágenes. En caso de encontrarse con estas dificultades una posible técnica para solventarlo es el aumento de datos. Este concepto se basa en generar más datos usando los ya existentes.

Para ello se aplican a los datos transformaciones aleatorias. La idea fundamental es que la red aprenda con un mayor número de datos de entrada sin que estos se repitan. Dichas transformaciones pueden ser aumentar o reducir una imagen, cambios de color, rotaciones, dar la vuelta a la mitad, tanto horizontalmente como verticalmente, aumento de pixeles, etc.

Un ejemplo de este tipo de técnicas aplicados a las imágenes que se utilizarán para el desarrollo del proyecto puede observarse en la siguiente figure 3.1 Se observa, a la izquierda, estas técnicas aplicadas a un único triangulo, y, a la derecha, aplicados a la imagen completa.

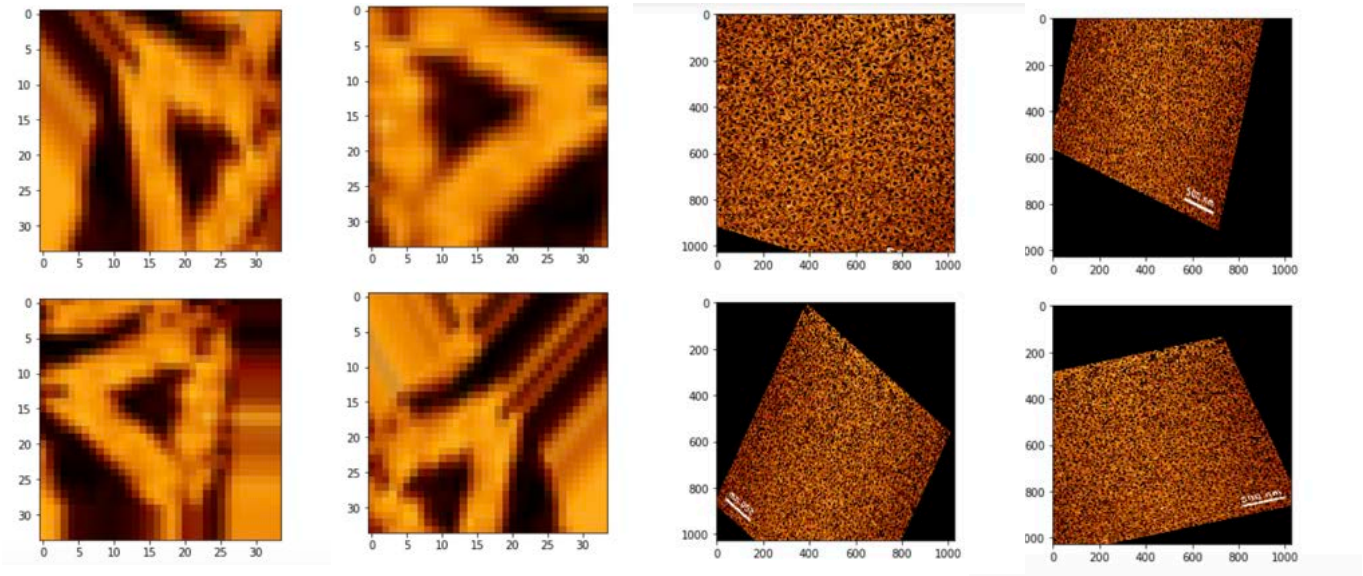


Figura 3.1 *Aumento de datos*

Otra aplicación para disminuir los efectos que producen la insuficiencia de imágenes es el uso de una red reentrenada. Una red reentrenada es una red ya entrenada con una gran cantidad de conjunto de datos y guardada. Si estos han sido, suficientemente genéricos y grandes, significa que la red ha aprendido una gran jerarquía espacial de características. Esto supone que este tipo de características aprendidas por la red pueden ser de gran utilidad incluso para otros problemas que busquen diferentes soluciones.

Existen dos métodos para usar esta técnica. El primero, se denomina extracción de características, en donde se hace uso de las representaciones aprendidas en la red ya entrenada, para extraer características interesantes de los nuevos datos. Estas características son ejecutadas en el nuevo clasificador, el cual se entrena desde cero.

Para poder aplicar este método se extrae la base convolucional del modelo ya entrenado. Es decir, se apropia de las capas de agrupación máxima (maxpooling) y convolucionales. Éstas serán añadidas a un nuevo clasificador adaptado al problema a solucionar. Tal y como se observa en la siguiente figura 3.2

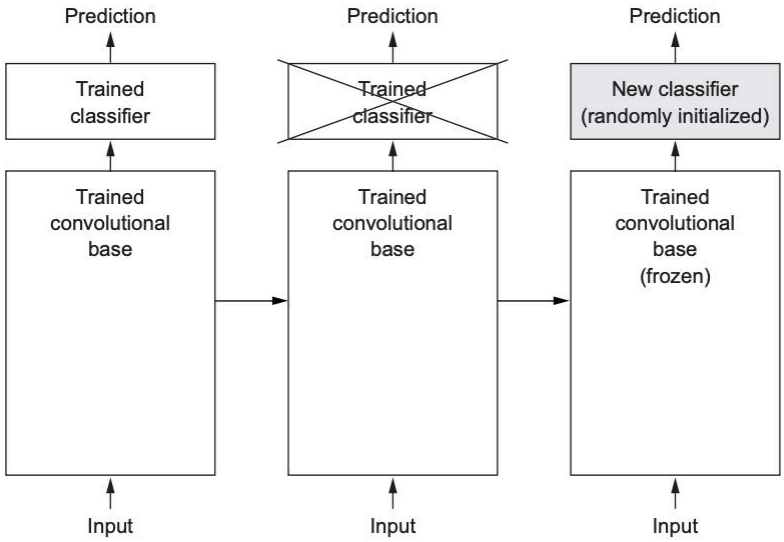


Figura 3.2 *Proceso de extracción de características [21]*

Las razones por las que se lleva a cabo este tipo de estrategias son porque en la base convolucional las capas han aprendido características espaciales más genéricas. Para poder mantener estas características aprendidas se han de congelar los pesos ya que ya fueron aprendidos en el modelo.

Es decir, en la red neuronal ya entrenada hará uso de miles de imágenes para aprender a extraer las características que dan mejor solución al problema. Es por ello por lo que estos pesos ya han sido aprendidos y han de mantener su valor para mantener su funcionalidad. Así es como da el efecto de aumento de datos, ya que esos valores han sido obtenidos gracias a miles de datos con los que se entrenó dicha red.

Finalmente, el otro método para usar una red neuronal es conocido como "*Fine-tuning" [22]*. En realidad, en esta técnica se aplica la misma idea que lo que se acaba de explicar. La diferencia entre ambos es que no se congelan todas las capas convolucionales. Esto es debido a que a medida que una imagen pasa por mayores convoluciones extrae características cada vez menos genéricas.

Debido a que se usan redes neuronales entrenadas en datos que pueden ser completamente diferente a los del nuevo problema, es interesante que los pesos de éstas últimas convoluciones se adapten al nuevo clasificador, tal y como se observa en la siguiente figura 3.3

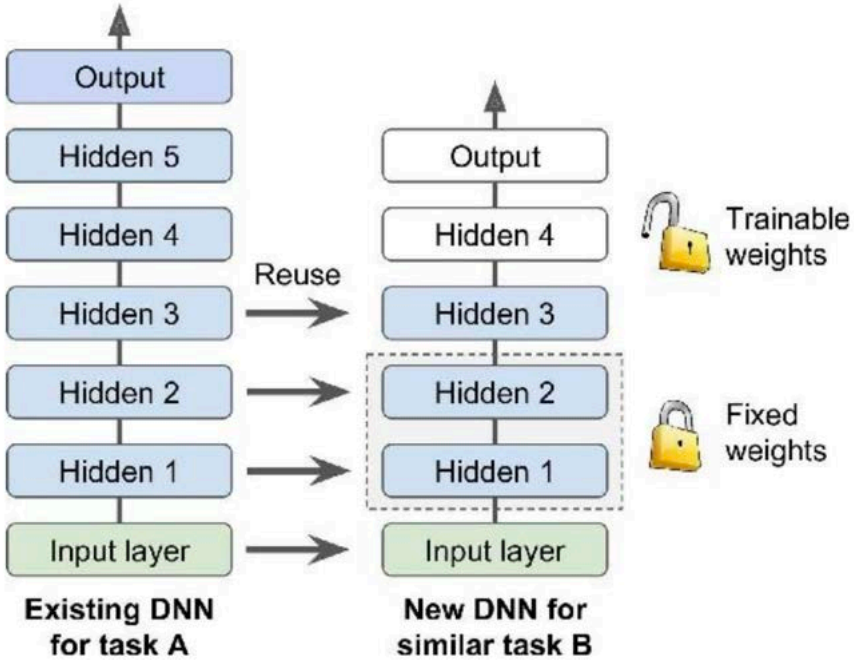


Figura 3.3 *Arquitectura "Fine-tuning" [23]*

3.1.1 CLASIFICADOR DE TRIÁNGULOS

Con el fin de comprobar ejemplos de este tipo de técnicas se ha desarrollado una red neuronal sencilla, en la cual como datos de entrada se utilizan los triángulos recortados del conjunto de datos del proyecto a solucionar, ya que, al haberse etiquetado todas las imágenes, se han obtenido las posiciones exactas de los triángulos y únicamente se ha procedido a recortarlos. Se han obtenido así aproximadamente 9000 imágenes de triángulos.

Antes de continuar, es necesario comprender que si a una red neuronal, únicamente se le enseñan imágenes de una única clase, ésta va a aprender que sea cual sea la imagen la solución única va a ser triangulo. Por ello, se debe crear otra clase que sea contraria a lo que se busca.

En este caso se ha definido una clase que es la opuesta a esta. Para ello se ha recortado las imágenes del conjunto de datos mencionado anteriormente, en las zonas en las que no se encuentra ninguna figura. Un ejemplo del conjunto de datos a entrenar puede observarse en la siguiente figura 3.4

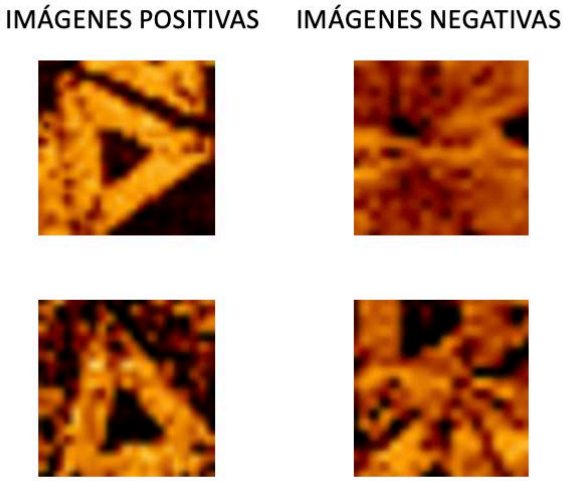


Figura 3.4 *Ejemplo del conjunto de datos a entrenar*

Para comprender los resultados es necesario tener en cuenta que, aproximadamente, se hace uso de 18000 imágenes para entrenar la red, lo cual constituye un volumen de información elevado. A continuación, en la figura 3.5, se muestran las distintas arquitecturas de los modelos a comparar.

Cabe destacar que en el primer ejemplo mostrado en la figura se hará uso, además, de la técnica de aumento de datos. Además, para la aplicación de extracción de datos y fine-tunning se hace uso de la red ya entrenada VGG16 [24].

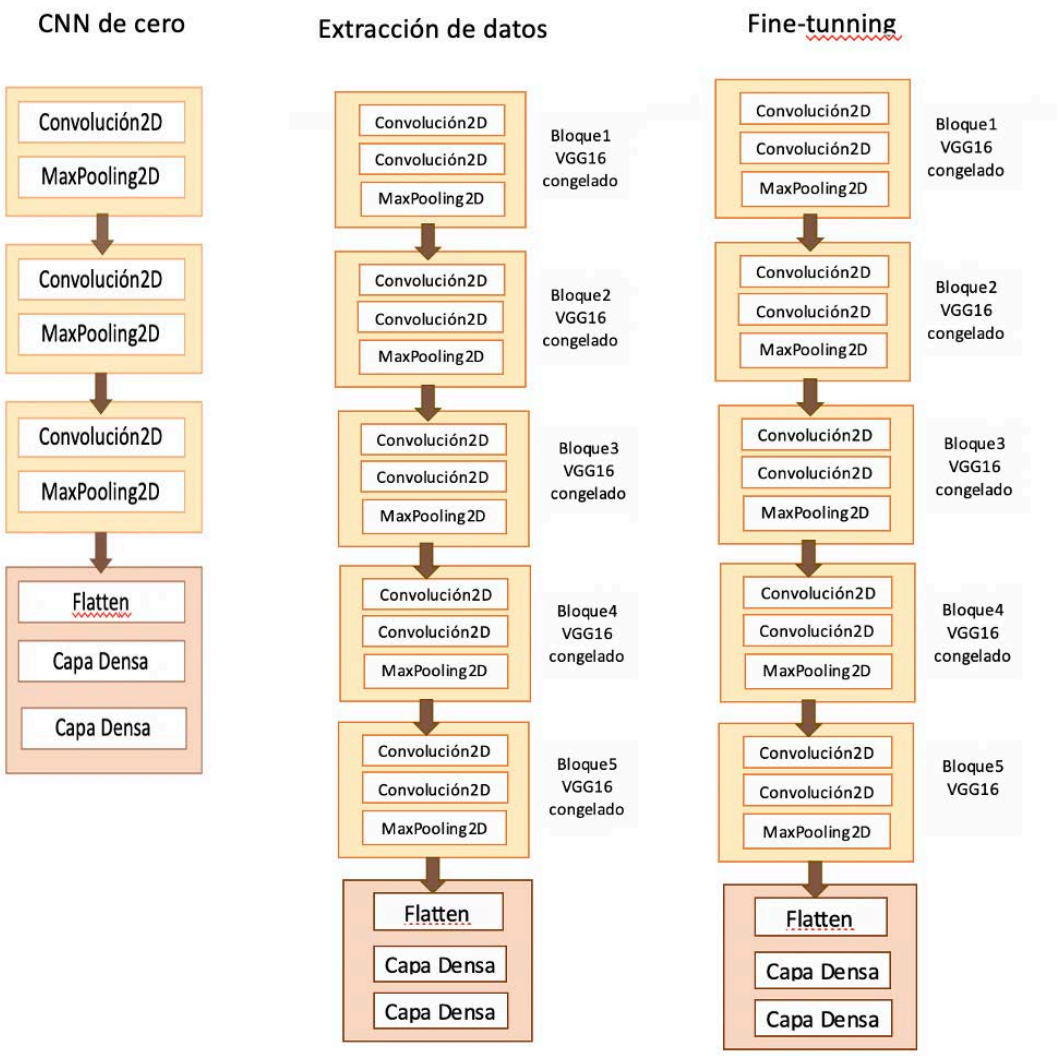


Figura 3.5 *Comparación de distintas arquitecturas de redes neuronales*

Una vez vistas las arquitecturas de cada una, se observan las gráficas, las cuales se definen en la siguiente figura 3.6, donde se encuentran dos gráficas por solución.

La primera gráfica muestra el porcentaje de precisión, tanto para los datos de entrada como para los de validación. La segunda gráfica muestra el procentaje de error. Ambas dependen del número de iteraciones que realiza la red para aprender.

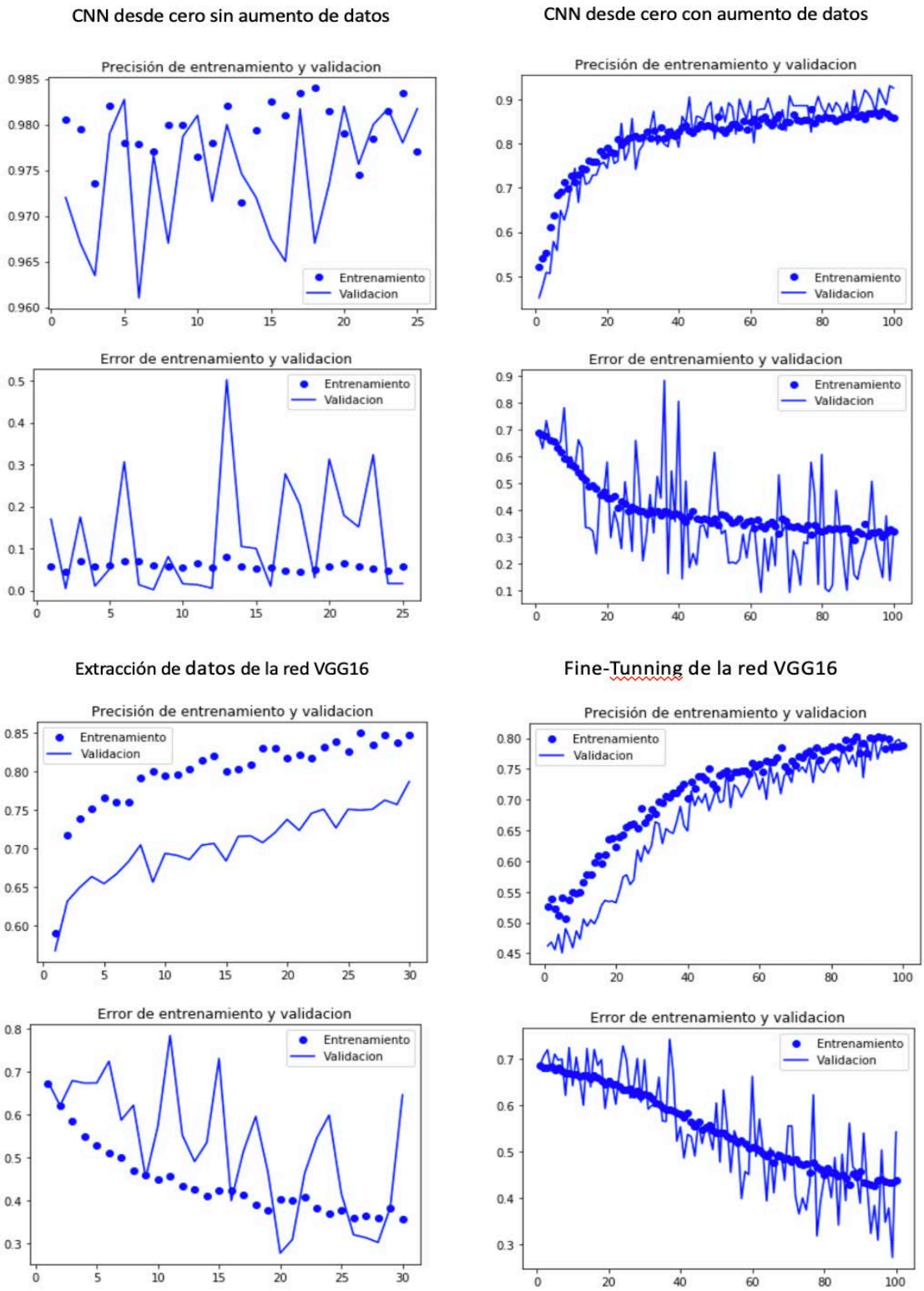


Figura 3.6 *Gráficas de los distintos modelos*

Como se puede comprobar, para el problema planteado, la red neuronal convolucional creada desde cero obtiene mejores resultados. Esto es debido a que este tipo de soluciones son apropiadas para la creación de modelos con pocos datos de entrenamiento.

Además, las imágenes con las que se entrena a la red no son de un gran tamaño, aproximadamente de unos 34x34 pixeles, y, estos modelos ya entrenados son muy profundas, es decir, contienen muchas capas convolucionales que no son necesarias para el problema a desarrollar.

Más adelante, se aplicará una de estas técnicas para el problema de detección ya que, a medida que aumenta la complicación del modelo, son necesarios más datos para la creación de una red neuronal óptima. Además, se explicarán los distintos parámetros utilizados para la creación de esta en el siguiente capítulo.

3.2 APLICACIÓN RED NEURONAL CONVOLUCIONAL

En el apartado anterior se explicaron distintas técnicas para la creación de una red neuronal. Sin embargo, es necesario entender los distintos parámetros a definir en la creación de una. En el siguiente capítulo se implementará.

En este apartado, se define una manera en la que puede llevarse a cabo el proyecto. Como ya se ha mencionado anteriormente, las redes convolucionales son un buen método para la clasificación de imágenes. Sin embargo, el proyecto a desarrollar es la creación de una red neuronal capaz de detectar la posición de los triángulos.

Para ello, una manera de dar solución a este problema es mediante la creación de un modelo de clasificación, tal y como se ha definido en el apartado anterior, y, con el fin de detectar los triángulos en la imagen, hacer uso de algoritmos definidos que faciliten el recorte de la imagen para poder pasar partes de la misma por el clasificador. Éste definirá si pertenece o no a un triangulo. A continuación, se procederá a la explicación de estos algoritmos.

La razón por la cual es necesario el uso de algoritmos es debido a la lentitud de tratar de localizar los triángulos de una imagen al pasar el clasificador por toda la imagen. Ya que pueden surgir varios problemas, los cuales se observarán en el siguiente capítulo en el cual se observa la implementación de este procedimiento.

Búsqueda Selectiva (Selective search) [25] es un algoritmo de búsqueda de objetos dentro de una imagen. El principal problema al que se enfrenta es la gran cantidad de localizaciones posibles dentro de una imagen en las cuales puede existir un objeto. Examinar todas estas localizaciones posibles puede ser computacionalmente inviable y por ello se han de seleccionar regiones analizando los píxeles de las imágenes. Es decir, el objetivo es explorar las estructuras dentro de una imagen para definir posibles localizaciones de objetos.

Este algoritmo está sujeto a las siguientes consideraciones de diseño:

1. Tener en cuenta todas las escalas, es decir, un objeto puede encontrarse en una posición lejana o simplemente ser más pequeño y ocupar una escala más pequeña, mientras otro objeto se encuentre más al frente o sea más grande y ocupe una escala mayor. Por ello se propondrán diversos tamaños de objetos

2. Diversificación, No existe ninguna estrategia óptima para agrupar distintas regiones, un objeto puede ser identificado por color, por textura o por cercanía, por ejemplo, un coche puede ser blanco y tener unas ruedas de color negras, no coinciden en color ni en textura, sin embargo, pertenecen al coche y el algoritmo es capaz de reconocerlo mediante la cercanía.

3. Rapidez. Se busca un algoritmo de implementación rápida

Un ejemplo del desarrollo de este algoritmo se observa en la siguiente imagen. Figura 3.7

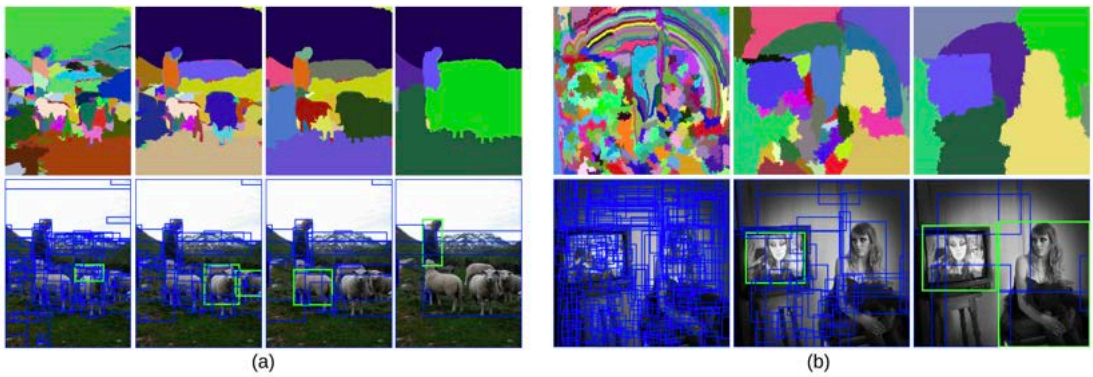


Figura 3.7 *Dos ejemplos de aplicación del algoritmo Selective Search [25]*

La manera en la que trabaja la búsqueda selectiva es aplicando, para empezar, un algoritmo de segmentación que define regiones por colores que pueden ser objetos, tal y como se muestra en la figura 3.2.1. Este algoritmo trata de crear 2000 áreas de posibles objetos.

Para ello aplica los siguientes pasos:

1. Primero se calculan las similitudes de las regiones vecinas, teniendo en cuenta la diversificación ya explicada anteriormente.
2. Las dos regiones más similares se unen y vuelven a calcularse las similitudes entre las nuevas regiones con sus vecinos.
3. Este proceso se repite hasta que la imagen aparece como una única región.

Sin embargo, mediante la aplicación de este método, la cual se observará en el siguiente capítulo, surgen varios problemas y no aparecen soluciones óptimas, ya que el problema de detección es más complejo de lo que puede parecer. En el siguiente apartado se explicará otra manera de dar solución al problema planteado.

3.3 MODELOS DE DETECCIÓN DE OBJETOS

Como se acaba de comprobar, la aplicación de una red neuronal de detección de objetos es mucho más compleja de lo que se puede esperar.

Sin embargo, otra manera de llevar a cabo este problema es mediante un modelo completo entrenado que tenga la función de detectar objetos, es decir, en vez de seleccionar regiones mediante algoritmos y pasarlos a través de un clasificador, estos algoritmos formarán parte del modelo a entrenar.

El objetivo fundamental es encontrar las coordenadas en la imagen en la que se posiciona el objeto. Para ello se hace uso de las cajas delimitadoras, en inglés se conoce como Bounding Box. Además, es necesario indicar el tipo de objeto que se encuentra en dicha posición.

Es decir, las salidas del modelo serán, para empezar, la posición del cuadrado que obtenga la imagen, el cual puede estar definido de dos maneras: Mediante la posición en la imagen de las esquinas (Xmin,Ymin,Xmax,Ymax). O, mediante la posición del centro junto con la altura y la anchura del cuadrado que lo rodea (Xi,Yi,W,H) tal y como muestra la siguiente figura 3.8 . Por otro lado, se obtendrán los porcentajes de las clases que se hayan definido.

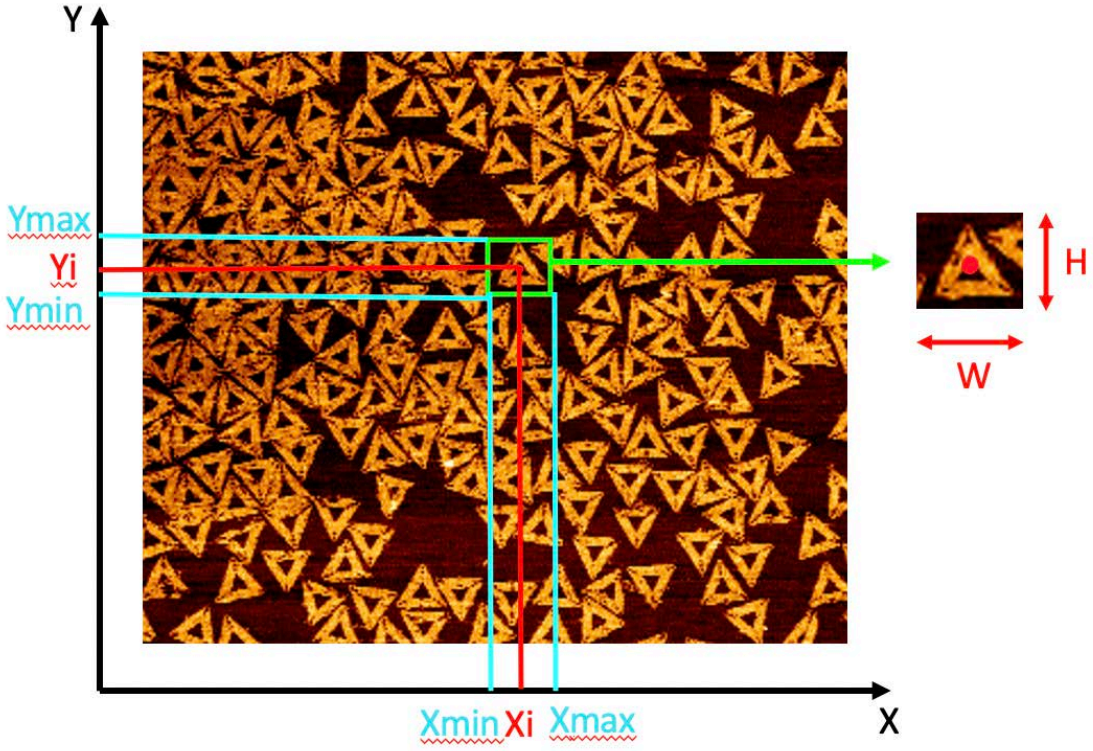


Figura 3.8 *Representación de la posición de un objeto*

La arquitectura de un modelo de detección de objetos se compone de cuatro componentes fundamentales, los cuales se definirán a continuación.

3.3.1 PROPUESTA DE REGIÓN (REGION PROPOSAL)

Este componente puede ser un modelo de aprendizaje profundo o un algoritmo, su función es encontrar regiones de interés. En esta sección se hace referencia a los algoritmos implementados en el apartado anterior.

En esta sección el sistema busca en una imagen distintas regiones con una alta probabilidad de contener, en su interior, un objeto. Para ello define un parámetro, denominado puntuación de objetividad, el cual servirá como umbral para definir si se ha de analizar la región o descartarla. En la figura 3.9 se puede observar un ejemplo de este apartado en una imagen.

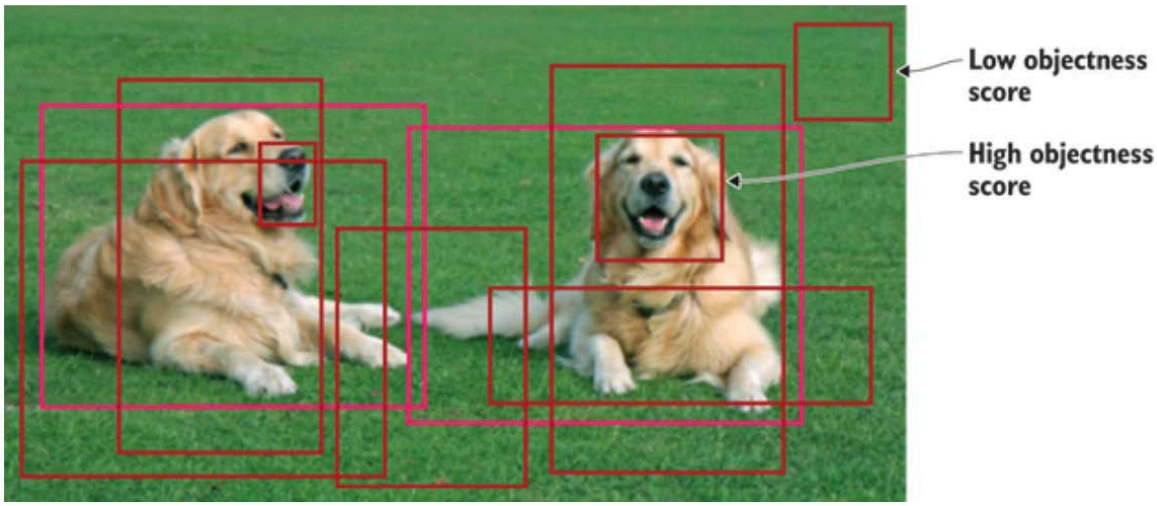


Figura 3.9 *Región de interés*

La manera en la que decide si una región contiene un objeto o no, como ya se ha comentado, es a través de un umbral, el cual determinará si una región pertenece a un fondo, baja puntuación de objetividad, o si pertenece a un primer plano.

3.3.2 PREDICCIÓN

En este apartado es cuando la red neuronal lleva a cabo sus predicciones. Para ello, normalmente, se basa en redes neuronales convolucionales ya reentrenadas. Utiliza este tipo de métodos para extraer las características de las imágenes de entrada y determinar las probabilidades de los objetos dentro de las regiones.

Este paso es muy importante ya que analiza todas las regiones para predecir las coordenadas del cuadro delimitador (Bounding Box) y la predicción de las clases. Uno de los problemas

que aparece tras la aplicación de estos dos componentes, como ya se ha podido comprobar previamente, es que, al pasar por el primer algoritmo, en el cual se proponen una gran cantidad de regiones y de distintos tamaños, es muy probable que existan diversas regiones de interés rodeando al mismo objeto.

Un ejemplo de este tipo de problemas se puede comprobar en la siguiente figura 3.10. Dicha imagen es un resultado obtenido tras aplicar lo explicado en el apartado 3.2. en el cual se ha hecho uso, primero de un algoritmo de propuesta de región, y, a continuación, ha sido analizado por el modelo creado.

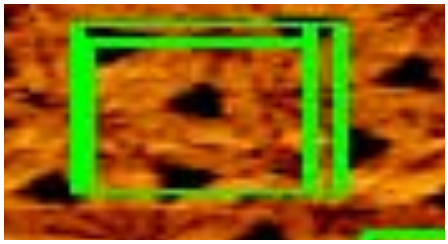


Figura 3.10 *Ejemplo de regiones de interés que rodean a la misma imagen*

Para solucionar este tipo de problemas se hace uso de técnica que se explicará a continuación, la cual se conoce como "*Non-Maximum Supression"*.

3.3.3 NON-MAXIMUM SUPPRESSION (NMS)

Como se acaba de mencionar, esta técnica es utilizada para que los algoritmos de regiones de interés, junto con las predicciones de la red convolucional, únicamente detecten una región por objeto.

Esta técnica se basa en seleccionar los cuadrados delimitadores con las probabilidades más alta que ha predicho la red, el resto serán descartados.

Los pasos que sigue este algoritmo son, para empezar, descartar todas aquellas predicciones las cuales son menores a un determinado umbral, conocido como umbral de confianza. Este umbral es ajustable.

A continuación, el algoritmo calcula la superposición de las cajas restantes que pertenecen al mismo objeto. Los cuadrados delimitadores que contengan una alta superposición y pertenezcan a la misma clase serán los que se comparen.

La métrica de superposición se explica en el siguiente apartado, es conocido como intersección sobre unión. Intersection Over Union en inglés. (IOU).

El valor que define si la superposición entre dos cajas delimitadoras es conocido como umbral de Non-Maximum Suppression. Un ejemplo de la salida de este algoritmo se aprecia en la siguiente figura 3.11

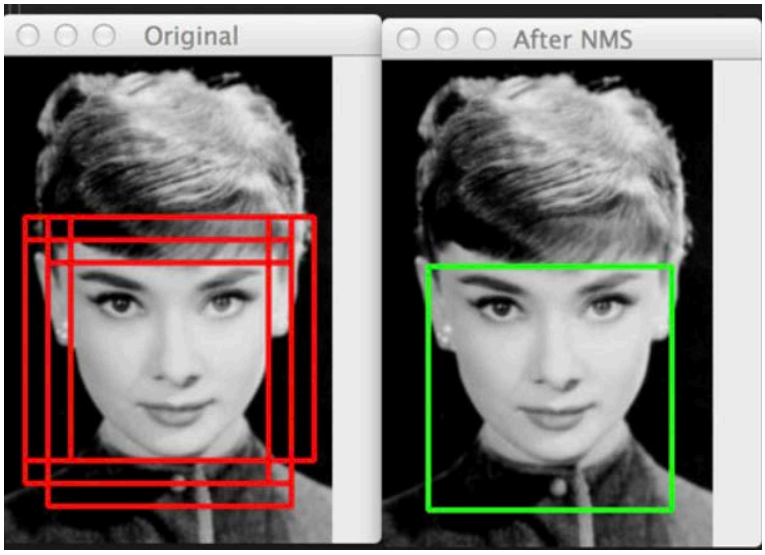


Figura 3.11 *Ejemplo de aplicación de Non-Maximum Suppression*

3.3.4 MÉTRICA

La métrica es la medida que utiliza el modelo para calcular el error. Es la medida que define la función de coste explicada en el segundo capítulo. Como ya se comentó en el capítulo 2, es diferente según el problema. A continuación, se explicará la medida utilizada para el proyecto a desarrollar.

A la hora de medir modelos de detección de objetos, se hace uso de distintas medidas, tales como fotogramas por segundo (FPS), el cual mide la rapidez de la red para detectar. así como la media de la precisión media, conocido en inglés como mean Average Precision (mAP), en el cual se mide la precisión de la red

La razón por la que se mide en este tipo de algoritmos la velocidad que tarda en detectar, es debido a que a son modelos que, al ser más complejos y necesitar mayores componentes para su funcionamiento, son más lentos de procesar.

Además, muchas implementaciones de este tipo de algoritmos son utilizados en tiempo real. Como ya se comentó previamente, el ejemplo de un coche capaz de conducir por si mismo, no solo ha de tener la capacidad de detectar a las personas, si no que, además, ha de ser rápido para poder reaccionar al momento.

Sin embargo, para el proyecto a desarrollar no es de gran necesidad y no será necesario que se priorice ante otros factores.

El mAP se calcula a partir de dos operaciones importantes que son necesarios para comprender el desarrollo de este, es por ello por lo que se explicará, primero, ambos conceptos, y, más adelante, se procederá a entender los pasos a seguir para el cálculo de esta medida de error.

La intersección sobre unión (IoU), mencionada en el apartado anterior, es una medida que mide la superposición entre dos cajas delimitadoras. Para ello cabe definir dos parámetros, la caja delimitadora real (B_{real}) y la caja delimitadora predicha por el modelo ($B_{predicha}$). Dicha medida se calcula de la siguiente manera: (fórmula 3.1)

$$IoU = \frac{\text{Breal} \cap \text{Bpredicha}}{\text{Breal} \cup \text{Bpredicha}}$$

Fórmula 3.1. Intersección sobre unión IoU

Para entender mejor la función véase la figura 3.12

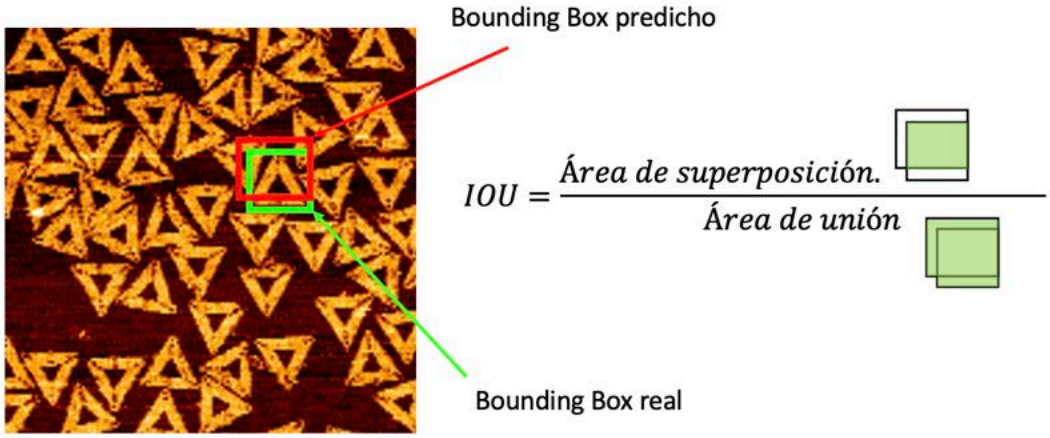


Figura 3.12 *Cálculo de IOU*

Como se puede comprobar el resultado de esta ecuación pertenece al rango [0,1]. Un valor igual a 0 supondría la inexistencia de superposición y 1 lo contrario. Es decir, a al hora de comparar los datos reales junto con los predichos por el modelo, cuanto mayor sea el número de IoU mejor es el algoritmo creado.

Para poder calcular este tipo de medida, es necesaria la previa definición de las posiciones de los bounding box. Es decir, se han de definir las coordenadas en las que se encuentran los objetos dentro de una imagen para que el algoritmo pueda comparar los resultados y así mejorar el modelo en la fase de entrenamiento.

Esta medida es utilizada para definir la predicción correcta. Esto quiere decir que se determina que una predicción es correcta si, definido un umbral, la IOU supera dicho valor, esto se define como verdadero positivo (TP=1). Y, se considera la predicción errónea, si es inferior, entonces se determina como falso positivo (FP=0). En la siguiente figura 3.13 se comprueba el cálculo matemático utilizado.

$$\frac{A_{int}}{A_{uni}} = \frac{B_I \times H_i}{B_P \times H_P + B_R \times H_R - B_{int} \times H_{int}} = \frac{(X_{R-MAX} - X_{P-MIN}) \times (Y_{P-MAX} - Y_{R-MIN})}{B_P \times H_P + B_R \times H_R - ((X_{R-MAX} - X_{P-MIN}) \times (Y_{P-MAX} - Y_{R-MIN}))} = \text{IOU}$$

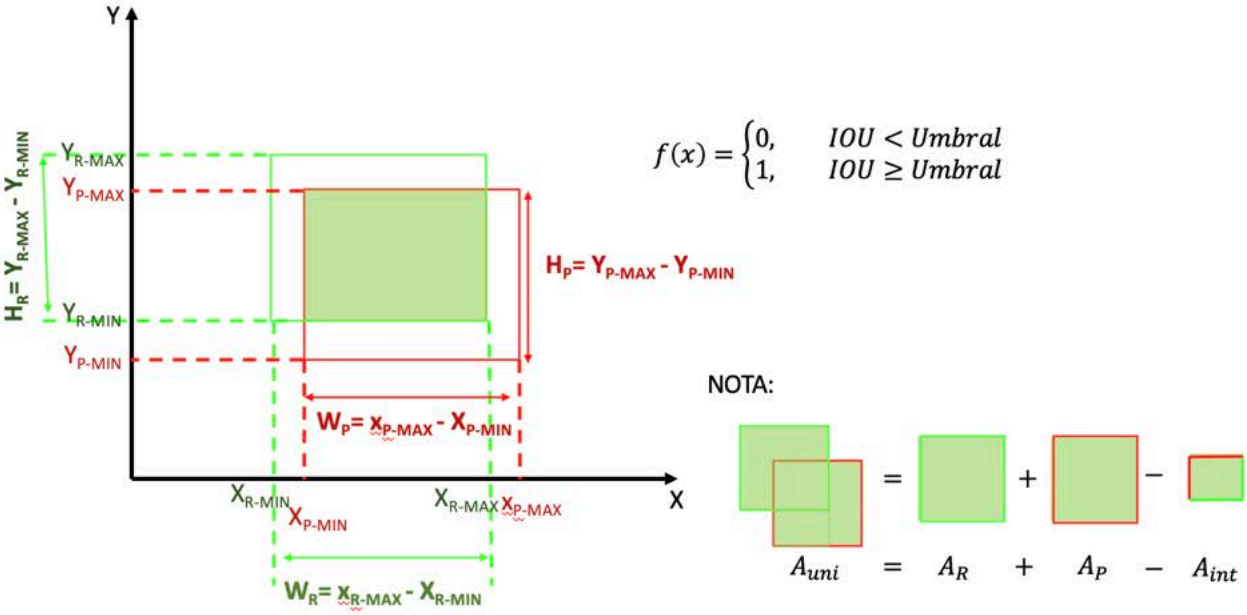
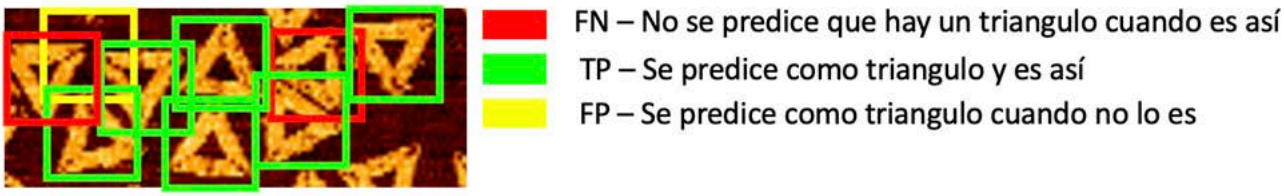


Figura 3.13 *Cálculo de FP y TP*

Finalmente, la siguiente operación que es necesaria se denomina Precision-Recall Curve [24]. Esta métrica es utilizada para evaluar los resultados predichos por un clasificador.

La precisión mide, sobre todos los resultados reales, cuantos se han acertado. En cambio, el recall mide sobre todos los resultados predichos cuantos se han acertado.

Las formulas son las siguientes (fórmula 3.2):



$$Recall(Re) = \frac{TP}{TP + FN} \;;\; TP(1 - Re) = FN \cdot Re \;;\; TP = \frac{FN \cdot Re}{(1 - Re)}$$

$$Precision(Pe) = \frac{TP}{TP + FP}$$

$$Pe = \frac{\frac{FN \cdot Re}{(1 - Re)}}{\frac{FN \cdot Re}{(1 - Re)} + FP} \;;\; Pe = \frac{\frac{FN \cdot Re}{(1 - Re)}}{FN \cdot Re + FP - FP \cdot Re} \;;\; Pe = \frac{FN \cdot Re}{FN \cdot Re + FP - FP \cdot Re}$$

Fórmula 3.2. Precisión y Recall

Una vez calculada la precisión y el recall para todas las clases, se procede a trazar la curva (PR Curve) de la siguiente manera. (Véase la figura 3.14)

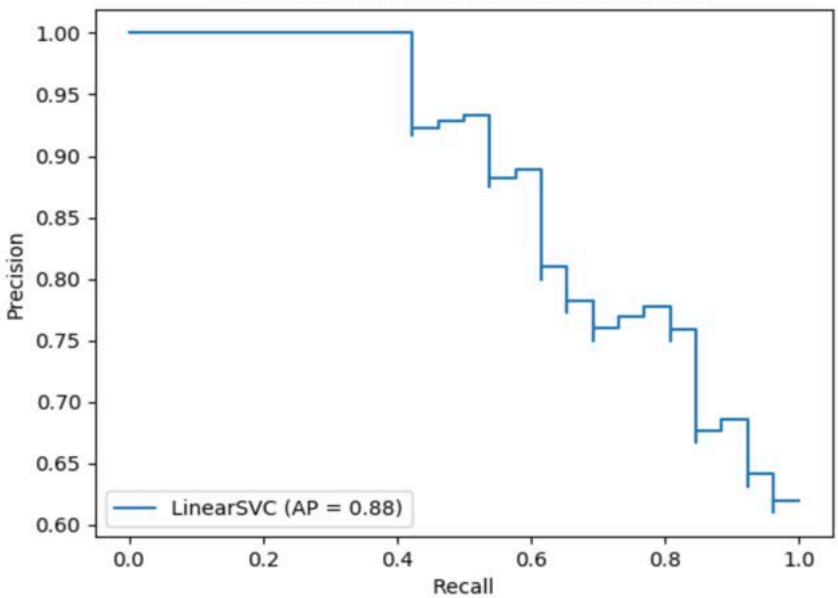


Figura 3.14 *Ejemplo Precision-Recall Curve [26]*

La curva PR es una medida óptima para el cálculo del rendimiento de un detector de objetos. La confidencia varía según la curva de cada clase. Un detector se considera bueno si la predicción se mantiene elevada mientras el recall se mantiene elevado [27].

Esto significa que si se varía el umbral de confidencia. Recuérdese que este umbral aparece en el NMS, en el momento en el que se descartan todas aquellas predicciones con un valor de probabilidad menor a éste. Se mantendría tanto el recall como la precisión elevada.

Sin embargo, un detector considerado pobre es aquel que ha de aumentar el valor de falsos positivos, lo cual implica una menor precisión, con el fin de mejorar el recall.

Una vez calculada esta curva, se procede al cálculo de la precisión media, average precision (AP) en inglés. Esta medida es únicamente el área bajo la curva PR. Una vez llegados a este punto. mAP es la media de todas las clases a detectar del área debajo de cada curva PR.

3.4 USO DE OPENCV

Como ya se ha comentado anteriormente, las redes neuronales convolucionales hacen uso de covoluciones con el fin de extraer las características propias de una imagen. Realmente aplican filtros a la imagen con el fin de determinar características del objeto a clasificar. Así, será posible la detección del mismo en una imagen diferente a pesar de su posición.

Con anterioridad al uso de redes neuronales, para detectar objetos, se hacía uso de una serie de algoritmos, los cuales aplican los filtros definidos por el desarrollador con el fin de encontrar patrones. Es decir, en vez de ser la máquina la que aprenda a determinar los valores de los pesos que han de utilizarse en los filtros, son los desarrolladores los que deciden los que se han de aplicar.

Este concepto coincide con lo comentado en el primer capítulo sobre el origen del machine learning, el cual surge de la necesidad de ser la máquina la que aprenda a crear las reglas óptimas del programa en vez de ser los humanos los que las definan.

El problema principal de las redes neuronales es el nivel de dificultad, la cantidad de datos necesarios para el buen funcionamiento, y la capacidad necesaria que ha de tener una máquina para poder entrenarlos de forma más rápida y eficiente.

Para este apartado, es necesario definir unos conceptos básicos sobre las imágenes. Como ya se comentó en el segundo capítulo, una imagen esta compuesta por tres matrices, las cuales representan los colores, rojo, verde y azul.

El hecho de que una imagen contenga una única matriz implica que es una imagen en escala de grises, es decir, está en blanco y negro. Sin embargo, existen otras maneras de definir una imagen, como por ejemplo HSV. Donde H define la tonalidad de la imagen, el color, S se

refiere a la saturación, y, finalmente, V hace referencia a la luminosidad. En la siguiente figura 3.15 se observa la representación del dominio HSV.

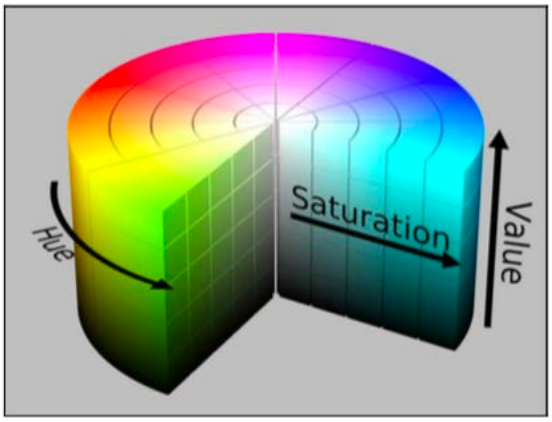


Figura 3.15 *Representación del dominio HSV [28]*

OpenCV [29] es un a librería de código que incluye una gran cantidad de algoritmos para el desarrollo de visión computacional. Mediante esta librería se podrán manipular las imágenes con el fin de encontrar soluciones para la detección de triángulos.

Sin embargo, no se realizará este método para este problema ya que los resultados obtenidos con esta práctica no suponen ninguna mejora con respecto a los explicados anteriormente.

Capítulo 4. MODELO DESARROLLADO

En este capítulo se procederá a comentar el desarrollo utilizado para dar solución al proyecto, para ello se han seguido los siguientes pasos:

1. Etiquetado de imágenes
2. Creación de un modelo de clasificación
3. Algoritmos de búsqueda de regiones con posibilidad de objetos
4. Rentrenamiento de una red neuronal ya existente
5. Pruebas con OpenCV

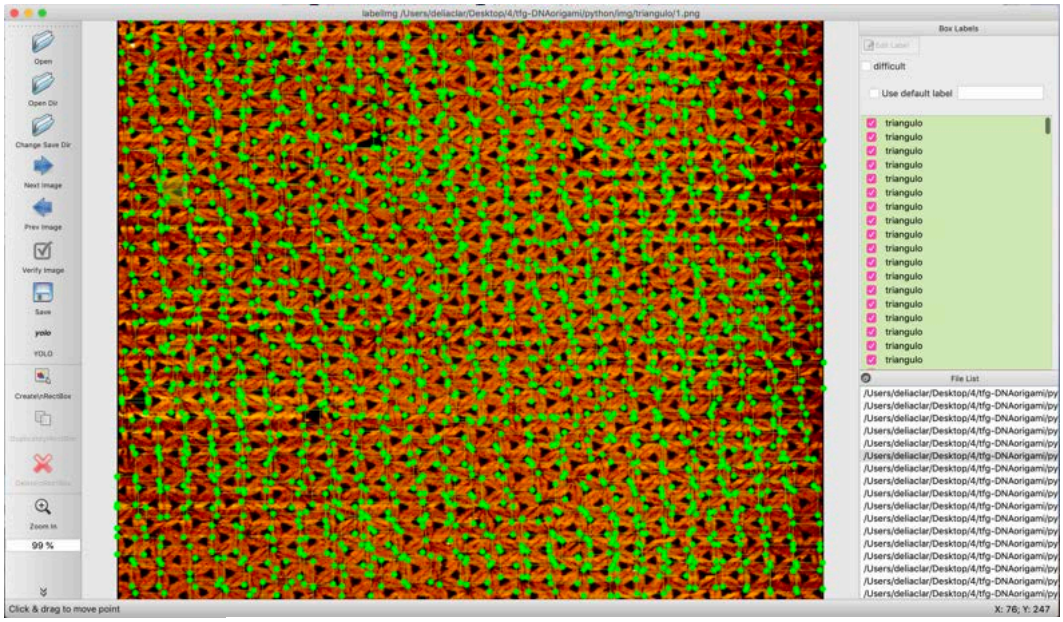
4.1 ETIQUETADO DE IMÁGENES

Para comenzar el desarrollo del proyecto es necesaria la preparación de los datos. Como ya se ha comentado previamente, para la creación de una red neuronal y su evaluación es necesario definir, en primer lugar, los datos de entrada junto con sus resultados.

El proyecto a desarrollar tiene como datos de entrada las imágenes con triángulos y, como resultado, la clasificación. Es decir, la probabilidad de que sea un triángulo o no junto con las coordenadas de la imagen donde se encuentra.

Tanto para la creación de una red de detección de objetos como para la creación del modelo de clasificación, es necesario definir el cuadrado en donde se encuentran los triángulos. Para ello se hace uso del programa LabelImg [19].

Mediante este programa se etiquetan todas las imágenes y se genera un archivo XML, en el cual aparece el nombre del archivo, la imagen, y todos los objetos que aparecen en ellas junto con sus posiciones, definidas como un cuadrado que rodea al objeto. Las coordenadas están representadas como (Xmin,Ymin,Xmax,Ymax). Un ejemplo de este proceso se observa en la siguiente figura 4.1



XML Ejemplo

```
<object>
    <name>triangulo</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
        <xmin>457</xmin>
        <ymin>515</ymin>
        <xmax>498</xmax>
        <ymax>556</ymax>
    </bndbox>
</object>
```

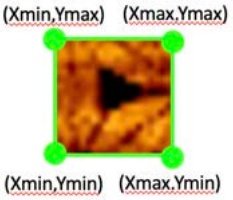


Figura 4.1 *Ejemplo de ejecución del programa LabelImg*

Una vez etiquetadas todas las imágenes, el siguiente paso que se seguirá a continuación será diferente según el proceso desarrollado.

4.2 CREACIÓN DE UN MODELO DE CLASIFICACIÓN

Para crear un modelo de clasificación, se han de definir, primero, los datos de entrada a la red. La idea de este punto es crear una red de clasificación de triángulos. Una vez creada, mediante algoritmos se decidirán regiones por donde pasar esta red por la imagen hasta encontrar los triángulos.

Para comenzar, se han de preparar los datos. Como ya se ha comentado anteriormente, se dispone de imágenes 1026x1026 en el que se encuentran una gran cantidad de triángulos. Para la creación de un modelo de clasificación es necesario disponer de una gran cantidad de figuras que contengan la clase a predecir. Para obtener estos datos se recortan las imágenes donde se encuentra cada triángulo.

Además, es necesario la creación de una clase en la cual no aparezcan triángulos, así la red aprenderá cuando es y cuando no lo es. Si solo se le enseña las imágenes de triángulos aprenderá que todas las imágenes contienen esta etiqueta. Para ellos se recortan las imágenes aleatoriamente y se eliminan aquellas que contengan triángulos. Dicho código se encuentra en el anexo II. CrearDatos.py

Antes de comenzar, cabe destacar la gran apariencia entre los datos definidos como positivos, los cuales son los que contienen los triángulos, y las negativas, ya que ambas

provienen de las mismas imágenes debido a que en ellas se utilizará el clasificador. En la figura 4.2 se observa las distintas imágenes utilizadas.

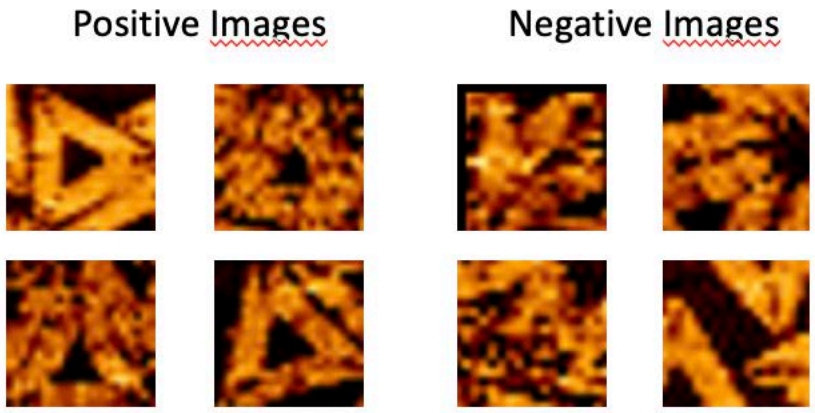


Figura 4.2 *Datos utilizados para la creación de un modelo de clasificación*

Una vez obtenidos los datos, se ha de definir la arquitectura de la red:

1. La función de activación de la última capa
2. Las capas del modelo y la cantidad
3. Los parámetros:

    a. El optimizador, que mide la variación del valor de los pesos con el fin de mejorar el error en la siguiente iteración
    b. La función de pérdida, la cual se encarga de calcular el error
    c. La métrica, la cual define como se mide la diferencia entre los resultados predichos y reales
    d. El número de iteraciones, la cantidad de veces que el algoritmo se entrena variando los pesos.

Para empezar, al tratar como datos de entrada imágenes las capas a utilizar serán convolucionales. Para encontrar la mejor arquitectura se han entrenado distintos modelos. Figura 4.3

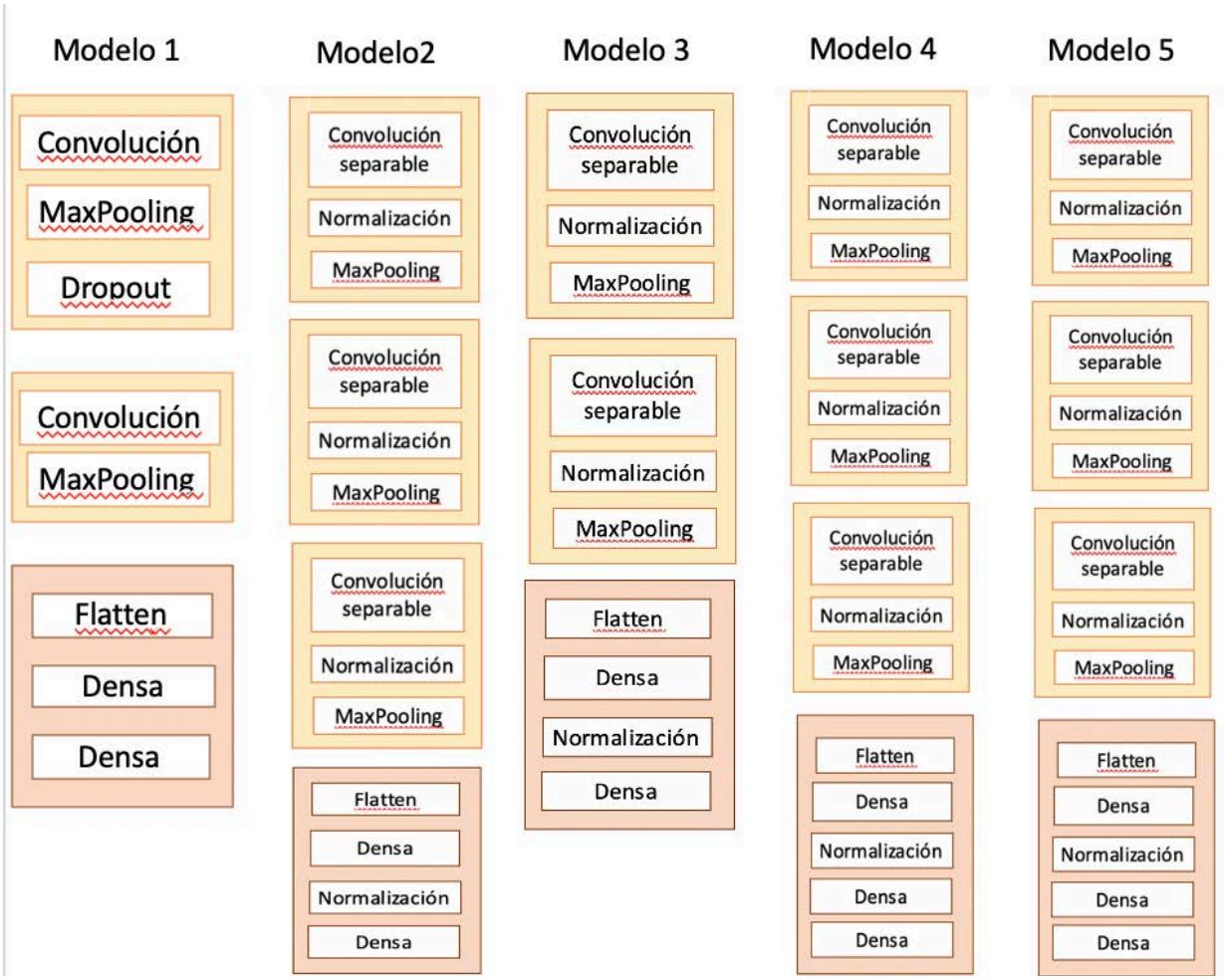


Figura 4.3 *Arquitecturas de los distintos modelos*

Para todas las arquitecturas del modelo, la función de activación a utilizar en la última capa. Al ser un problema de clasificación en el cual únicamente nos encontramos con 2 posibles resultados, positiva si es un triangulo y negativa si no, será la función Sigmoide, también conocida como función logística. Esta función fuerza a los resultados pertenecer al rango [0,1], para así, convertirlos en probabilidades. El resultado del modelo es, por tanto, la probabilidad de que en la imagen de entrada se encuentre un triángulo. La función que define dicha probabilidad es la siguiente (figura 4.4 y fórmula 4.1)

$$\sigma(z) = \frac{1}{1+ e^{-z}}$$

Fórmula 4.1 Sigmoide

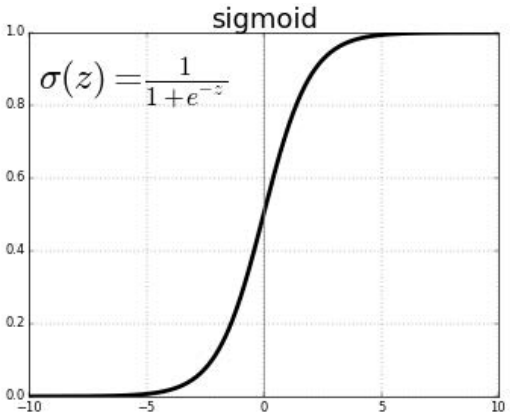


Figura 4.4 *Gráfica Función Sigmoide*

Tal y como se puede comprobar en la figura 4.3, el modelo 4 y el 5 tienen los mismos componentes. La diferencia entre ambos es la profundidad, como ya se ha comentado anteriormente. Al pasar una imagen por una red convolucional, disminuye en tamaño, pero aumenta en profundidad. Más adelante se observa el recorrido de una imagen sobre el modelo seleccionado.

Uno de los grandes problemas al entrenar un modelo es saber en que momento se ha de dejar de entrenar, con el fin de obtener unos resultados óptimos sin sobreajuste. Existen distintas técnicas para frenar el modelo en el momento en que deje de aprender, antes de caer en este tipo de problemas. Para ello se hace uso de los callbacks, los cuales se explican en el Anexo II Clasificacion.py. En el momento en el que son utilizados.

Para decidir el modelo a utilizar se han entrenado los distintos modelos, y se han obtenido los siguientes resultados

4.2.1 PRIMER MODELO

El primer modelo, que es el más sencillo, está compuesto por dos bloques convolucionales. En el primer bloque observamos un componente nuevo llamado Dropout [16]. Es utilizado únicamente en la fase de entrenamiento para evitar sobreajuste. Para ello, aleatoriamente pone a 0 un conjunto de valores. Exactamente un porcentaje de los datos. En este caso a un 20% de los datos. Esto evita que memorice los datos de entrada. Un ejemplo de la aplicación de este componente se puede ver en la siguiente figura 4.5.

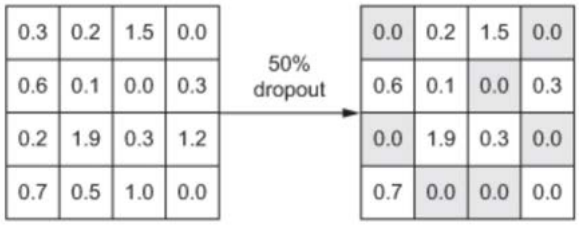


Figura 4.5 *Ejemplo de aplicación de Dropout al 50% [16]*

Los resultados de aplicar este modelo a todas las imágenes disponibles, tanto las imágenes utilizadas para entrenar el modelo como las definidas para probar el modelo con imágenes nunca vistas, figura 4.6.

Para entender dicha gráfica primero es importante definir:

1. TP: En una imagen se encuentra un triángulo y el modelo lo ha predicho bien
2. TN: En una imagen no se encuentra un triangulo y el modelo lo ha predicho bien
3. FP: En la imagen no se encuentra nada y el modelo lo ha predicho mal
4. FN: En la imagen se encuentra un triángulo y el modelo no lo ha encontrado

La matriz de confusión muestra, en el eje y los valores reales de las imágenes, y en el eje x los valores predichos. Dentro de cada caja se observan los porcentajes de cada caso.

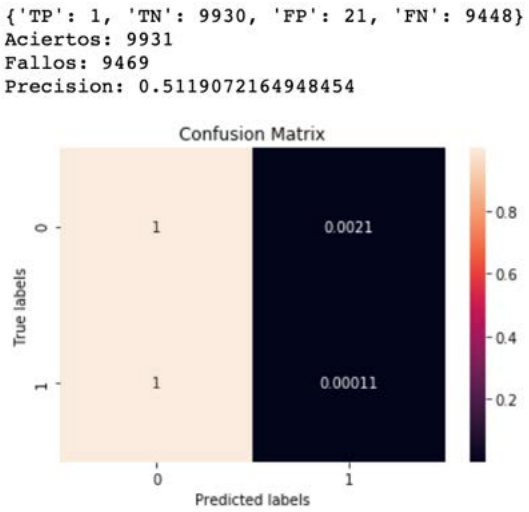


Figura 4.6 *Matriz de confusión de los resultados del modelo 1*

Tal y como se puede comprobar en la matriz de confusión, el modelo no ha sido lo suficientemente profundo, tras analizar los resultados se concluye que ha aprendido a definir a la mayoría de las imágenes negativas.

4.2.2 SEGUNDO MODELO

En el segundo modelo creado, aparecen dos nuevos componentes, convolución separable y normalización.

La convolución separable en profundidad es una manera de reducir los parámetros de los pesos a entrenar. Separa las convoluciones aplicadas a los distintos canales, es decir, las distintas matrices que representa cada una diferentes características de la imagen. Las analiza independientemente y, mezcla los canales de salida mediante otra convolución de 1x1, tal y como se muestra en la figura 4.7

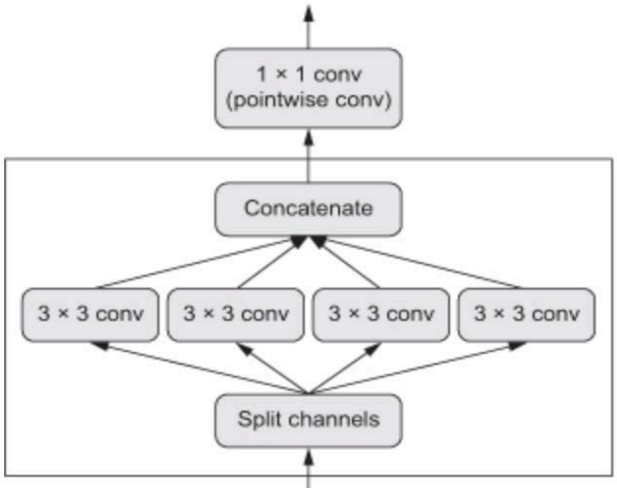


Figura 4.7 *Convolución separable en profundidad [30]*

El componente de normalización, "batch normalization" en inglés, normaliza los datos de salida de los distintos componentes. Ya que es una manera de hacer al modelo ver diferentes muestras similares entre sí, para así poder generalizar de cara a nuevos datos.

Este modelo se compone de tres bloques. Con el fin de entender los pesos resultantes del modelo se observan los histogramas de los mismos una vez entrenada la red, figuras 4.8, 4.9, 4.10, 4.11.

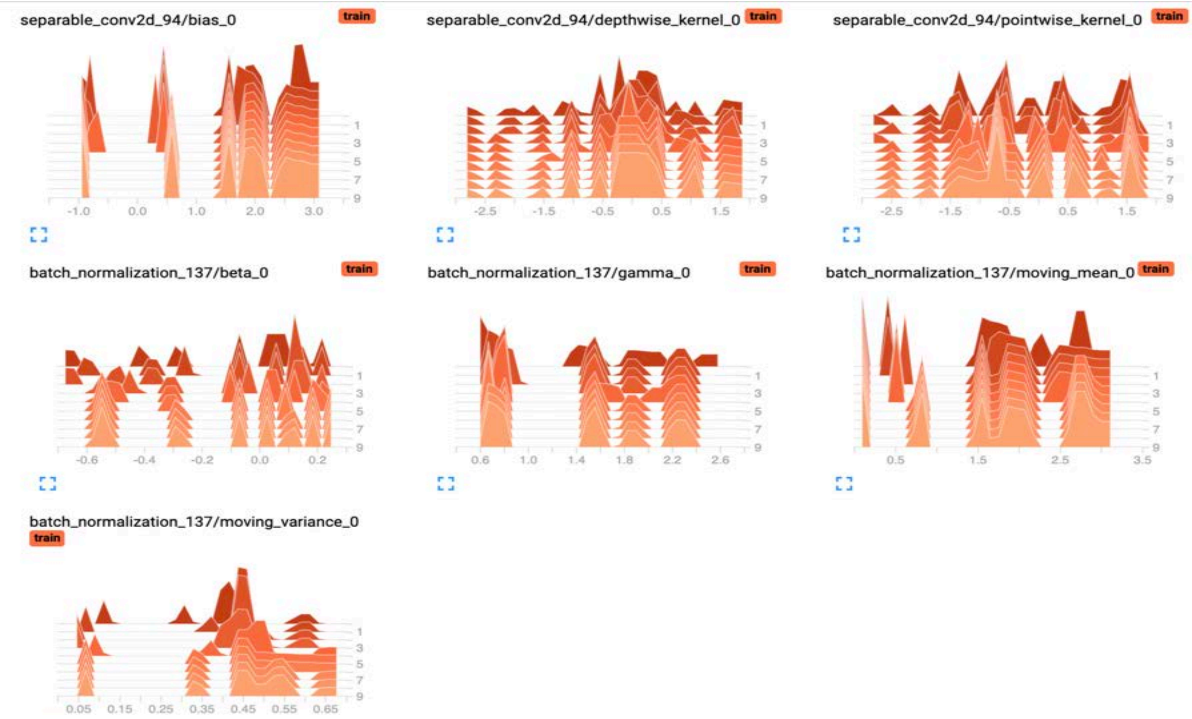


Figura 4.9 *Histogramas de los pesos del primer bloque del modelo*

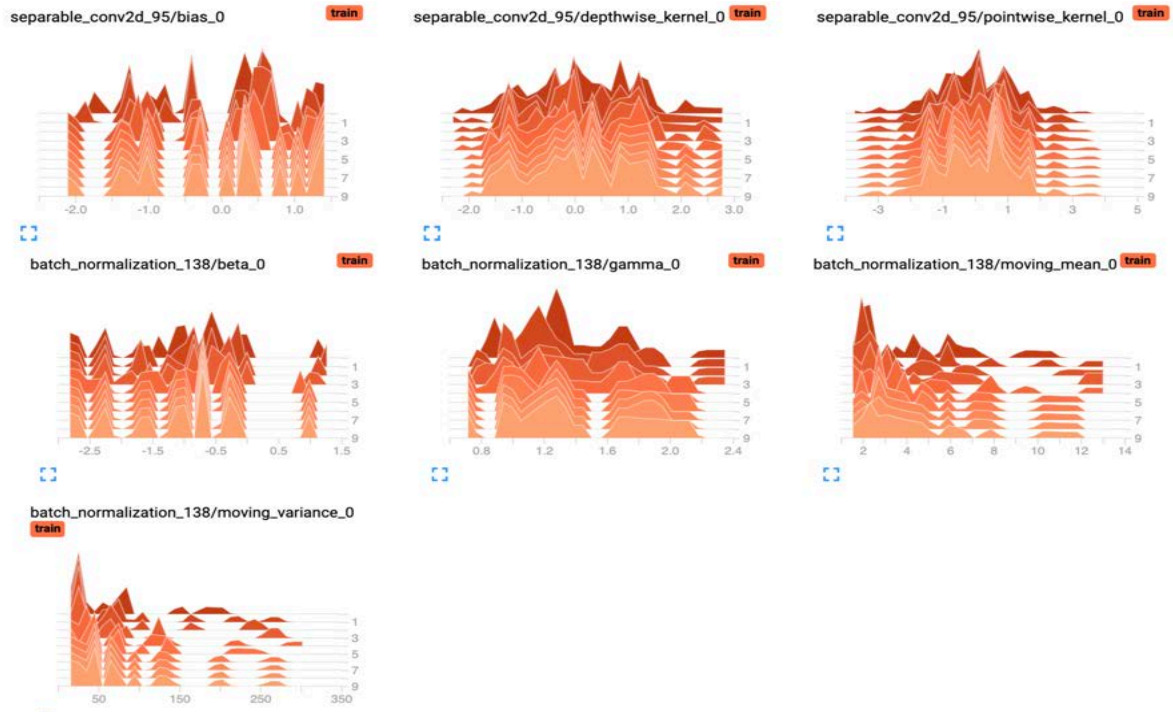


Figura 4.8 *Histogramas de los pesos del segundo bloque del modelo*

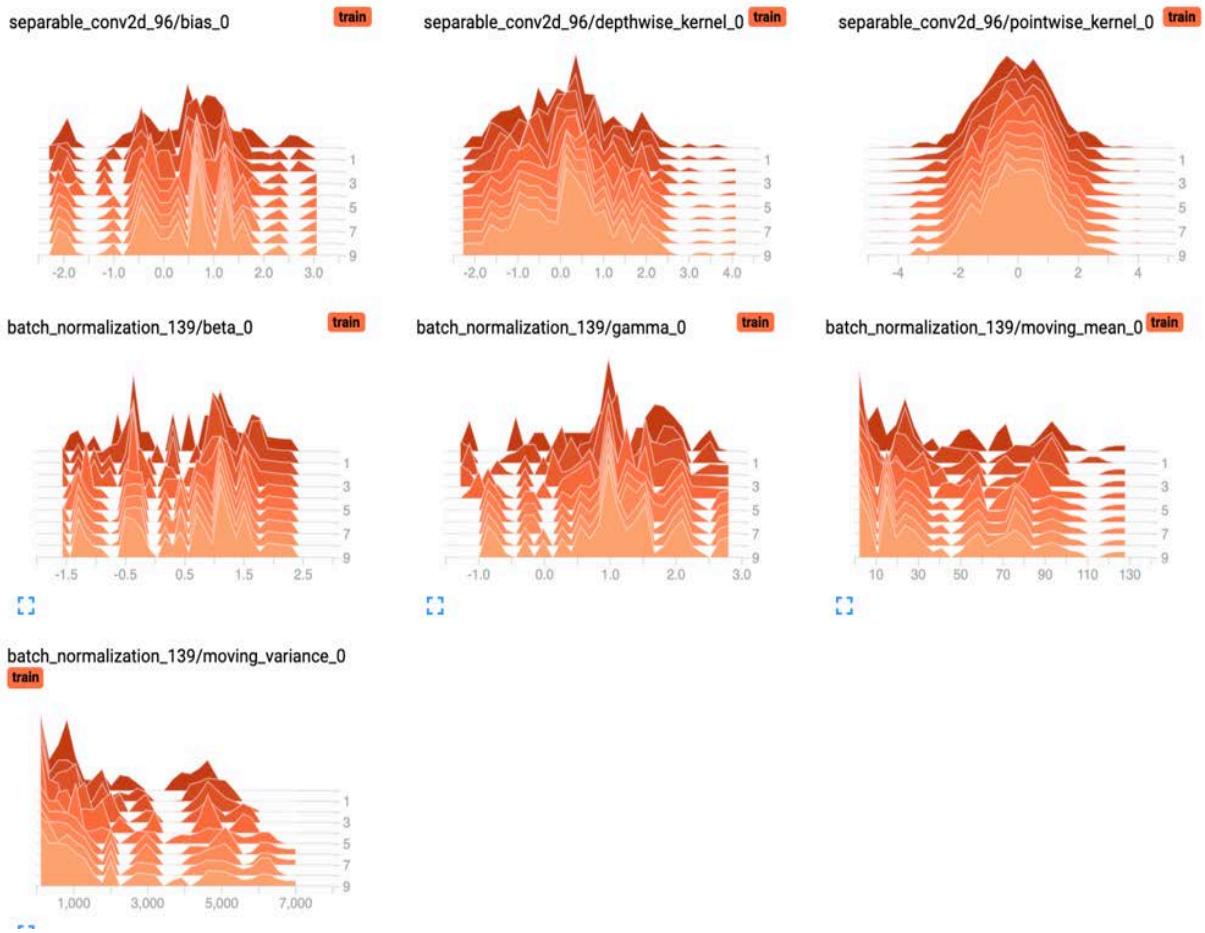


Figura 4.10 *Histogramas de los pesos del tercer bloque del modelo*

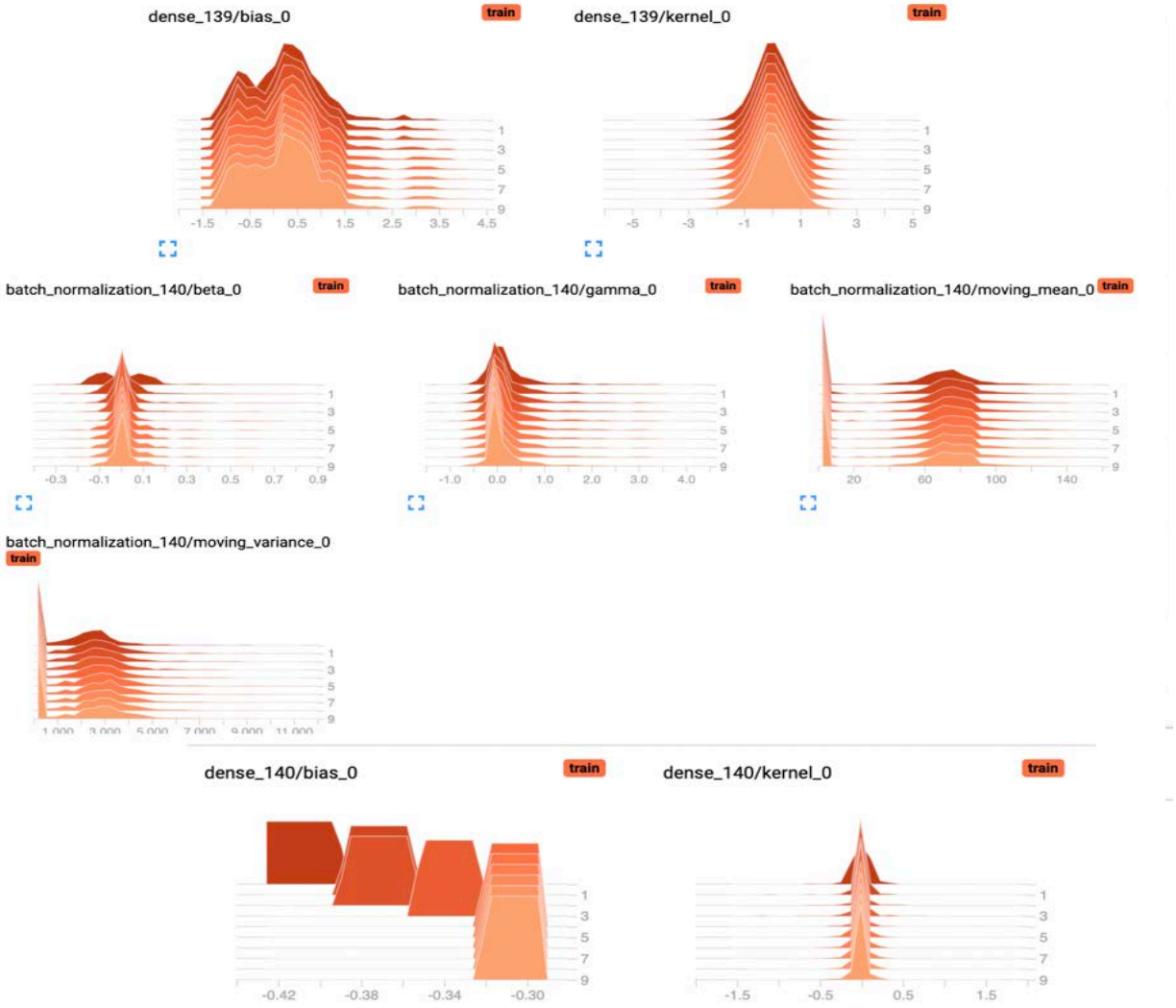


Figura 4.11 *Histogramas de los pesos del cuarto bloque del modelo*

Tal y como se comprueba en las gráficas mostradas, el componente de normalización realiza, en distintos pasos, el movimiento de la varianza y la media. Además, se puede comprobar que los valores de los pesos, gracias a la normalización, siguen, también, aproximadamente distribuciones normalizadas.

En el proceso de entrenamiento, tras realizar distintas pruebas variando los parámetros, se han obtenido los mejores resultados tras hacer uso de los siguientes:

1. Como función de perdida, se ha utilizado cross-entropy binario. Ésta es utilizada para medir la similitud de dos funciones de probabilidades, Q y P, cabe recordar que como resultado obtenemos probabilidades. La función es la siguiente fórmula 4.2:

$$H(Q,P) = \frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{2} p_j(x_i)\log_b q_j(x_i)$$
$$= -\frac{1}{m}\sum_{j=1}^{m}(y^j \log_b(y^{j*}) + (1-y^j)\log_b(1-y^{j*}))$$

Fórmula 4.2 Cross-entropy binario

Siendo:

- x_i la clase i, la cual puede ser positiva o negativa
- $p_j(x_i)$ la probabilidad real de que en la imagen j sea de la clase i. El valor es 1 o 0
- $q_j(x_i)$ la probabilidad predicha por el modelo de que en la imagen j contenga la clase i.
- $p_j(x_1) = 1 - y^j$ y $p_j(x_2) = y^j$ siendo y^j la representación de las etiquetas reales. 0 para la clase 1 y 1 para la clase 2
- $q_j(x_1) = 1 - y^{j*}$ y $q_j(x_2) = y^{j*}$ siendo y^{j*} la probabilidad de que la imagen pertenezca a la clase 2

2. Como optimizador se ha escogido Adam [31]. Para entender este optimizador es necesario explicar el descenso estocástico del gradiente (SGD). Es utilizado para encontrar mínimos por iteración sobre funciones diferenciables. Para ello, indica la dirección en la cual disminuye en mayor medida el gradiente para que los valores de los pesos se modifiquen dirigiéndose por dicha dirección. Sin embargo, no indica la cantidad a avanzar, se ha de definir lo que se conoce como ratio de aprendizaje. Adam resuelve este problema adaptando el ratio en función de la distribución de los parámetros. Si están muy dispersos aumentará.

3. La métrica utilizada es la precisión binaria, la cual mide las instancias que han sido clasificadas de forma correcta.

4. Como número de iteraciones realizados para entrenar la red, son escogidos por el propio modelo una vez deja de aprender cosas nuevas deja de entrenar. Este método es una manera de no entrar en sobreajuste.

Una vez realizado el entrenamiento se prueba el modelo y se obtienen los siguientes resultados. Figura 4.12

```
{'TP': 9449, 'TN': 9951, 'FP': 0, 'FN': 0}
Aciertos: 19400
Fallos: 0
Precision: 1.0
```

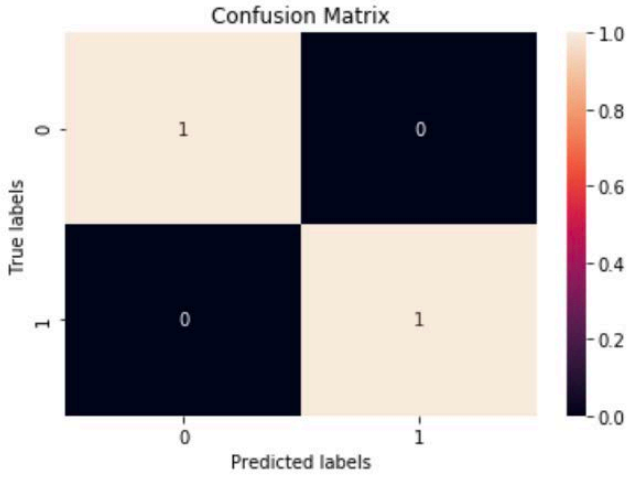


Figura 4.12 *Resultados obtenidos tras aplicar el segundo modelo*

Debido a que se obtienen los mejores resultados con este modelo, será utilizado para detectar los triángulos en las imágenes originales. En las siguientes gráficas se pueden comprobar las gráficas de error y precisión durante el entrenamiento. Figura 4.13 y Figura 4.14

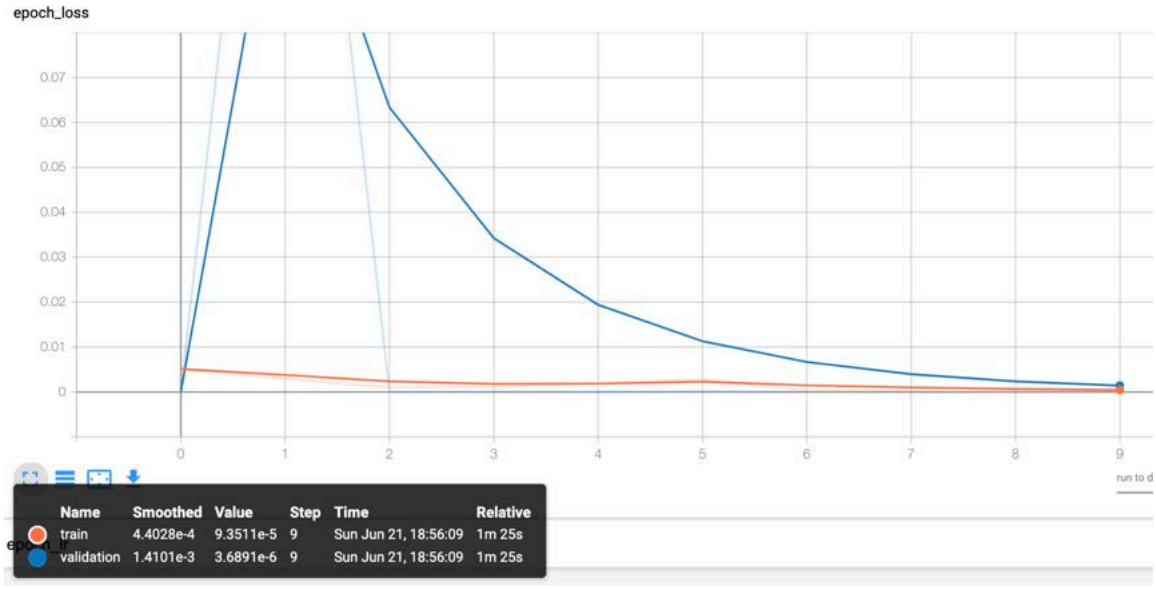


Figura 4.13 *Función de pérdida en el proceso de entrenamiento*

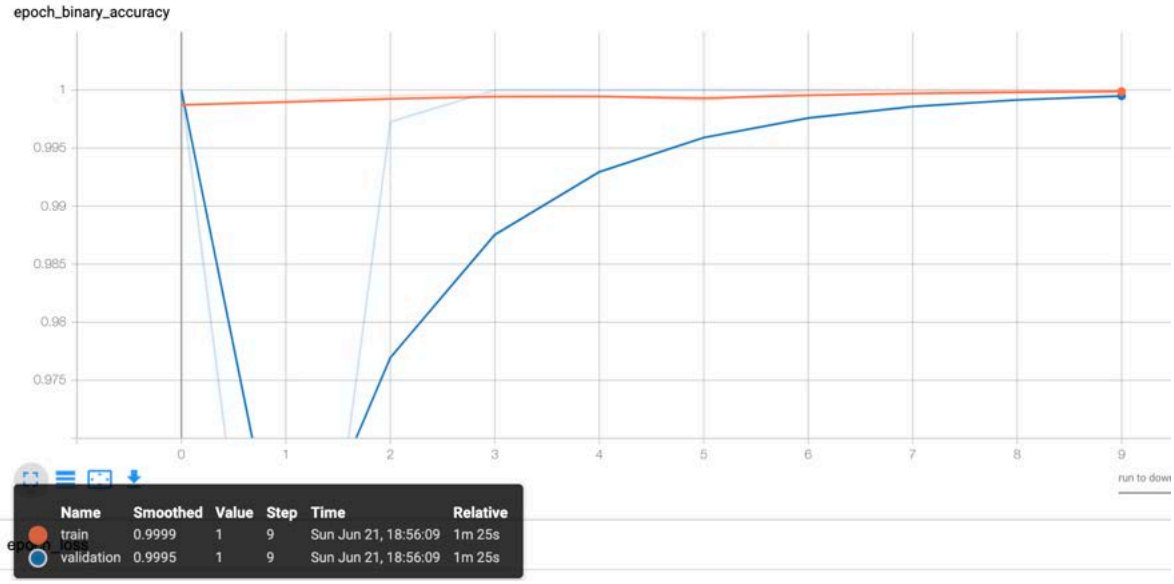
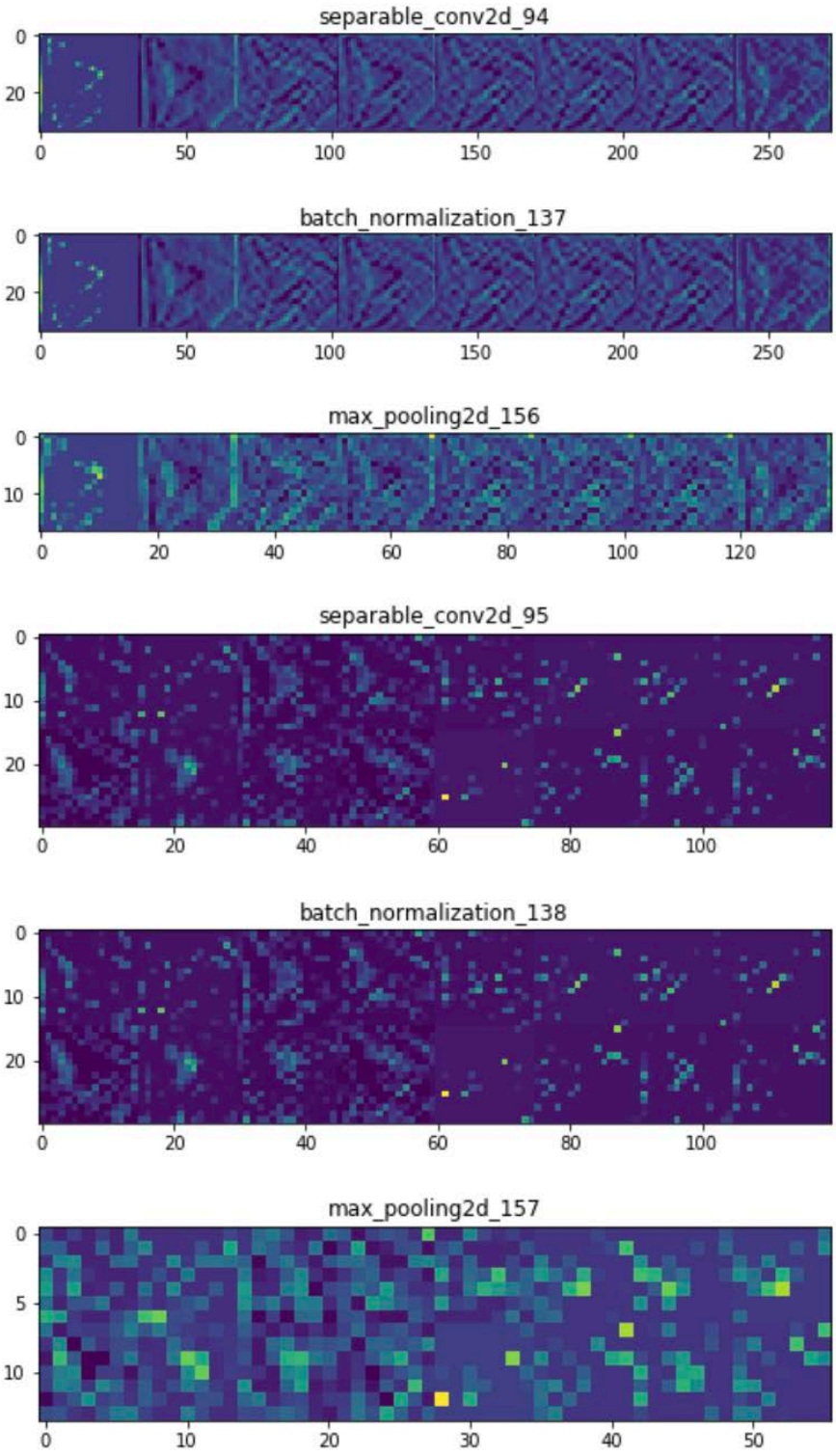


Figura 4.14 *Función de precisión en el proceso de entrenamiento*

Para comprender el funcionamiento de las redes convolucionales, a continuación, se muestra el recorrido de una imagen a través de estas capas. Figura 4.15


separable_conv2d_94
batch_normalization_137
max_pooling2d_156
separable_conv2d_95
batch_normalization_138
max_pooling2d_157

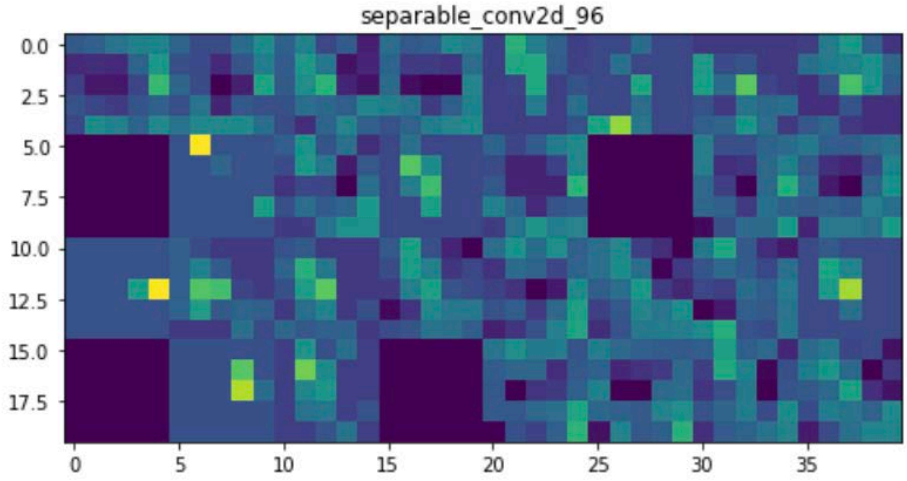


Figura 4.15 *Patrones de la imagen pasada por los bloques convolucionales*

Tras observar la imagen en las distintas fases, se comprueba como la normalización no varía los datos, mantiene la misma información. Además, se comprueba como el max pooling conserva únicamente los píxeles necesarios. A medida que avanza la imagen por los distintos bloques, aumenta en profundidad, tal y como se explicó anteriormente, y, por ello, deja de visualizarse el patrón de la imagen como en el inicio. En la última convolución, el conjunto de la gran cantidad de imágenes obtenidas, debido a la profundidad, es donde se han almacenado los patrones, los cuales, viendo de una en una no se aprecian.

El código de creación de este modelo se encuentra en el anexo II. Clasificacion.py. Un ejemplo de la aplicación del modelo a una imagen se observa en la figura 4.16

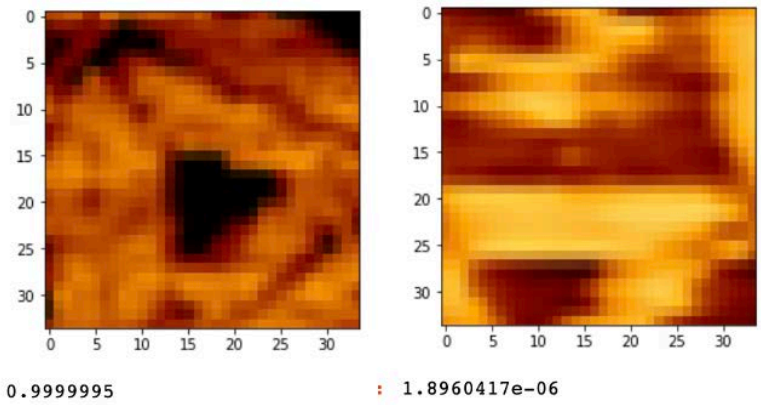


Figura 4.16 *Imágenes pasadas al modelo con los resultados obtenidos*

4.2.3 MODELOS RESTANTES

En este apartado se mostrarán los resultados de los modelos restantes. En la Figura 4.17 se encuentran las matrices de confusión de los modelos aplicados a todos los datos, tanto los datos con los que se han entrenado los modelos, como los datos que no se han visto nunca.

Con el fin de comprobar el sobreajuste se generan también las matrices de confusión únicamente para las imágenes nunca vistas por el modelo. Figura 4.18

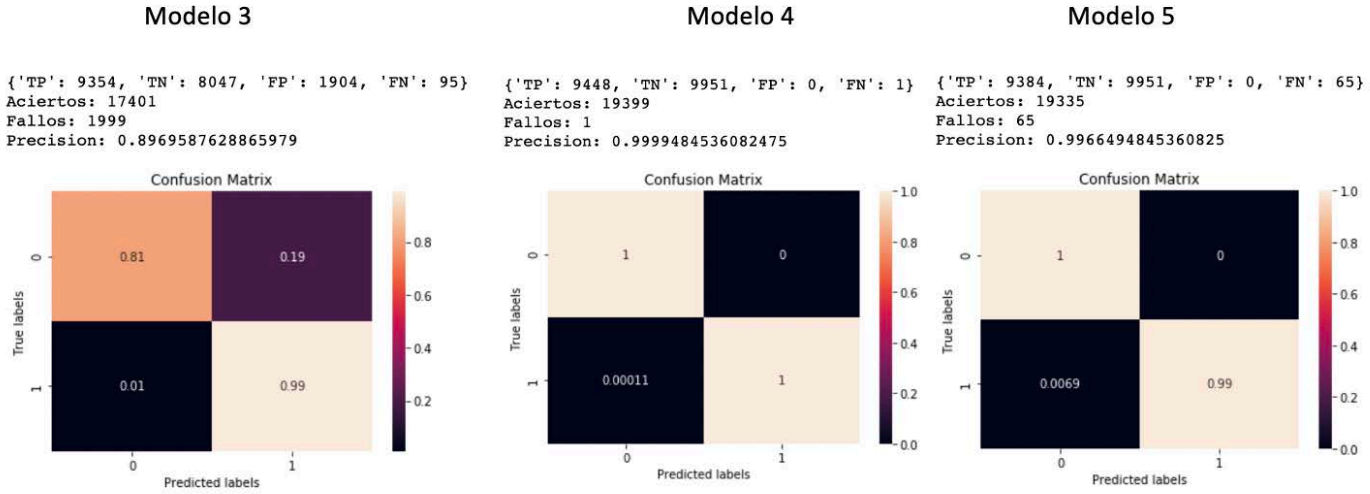


Figura 4.18 *Matrices de confusión de los modelos creados aplicados a todas las imágenes*

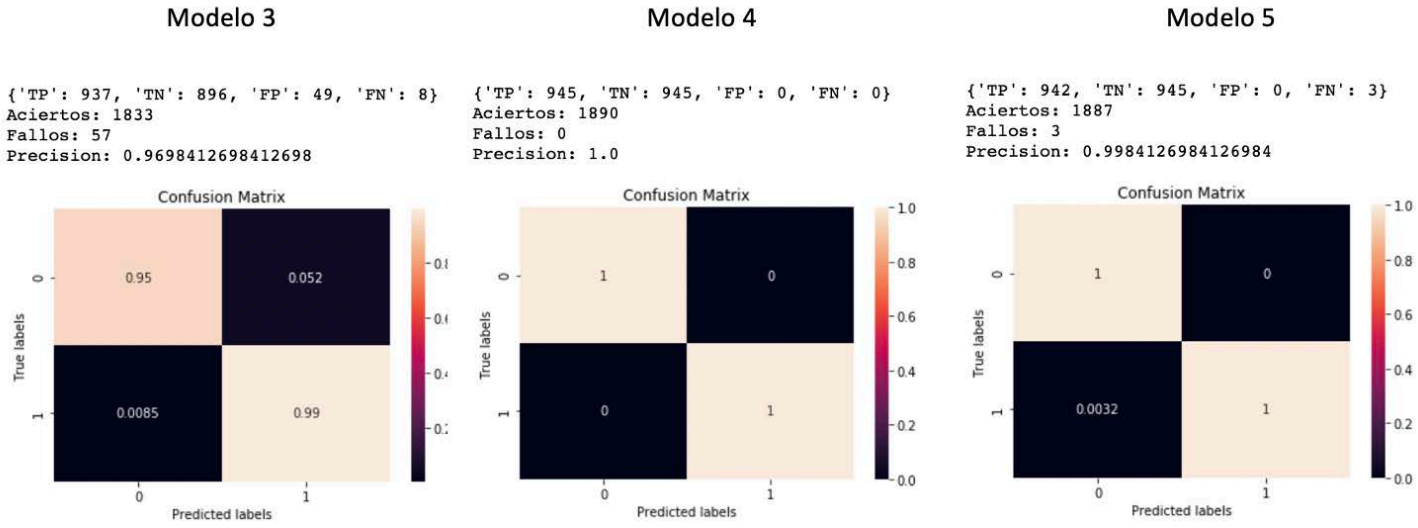


Figura 4.17 *Matrices de confusión de los modelos creados aplicados a imágenes nunca vistas por el modelo*

Tal y como se puede comprobar, no se ha producido sobreajuste en ninguno de los modelos creados.

4.3 APLICACIÓN DE ALGORITMOS DE DETECCIÓN DE REGIONES

Una vez escogido el modelo de clasificación, para detectar las posiciones de los triángulos dentro de una imagen es necesario pasar el clasificador por distintos lugares de la misma hasta que encontrar un fragmento en el cual, al pasarlo por el clasificador, de como resultado una probabilidad superior a un umbral definido.

Para ello se hace uso de dos técnicas. En la primera, el clasificador recorrerá la imagen, para ello se ha de definir:

1. El tamaño de los cuadrados con los que se recorrerán las imágenes. Se ha definido 34x34 ya que es el tamaño que han de tener los datos de entrada a la red.

2. El salto de recorrido del cuadrado que pasa por la imagen. Es decir, si el cuadrado recorre la imagen de píxel en píxel, se supondrían los cuadrados que obtuviesen los triángulos. Además, el proceso se haría lento. El valor escogido es 13.

3. El umbral, el valor del porcentaje el cual decide si se considera como triángulo o no, debido a la precisión del modelo se ha escogido un umbral igual a 0.9

Una vez decididos estos parámetros se observan los resultados, Figura 4.19

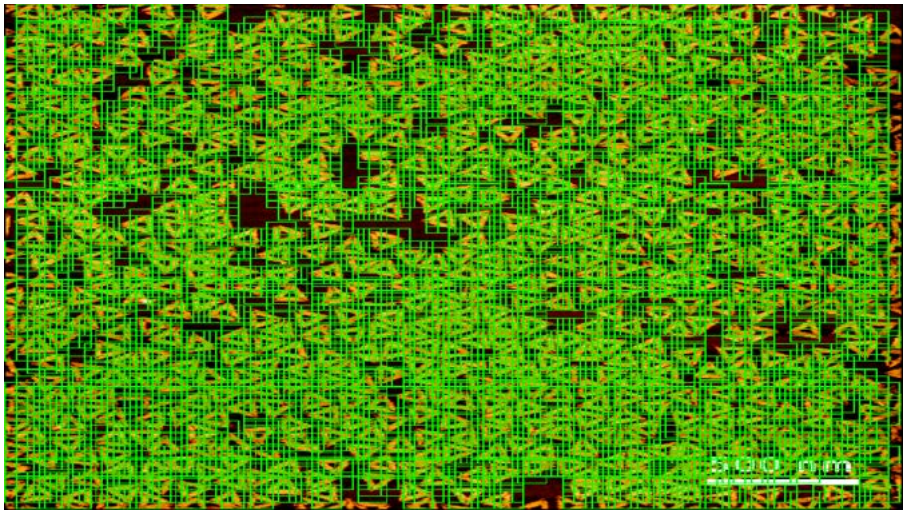


Figura 4.19 *Aplicación de recorrido del clasificador a través de la imagen*

Como se puede comprobar se superponen los cuadrados que rodean los triángulos, es decir distintas cajas rodean al mismo triángulo. Para solucionar este problema se hace uso del algoritmo de Non-Maximum Suppresion ya explicado en el capítulo anterior. Figura 4.20

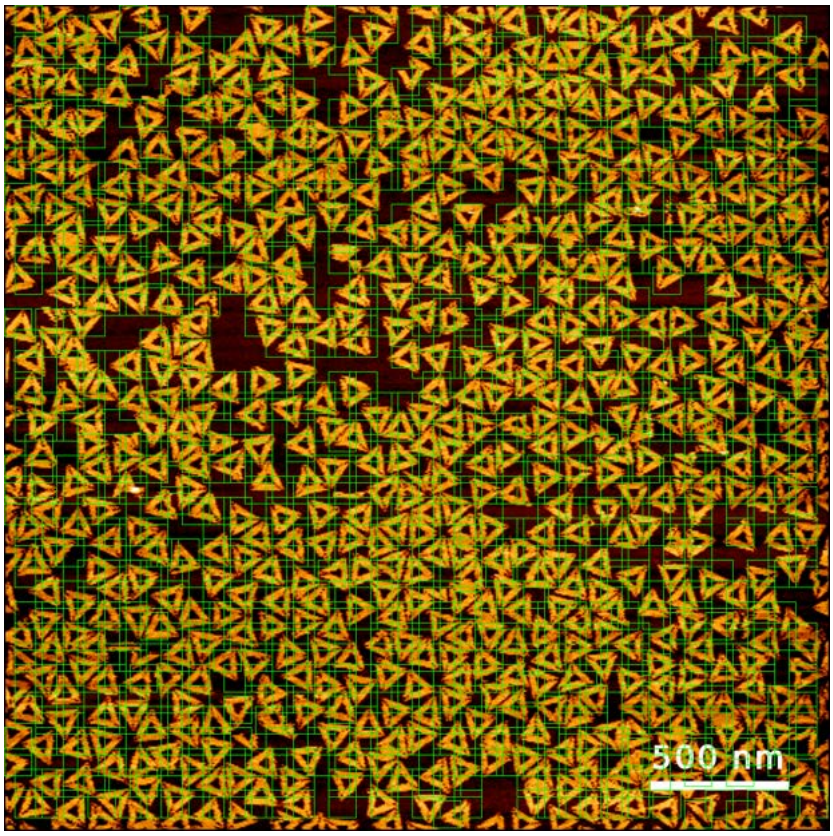


Figura 4.20 *Aplicación del clasificador a través de la imagen más uso de Non-Maximum Suppression*

Finalmente, otra manera de detectar objetos es, como ya se comentó en el capítulo anterior, a través del algoritmo de Selective Search. Sin embargo, se encuentran distintos problemas

Para empezar, es un algoritmo utilizado para encontrar objetos de distintos tamaños. En el problema descrito, los triángulos son aproximadamente del mismo tamaño. Además, el modelo de clasificación creado está entrenado para recibir como dato de entrada imágenes de 34x34. El algoritmo detectará regiones de mayor tamaño los cuales obtendrán no solo uno, si no varios triángulos. Es por ello por lo que el clasificador dará como resultado probabilidades altas a regiones que no deberían ser seleccionadas. En la figura 4.21 se observa un ejemplo de la aplicación de este algoritmo tras pasar las regiones detectadas por el clasificador.

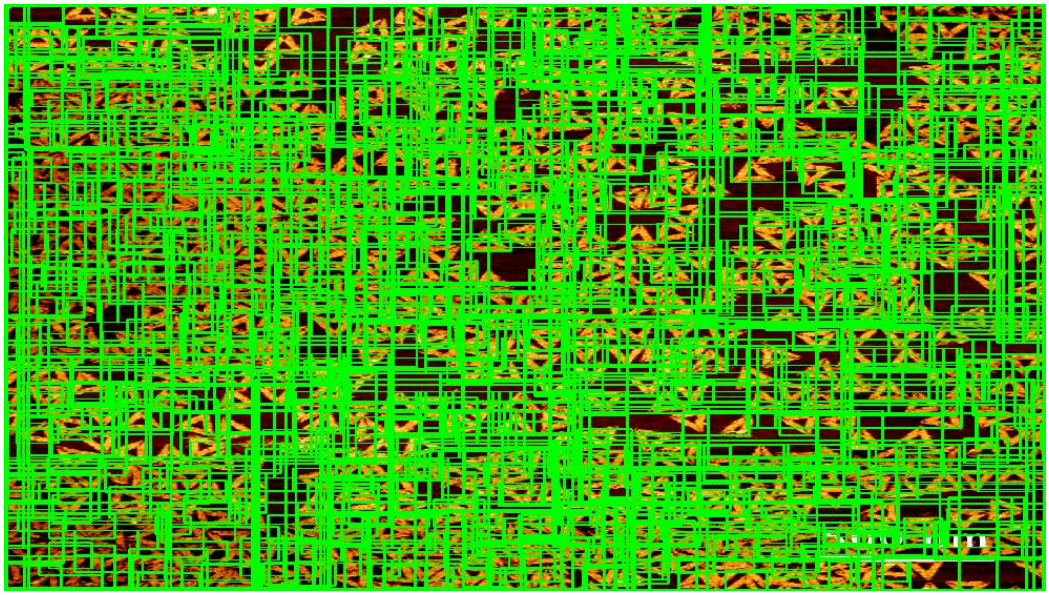


Figura 4.21 *Aplicación del clasificador sobre regiones detectada por el algoritmo Selective Search*

4.4 REENTRENAMIENTO DE UN MODELO

Una vez visto el desarrollo de una red que detecte las coordenadas de los triángulos hecha de cero, se procede a otra técnica que ya ha sido explicada previamente, el reentrenamiento de una red neuronal ya existente.

Existen tres tipos de redes neuronales de detección de objetos que son los más conocidos, entre ellos nos encontramos con YOLO (You-Only-Look-Once) [32], Faster R-CNN (Faster Region-Based Convolutional Network) [33] y SSD (Signle Shot Detector) [34].

YOLO es la red más reciente. Presenta un mayor rendimiento en cuanto a la velocidad a la que procesa las imágenes, permitiendo así, su uso en tiempo real. Sin embargo, en el problema a desarrollar no es una prioridad. La razón por la que se ha descartado esta red es debido a la dificultad en detectar correctamente los objetos pequeños. [35]

En cuanto a SSD y Faster R-CNN, se ha elegido reentrenar el modelo Faster R-CNN debido a que, a pesar de ser más lenta, la localización de objetos es más precisa que SSD. Y, como ya se ha comentado, la ejecución en tiempo real no es una necesidad en este problema. Además, esto no significa que sea lento, ya que, requiere únicamente 0.2 segundos por imagen.

4.5 FASTER R-CNN

Antes de comenzar con la implementación, es necesario conocer la arquitectura de esta red, para así comprender su funcionamiento y los parámetros escogidos. Faster R-CNN surge en 2016 por Ross Girshick [33].

En vez de usar el algoritmo de Selective Search, tal y como se ha implementado en el apartado anterior, una red es utilizada para predecir propuestas de regiones como parte del proceso de entrenamiento, llamado red de propuestas de región. Conocido en ingles como Region Proposal Network (RPN).
Las regiones predichas se vuelven a formar utilizando una capa de agrupación de región de interés, conocido como Region of Interes (ROI), propuesta de regiones candidatas por imagen. Posteriormente, son utilizadas para clasificar la imagen que se encuentre dentro de esta región y, así, predecir tanto el valor del porcentaje de clasificación junto con los cuadros delimitadores que rodeen el objeto, tal y como se observa en la figura 4.22 A continuación, se analizará su arquitectura.

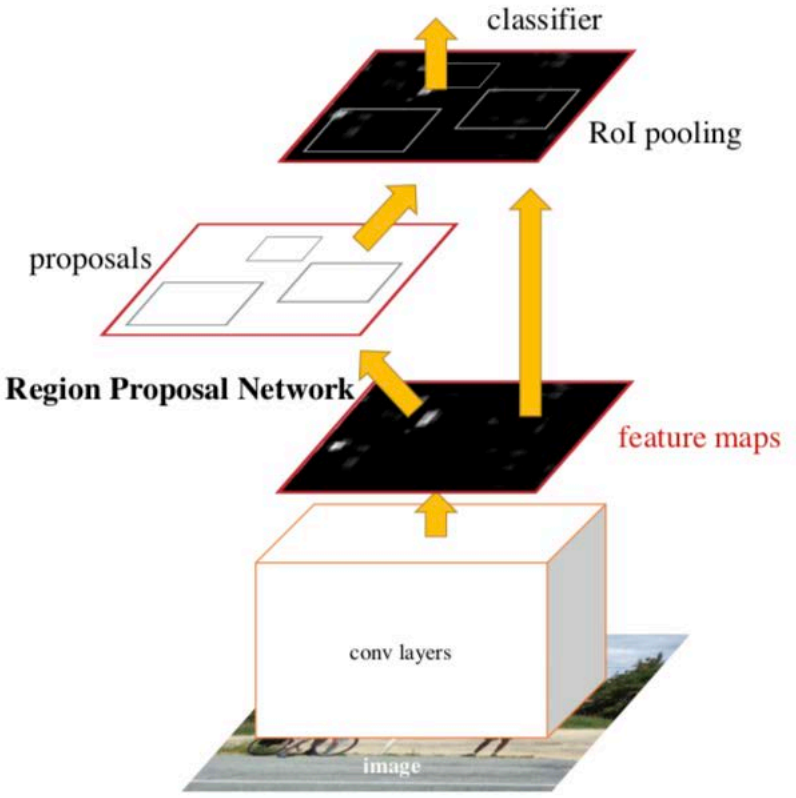


Figura 4.22 *Proceso de Faster R-CNN [33]*

4.5.1 ARQUITECTURA

La arquitectura de este modelo esta compuesta fundamentalmente por dos redes principales. Para empezar, el primero componente es una red de propuesta de región, esta parte del modelo es una red convolucional que propone regiones de interés, basándose en un extractor de mapas de características que se considerarán para futuras investigaciones. Como datos de salida de esta parte se encuentra el porcentaje de objetividad, en inglés conocido como "objectness score", es decir el porcentaje de lo que se encuentre dentro de la región sea un objeto, y la localización de la caja. Véase la figura 4.23.

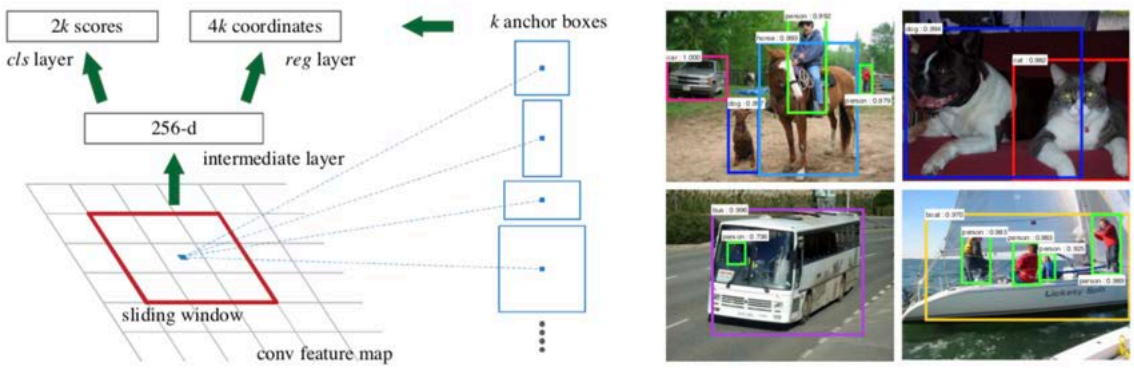


Figura 4.23 *Region Proposal Network [33]*

El segundo gran componente de este modelo es la red Fast R-CNN [36], perteneciente al mismo grupo que el modelo de Faster R-CNN, el cual es una mejoría de éste.

Primero, consiste en una red base de extracción de características, se hace uso de una red convolucional ya entrenada para extraer estas representaciones, se hace uso de la red de clasificación VGG-16 [37], que, como ya se conoce, la técnica de utilizar una red ya entrenada es quitando su última capa en la cual se predicen las clasificaciones. Además, hace uso de una capa de ROI, en la cual se extraen regiones de interés de un tamaño fijo.

Como capa de salida, finalmente, contiene dos capas completamente conectadas, se encuentra, entonces, el clasificador que hace uso de la función de activación softmax, el cual se explicará con más detalle en el siguiente párrafo, que predice las probabilidades de pertenencia del objeto a cada clase y, finalmente, una regresión que predice los cuadros delimitadores. La arquitectura completa puede observarse en la siguiente figura 4.24.

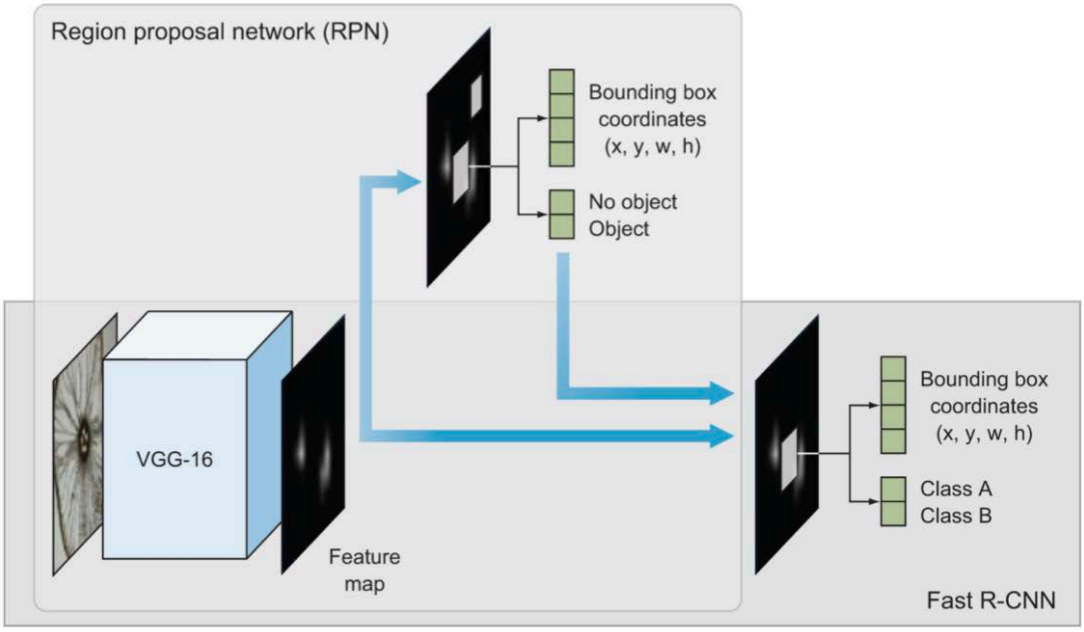


Figura 4.24 *Arquitectura Faster R-CNN [38]*

La función softmax, mencionada recientemente, es una función generalizada de la función de sigmoide. Es utilizada para obtener las probabilidades de clasificación cuando se tienen más de dos clases. Cabe destacar que a pesar de que el proyecto a desarrollar únicamente posee una clase, la red Faster R-CNN que se utilizará para la implementación de éste, ha sido previamente entrenada para un conjunto de datos compuesto por 91 clases.

Esta función de activación fuerza a los resultados predichos de la red neuronal sumen 1 y sean superiores a 0. La función es la siguiente, fórmula 4.3:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Fórmula 4.3 Función Softmax

Tal y como se observa en la figura 4.24. La imagen entra en la red reentrenada de VGG-16. La imagen de salida, que, como ya se conoce, son características sacadas de la original, es enviada, paralelamente, a dos componentes de la arquitectura de Fast R-CNN. Por una parte, son enviados a la red de propuesta de región, en el cual se determina en que puntos de la imagen puede encontrarse un objeto. Por otro lado, se envían a la capa de agrupaciones de región de interés con el fin de extraer ventanas de características de tamaño fijo.

La salida de estos componentes se transfiere a dos capas completamente conectadas, una para la clasificación de los objetos y otra para la predicción del cuadro delimitador, y así es como se obtienen las localizaciones finales junto con el porcentaje del objeto.

Para recapitular, la arquitectura de esta red esta compuesta por:

1. Una red base ya reentrenada, VGG-16, para extraer características de los objetos, la arquitectura de esta red ya ha sido mostrada en capítulos anteriores

2. Una red de propuestas regiones, el cual se compone de: (Figura 4.23)
    a. Una capa completamente convolucional 3x3 con 512 canales. Esta, se desliza por las imágenes de características pasados por el primer componente

    b. Dos capas convolucionales 1x1 en paralelo:

        i. Una capa de clasificación que predice si existe o no un objeto dentro de una región, se define si es un primer plano, si se encuentra un objeto, o un fondo si no. No se busca clasificar los objetos, únicamente determina si en la región puede hallarse uno para su futura investigación
        ii. Capa de regresión que predice la caja delimitadora que rodea al objeto. Más adelante se explicará con mayor detalle

3. Capas completamente conectadas, en esta parte, tal y como se observa en la figura 4.24 toma dos entradas, las imágenes o mapas de características salidas de la red convolucional VGG-16, junto con la salida de la red de propuestas de regiones. Clasifica entonces las regiones seleccionadas y devuelve como resultado dichas predicciones junto con las coordenadas, (x, y, anchura, altura) siendo x e y las coordenadas de la imagen que definen el centro de la caja que los rodea.

4.5.2 PREDICCIÓN DEL CUADRADO DELIMITADOR

Como se acaba de comentar, una caja delimitadora está definida por una tupla (x, y, anchura, altura). Sin embargo, a la hora de predecir las coordenadas en la imagen del centro de esta caja, supone definir unas reglas para no predecir valores que sobrepasen los límites de las coordenadas de la imagen.

Para que esto no ocurra se definen unas cajas de anclas, conocidas en inglés como anchor boxes [33]. La capa de regresión definirá entonces el desplazamiento de estas cajas. Para ello, se hace uso de una ventana deslizante.

Para generar, propuestas de posibles regiones, se desliza una pequeña red, específicamente una convolución 3x3 tal y como se ha comentado en la arquitectura de la red de propuesta

de región, y, posteriormente, le siguen dos convoluciones 1x1, sobre los mapas de características convolucionales, datos obtenidos por la red reentrenada VGG-16.

En cada ubicación por la que pasa la ventana deslizante, se predice simultáneamente múltiples regiones, el valor máximo de propuestas se conoce como k. Éste número define la salida de la capa de regresión como $4 \cdot k$ coordenadas pertenecientes a k regiones. Recuérdese el formato de las coordenadas (x, y, altura, anchura), y la capa de clasificación de objetividad proporciona $2 \cdot k$ probabilidades sobre la estimación de si se encuentra un objeto o no en la región. Se implementa en este caso la función softmax de dos clases, una que defina probabilidad de ser un primer plano y otra de ser un fondo.

Las k proposiciones representan relativamente k cajas de referencia, las ya mencionadas anchor boxes. Esta ancla está centrada en la ventana deslizante, la cual se desliza por toda la imagen, y está asociado a unas determinadas escalas y relaciones de aspecto, este proceso se observa en la figura 4.23

Es importante entender que, la ventana deslizante, se desliza por un mapa de características en vez de por la imagen de entrada a la red, lo cual hace que sea un modelo más rápido, ya que el mapa de características es el que sale de la red VGG-16 y es menor a la imagen de entrada al modelo.

4.5.3 FUNCIÓN DE PÉRDIDA

Esta red hace uso de una función de pérdida de tareas múltiples, en el cual combina las pérdidas de clasificación junto con la regresión del cuadro delimitador. La función de pérdida para una única imagen se define como: (fórmula 4.4)

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \frac{\lambda}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*)$$

Fórmula 4.4 Función de pérdida de tareas múltiples

Donde:

1. N_{cls} y N_{reg} son términos de normalización, uno para la pérdida de clasificación y otro para la regresión de clasificación. Los valores utilizados son 256 y 2400 respectivamente

2. p_i es la probabilidad predicha de que la caja ancla sea un objeto

3. p_i^* es un valor binario, es 1 si en la caja ancla realmente hay un objeto y 0 si no

4. t_i son los cuatro parámetros predichos que definen el cuadro delimitador que rodea al objeto. (x,y,w,h). Donde x e y son las coordenadas en la imagen del centro del cuadrado predicho. w la anchura y h la altura.

5. t_i^* son los cuatro parámetros reales que definen el cuadro delimitador
6. $L_{cls}(p_i, p_i^*)$ la función de pérdida de clasificación la cual se define como: (fórmula 4.5)

$$L(p_i, p_i^*) = -p_i^* \cdot \log p_i - (1 - p_i^*) \cdot \log(1 - p_i)$$

Fórmula 4.5. Función de pérdida de clasificación

7. $L_{reg}(t_i, t_i^*)$ es la función de pérdida de regresión, en la cual se hace uso de la función conocida como smooth L1, la cual se define a continuación, fórmula 4.6.

$L_{reg}(t_i, t_i^*) = L1_{smooth}(t_i - t_i^*)$;

$$L1_{smooth}(x) = \begin{cases} 0.5x^2, & if \quad |x| < 1 \\ |x| - 0.5, & otherwise \end{cases}$$

Fórmula 4.6. Función de pérdida de regresión

La función se observa en la siguiente gráfica. Figura 4.25.

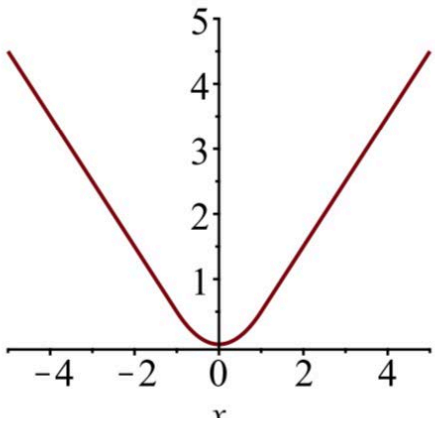


Figura 4.25 *Gráfica L1 smooth [39]*

4.5.4 REENTRENAMIENTO

A pesar de encontrar distintos modelos de Faster R-CNN entrenados con distintos conjuntos de datos [40], se ha escogido el que ha sido ya entrenado con el conjunto de datos propuesto por Microsoft COCO: Common Objects in Context [41], el cual está compuesto por 91 tipos de objetos diferentes con un total de 2.5 millones de etiquetas en 328 mil imágenes.

Una vez escogido el modelo, antes de comenzar, se ha de proceder al etiquetado de imágenes con las que se volverá a entrenar la red, tal y como se ha descrito en el apartado 4.1. Una de las técnicas explicadas en el capítulo anterior, hace referencia al aumento de datos. Y, a pesar de contener 9450 etiquetas de triángulos, provienen únicamente de 15 imágenes. Por eso mismo, se hace uso de esta técnica.

Sin embargo, al tener ya los archivos xml de las imágenes junto con todas las etiquetas, las técnicas de aumento de datos han de modificar también estos valores. Es decir, se hacen transformaciones en la imagen y, las coordenadas de x e y, variarán acorde a éstas. El código desarrollado para la creación de nuevas imágenes junto con sus respectivos xml se encuentra en el anexo II, DataAugmentationForFasterRCNN.py.

Una vez creados los nuevos datos se dividen en dos carpetas, una que definirá las imágenes utilizadas para el entrenamiento de la red, aproximadamente el 80%, y otra para la validación.

Las imágenes con sus respectivos xml, se deberán transformar en archivos denominados TFRecord, un formato binario que sirve para almacenar registros de datos para entrenamiento de redes neuronales pertenece a la librería de TensorFlow. [42]. Para ello, primero se pasan los archivos a un archivo CSV, código de este proceso se encuentra en el anexo II, XMLToCSV.py. Una vez pasados a un archivo csv se procede al cambio a TFRecords, el código se encuentra en internet [43].

A continuación, se echa un vistazo a los parámetros que se han definido para reentrenar la red. Una vez descargado el modelo a reentrenar [40], específicamente el archivo faster_rcnn_resnet101_coco, se procede a la modificación de ciertos parámetros. Para ello, se abre el archivo faster_rcnn_resnet101_coco.config [40]. Se puede comprobar el archivo en el anexo II en el cual se han subrayado los valores modificados y se han explicado como comentario el significado de los parámetros, subrayado en gris oscuro.

Los valores del modelo más importantes se resumen en:

1. Número de clases, como ya se ha comentado anteriormente, esta red está creada para detectar 91 tipos de clases diferentes. Debido a que únicamente es necesario buscar triángulos, se le asigna el valor 1.

2. La dimensión máxima de la imagen debe ser 1026, debido a que el conjunto de datos a utilizar es 1026x1026

3. Características de las cajas de anclas:

    a. Se definen las diferencias de tamaños de las cajas de anclas con respecto a la ventana deslizante

b. También las relaciones de aspecto.
c. Pasos que da la ventana deslizante de altura y anchura

4. Del primer componente, como ya se ha comentado previamente, se predicen regiones con posibles objetos en su interior, el umbral de IoU, explicado en el capítulo 3, para esta fase, es de 0.6.

5. Del primer componente únicamente pueden definirse 1100 regiones para un futuro análisis

6. Una vez pasados por el clasificador, únicamente pueden detectarse 1000 cajas delimitadoras.

7. Para hacer uso de la técnica de Non-Maximum Suppression definida en el capítulo 3, se considera que dos cuadrados rodean al mismo objeto si la función de IoU supera el umbral de 0.5

Una vez modificados estos parámetros, son necesarios otros dos archivos guardados en una carpeta llamada configuración. los cuales son label_map.pbtxt y labels.txt. Estos ficheros se encargan de definir las clases del modelo. En la figura 4.26 se observan estos archivos.

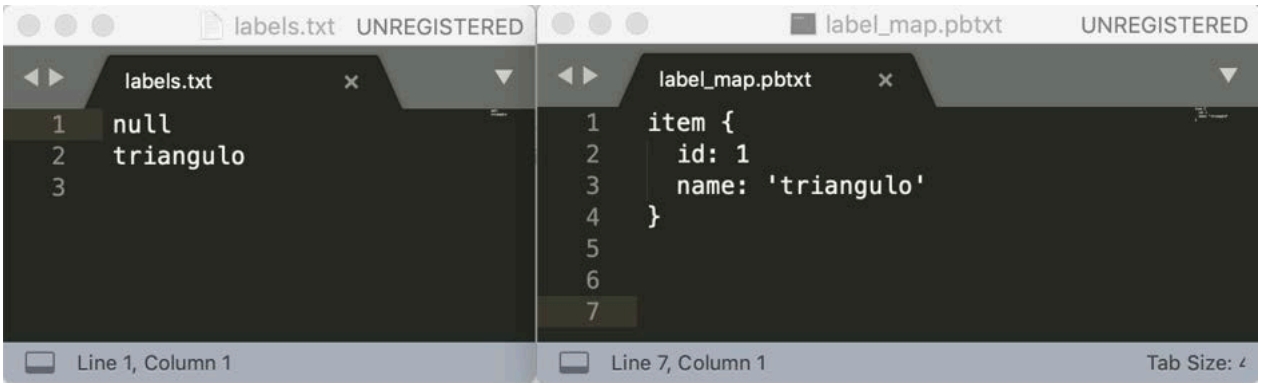


Figura 4.26 *Ficheros de la configuración del modelo*

Una vez creados estos ficheros se procede al entrenamiento del modelo. Para ello, es necesario el código encontrado en el Github de la Api de TensorFlow [44], donde el fichero para entrenar la red train.py se encuentra específicamente dentro de la carpeta legacy [45].

El siguiente paso será la ejecución del archivo train.py, para ello, como parámetros de entrada se indican dos valores, para empezar, se indica en que carpeta se encuentra el modelo, es decir, el archivo faster_rcnn_resnet101_coco.config, en el cual define la ruta hacia los datos de entrenamiento y los de validación mediante los ficheros TFRecords

definidos anteriormente más los parámetros del modelo. Finalmente, se indicará donde se va a guardar el modelo una vez entrenado.

En cuanto finalice la ejecución del entrenamiento, se generan, en la carpeta definida en la ejecución del entrenamiento, unos archivos con la información necesaria del modelo para predecir las clases definidas, para poder hacer uso de él se procede a congelar el modelo, es decir, en el momento de entrenamiento el modelo generado sirve para poder volver al entrenamiento en el punto en el que se quedó, en vez de volver a empezar, para poder utilizarlo se ha de ejecutar el fichero export_inference_graph.py [46].

Al ejecutar este fichero es necesario indicar los siguientes parámetros de entrada:

1. –input_type: image_tensor, implica que nuestros datos de entrada son de tipo imagen
2. –pipeline_config_path ruta donde se encuentra el fichero que define el modelo faster_rcnn_resnet101_coco.config.
3. –trained_checkpoint_prefix ruta donde se encuentra el modelo reentrenado a congelar.
4. --output_directory ruta donde se guardará el modelo congelado para su uso

Una vez congelado el modelo ya puede ejecutarse y consiguientemente, detectar los triángulos de la imagen, en la API de TensorFlow se encuentran distintas funciones ya creadas para dibujar los cuadros delimitadores en las imágenes. El código que carga el modelo y genera las predicciones se ha basado en distintos códigos ya generados y se han modificado para el problema en particular [47] [48]. Se puede comprobar el código generado en el anexo II. FasterRCNN.py, se ha comentado el desarrollo del código para un mejor entendimiento.

Los resultados obtenidos se detallarán el siguiente capítulo. Sin embargo, se puede observar un ejemplo de los triángulos detectados en una imagen en la figura 4.27

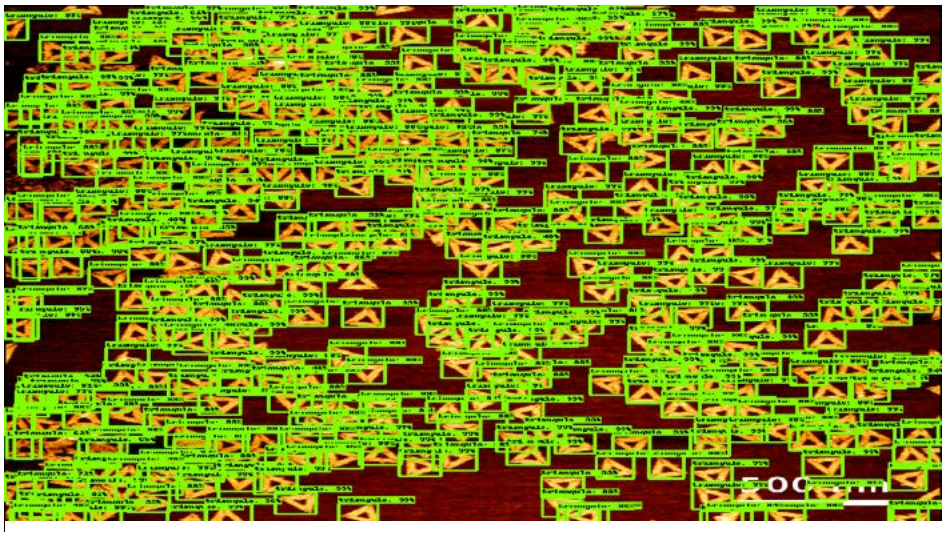


***Figura 4.27**. Resultado de reentrenamiento de Faster R-CNN*

Capítulo 5. ANÁLISIS DE RESULTADOS

Tal y como se ha comentado previamente, en este capítulo se analizarán los resultados obtenidos por cada implementación desarrollada.

5.1. RESULTADOS DEL CLASIFICADOR

Una vez implementado el código y creado un modelo de clasificación que recorra la imagen, se compararán los resultados predichos junto con los reales. Para ello se hace uso del siguiente código, detallado en el anexo II ResultadosClasificador.py, en el cual se hace uso de la clase ObjectDetectionEvaluation sacada de la API de Tensorflow dedicada a la detección de objetos [49].

Esta clase se encarga de calcular, la media de las precisiones medias de cada etiqueta (mAP), la precisión media de cada clase (AP) y el recall de cada clase.

Se recuerda que la precisión mide, sobre todos los resultados reales, cuantos se han acertado y el recall mide sobre todos los resultados predichos cuantos se han acertado.

La precisión media (AP) es el área de la curva que relaciona la precisión y el recall. Y la media de las precisiones medias (mAP) es la media de la precisión media de todas las clases a detectar por el modelo.

Se obtienen los siguientes resultados (Tabla 5.1), en los cuales se aplica la evaluación al conjunto de todas las imágenes y, separadas, a dos imágenes nunca vistas por el modelo.

Datos	**mAP**	**Precisión Media (AP)**	**Recall**
Global	0.105	0.105	0.250
Imagen1	0.157	0.157	0.289
Imagen2	0.118	0.118	0.268

Tabla 5.1. *Resultados del Clasificador*

Como se puede comprobar la mAP y la precisión media (AP) por clases es la misma ya que en el problema descrito únicamente existe la etiqueta triángulo. Además, a pesar de que visualmente, (véanse las siguientes figuras 5.1 y 5.2), puede parecer que predice de manera óptima, los resultados obtenidos no son los adecuados y, por tanto, no dan solución al problema definido.

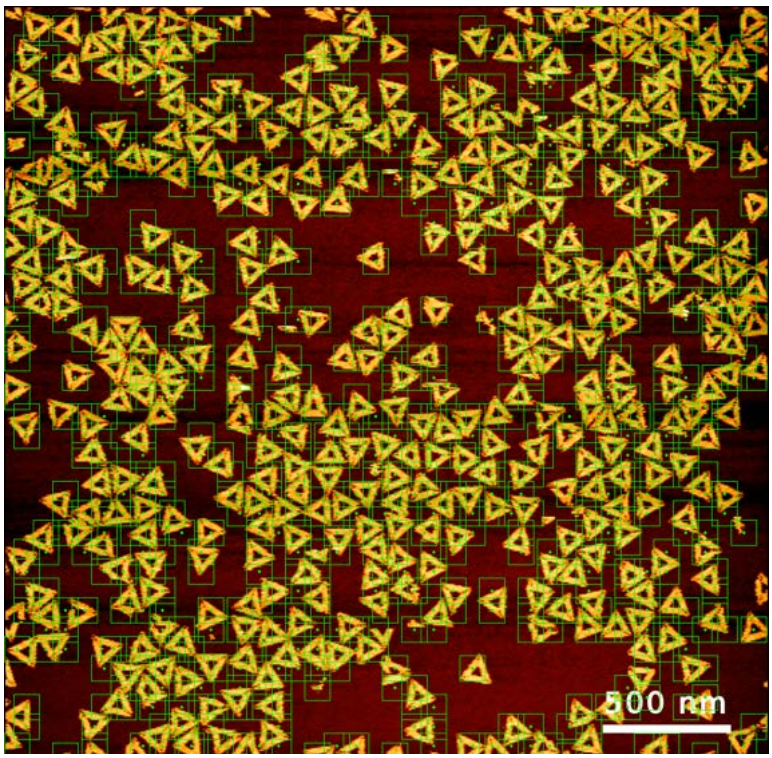


Figura 5.1 *Aplicación del modelo de clasificación*

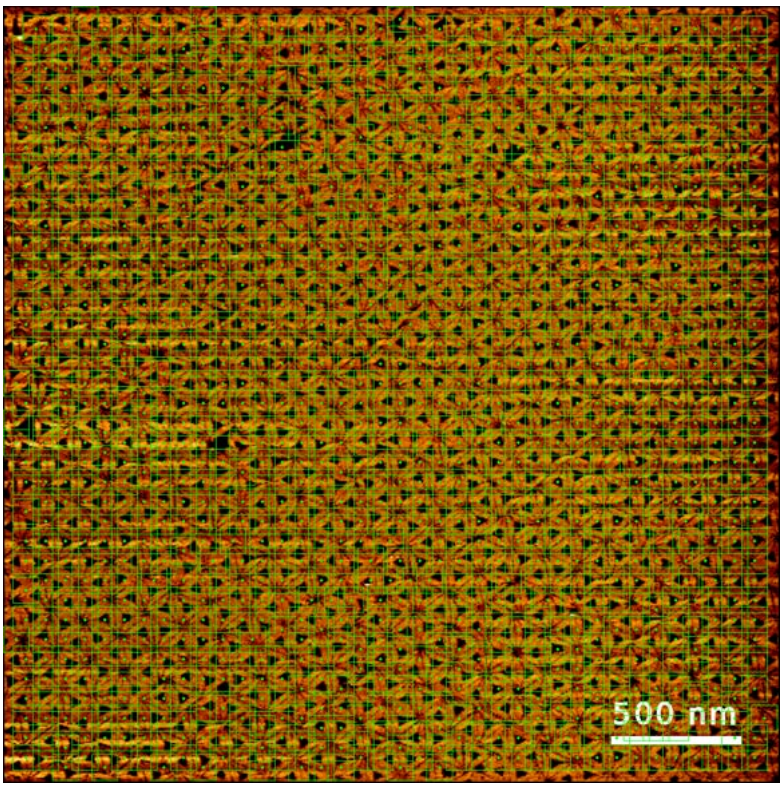


Figura 5.2 *Aplicación del modelo de clasificación*

Esto es debido a las distintas limitaciones de este tipo de modelos. Para empezar, los tamaños de los cuadrados delimitadores son del mismo tamaño (34x34), ya que definir distintos tamaños supondría una gran carga computacional, además de requerir una gran cantidad de tiempo en la ejecución del modelo.

Seguidamente, las esquinas de las imágenes no suelen ser procesadas, debido a que al deslizar una ventana 34x34, cabe la posibilidad de que al recortar la imagen por la zona no llegue a este tamaño requerido y se descarte.

Para visualizar el problema que pueden suponer los tamaños de los cuadrados delimitadores, se aplica el modelo a una imagen que posee unas moléculas de ADN, triángulos, de menor tamaño. Figura 5.3, tal y como se puede comprobar las cajas no son del tamaño adecuado.

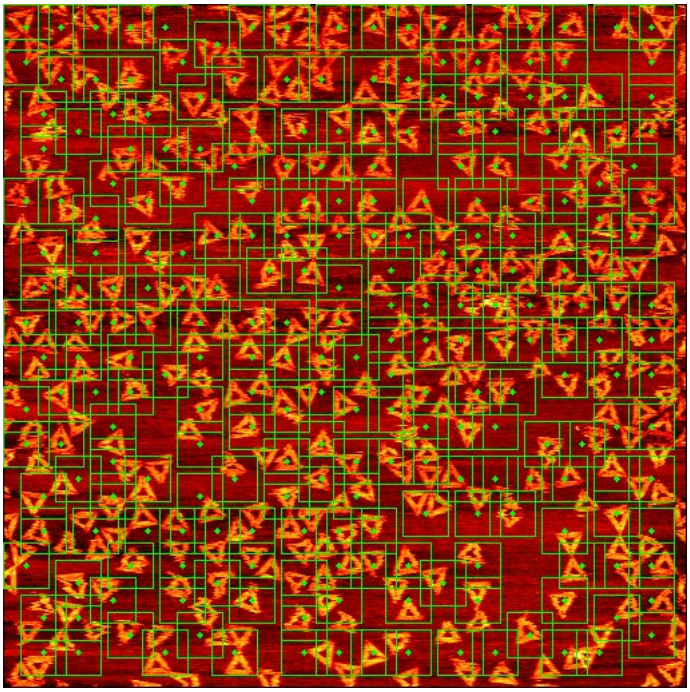


Figura 5.3 *Aplicación del modelo de clasificación a una imagen con triángulos más pequeños*

5.2. RESULTADOS. MODELO REENTRENADO FASTER R-CNN

Una vez analizados los resultados del modelo anterior, se procede al análisis de las soluciones obtenidas por el modelo reentrenado de Faster R-CNN.

Tal y como se explicó en el capítulo anterior, Faster R-CNN está compuesto por un componente que se encarga de definir regiones al recibir una imagen de entrada con posibles objetos en su interior, Region Proposal Network (RPN).

Además, también se compone de un clasificador, que predice dentro de las regiones las probabilidades de que en ellas se encuentre cada una de las clases definidas.

La función de pérdida utilizada combina estos dos procesos. La clasificación y la predicción de cajas delimitadoras. Con el fin de visualizar el proceso de entrenamiento, se obtienen las gráficas en el cual se observa como reduce esta función a medida que se itera el entrenamiento de la red. Primero, el del componente RPN, (figura 5.4). Después, las pérdidas de clasificación, (figura 5.5). Finalmente, el conjunto de ambas. (Figura 5.6).

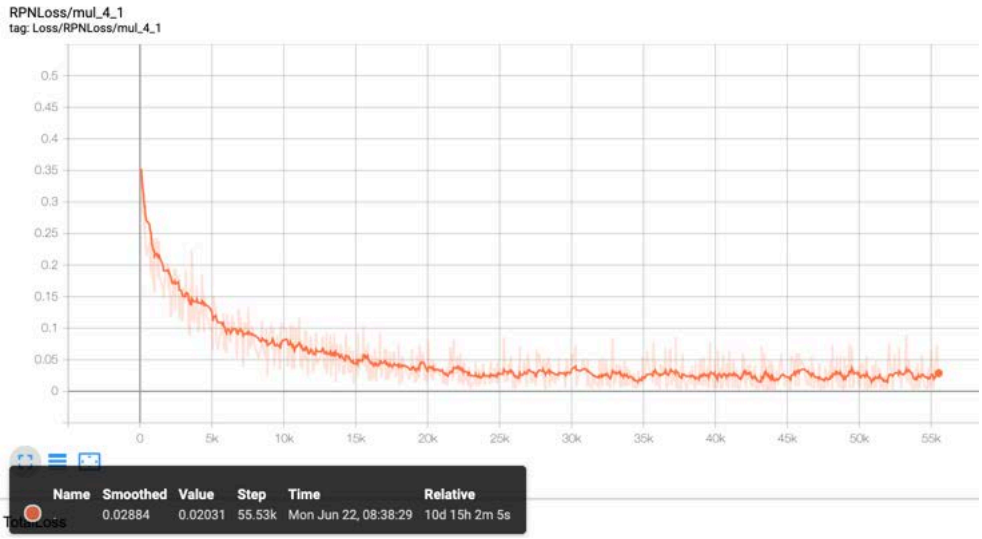


Figura 5.5 *Pérdida del componente RPN durante el proceso de reentrenamiento de Faster R-CNN*

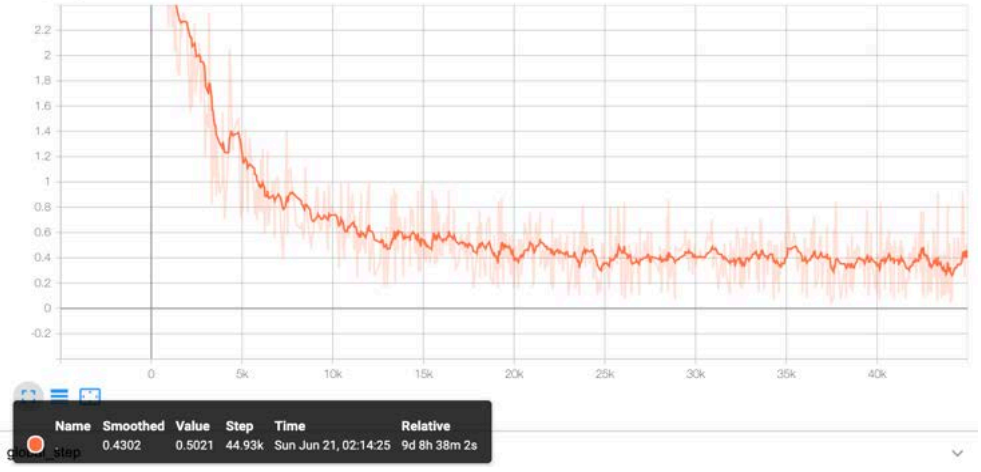


Figura 5.4 *Pérdida del clasificador durante el proceso de reentrenamiento de Faster R-CNN*

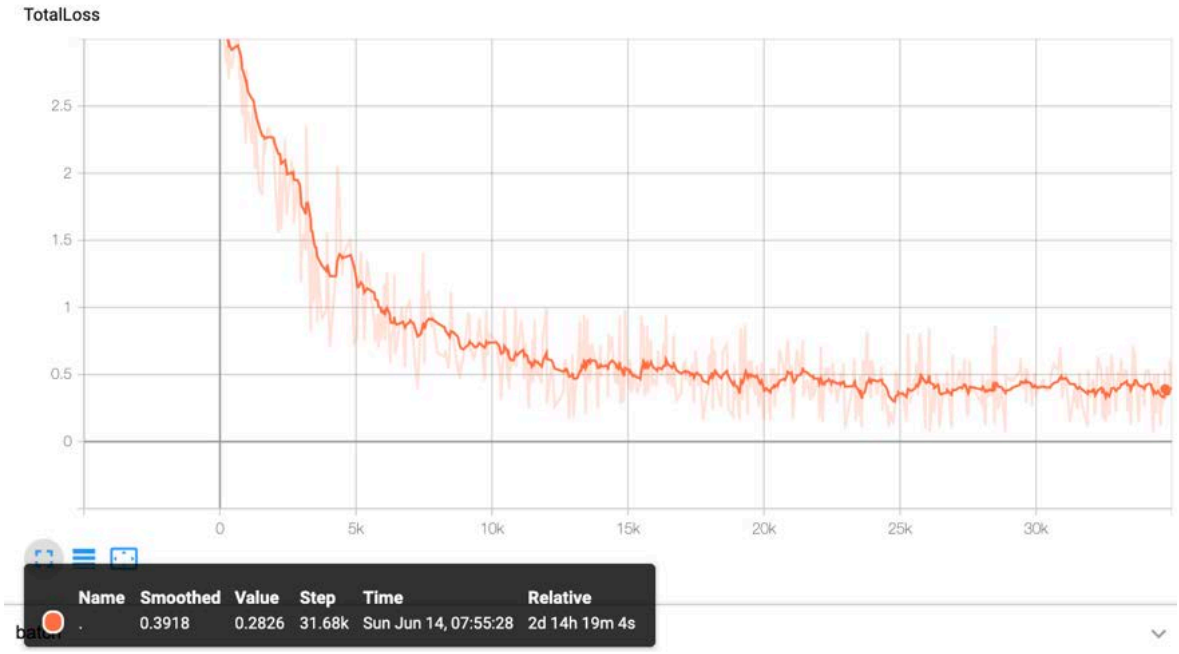


Figura 5.6 *Pérdida total durante el proceso de reentrenamiento de Faster R-CNN*

Tal y como se puede comprobar en las gráficas, en el comienzo del entrenamiento las pérdidas disminuyen. Sin embargo, a medida que avanza llega a un límite en el cual ya no aprende nada nuevo. Es entonces cuando se procede a parar el entrenamiento y a probar el modelo con imágenes nunca vistas por la red con el fin de comprobar que no se haya producido sobreajuste.

Además, se observa a continuación los histogramas de los pesos de la primera etapa, RPN, y la segunda etapa, el clasificador, que, tal y como se puede comprobar en la figura 5.7, se hace uso de datos normalizados.

Tal y como se puede comprobar, (figura 5.7) los valores de los pesos siguen distribuciones simétricas, seguidamente, se analiza como sus valores no son aleatorios, es decir, se han aprendido correctamente los valores de los mismos que encuentran las características propias de los triángulos.

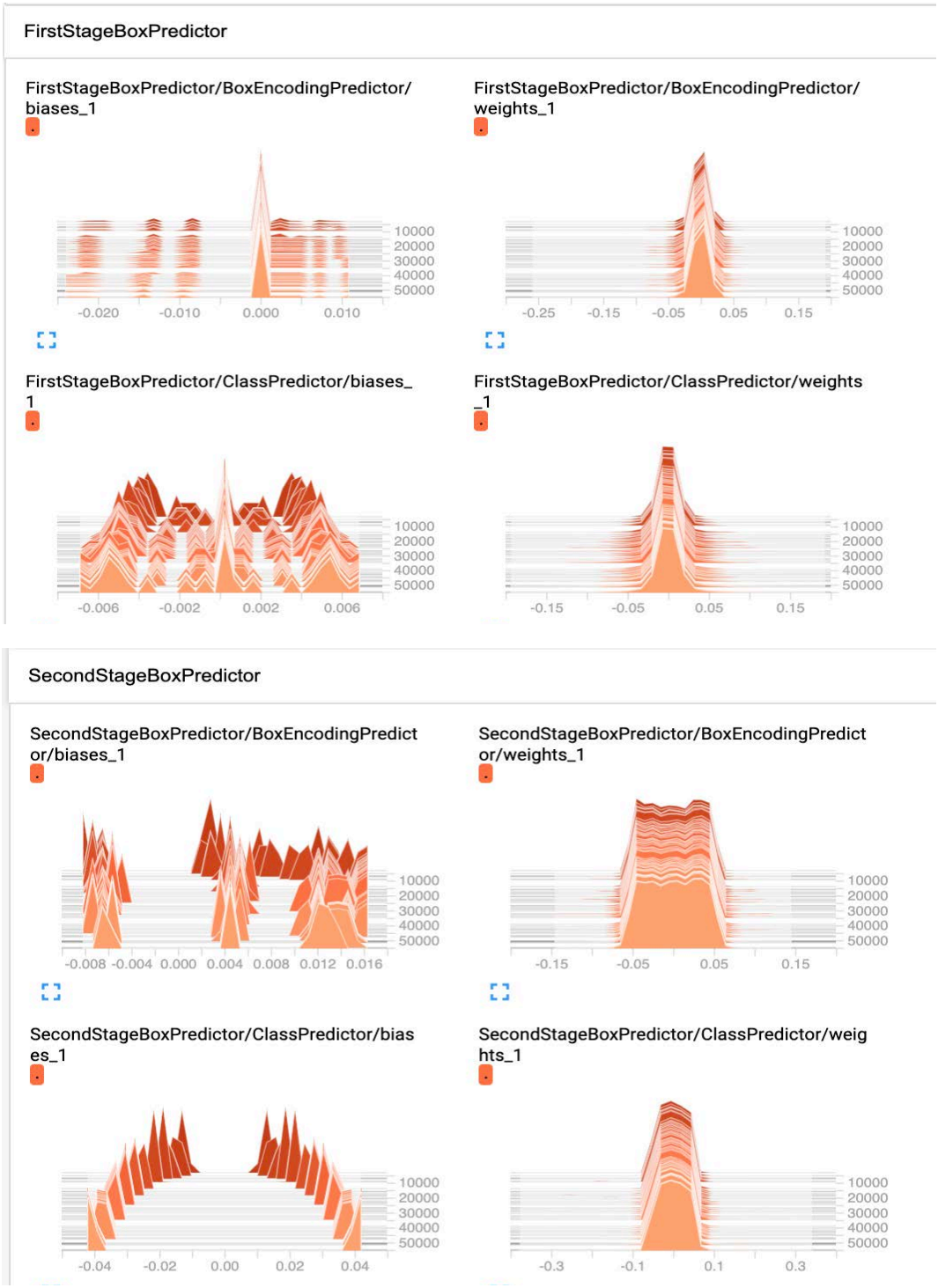


Figura 5.7 *Histograma de los pesos del modelo reentrenado Faster R-CNN*

Una vez visto el interior del modelo, se procede al análisis de resultados, a continuación, en la tabla 5.2, se observa la precisión media (AP) y el recall al aplicarse el modelo al conjunto global de datos. Además, con el fin de comprobar la posibilidad de sobreajuste, se muestran los resultados obtenidos tras aplicar el modelo a dos imágenes nunca vistas por la red.

El código realizado para sacar estos resultados se encuentran en el Anexo II. Resultados.py

Datos	**mAP**	**Precisión Media (AP)**	**Recall**
Global	0.853	0.853	0.890
Imagen1	0.950	0.950	0.786
Imagen2	0.751	0.751	0.936

***Tabla 5.2**. Resultados del reentrenamiento de Faster R-CNN*

En el apartado anterior, se mostraron los resultados al aplicar el modelo de clasificación a una imagen en la que se encuentran unos triángulos de menor tamaño. Para comprobar la eficacia de este modelo en este tipo de imágenes se prueba la red y se observan los resultados en la siguiente figura 5.8

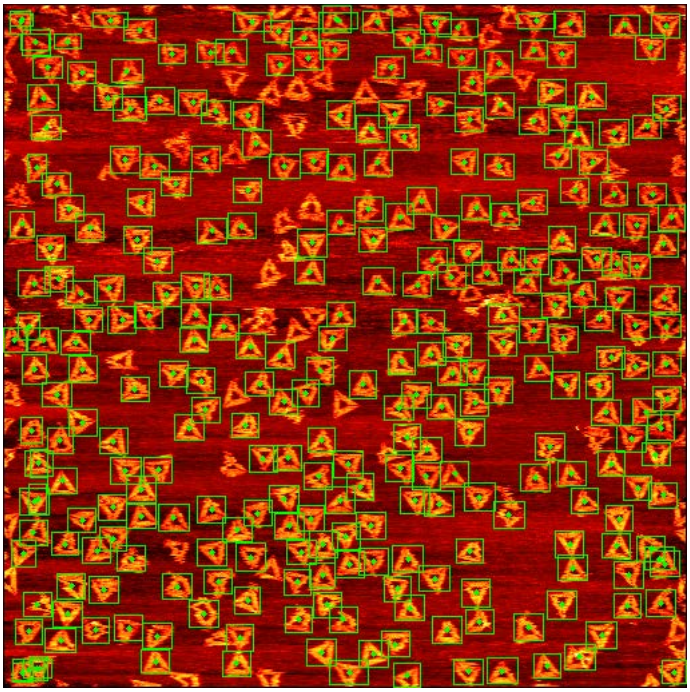


***Figura 5.8**. Aplicación del modelo de Faster R-CNN*

Tal y como se puede comprobar el modelo se ajusta al tamaño de los objetos y detecta los triángulos de forma adecuada. A continuación, en las siguientes figuras, 5.9 y 5.10, se muestran los resultados tras aplicar el modelo a distintas imágenes.

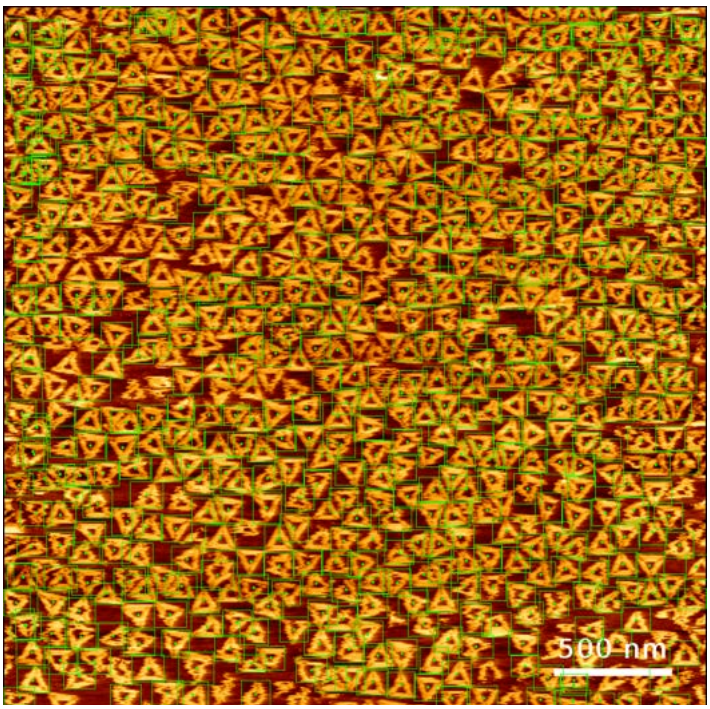


Figura 5.10 *Aplicación del modelo Faster R-CNN*

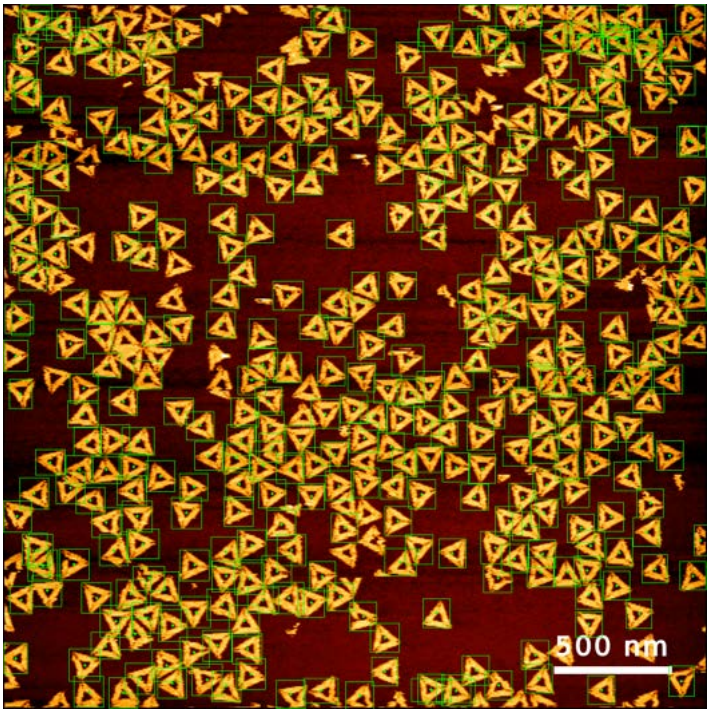


Figura 5.9 *Aplicación del modelo Faster R-CNN*

Tal y como se puede comprobar a pesar de detectar la gran mayoría de los triángulos aún se encuentran triángulos no detectados. Esto explica que la precisión total se mantiene, aproximadamente, en 0.85.

A continuación, se comparan los resultados obtenidos por el modelo de Faster R-CNN original entrenado para el conjunto de datos de PASCAL VOC y MS COCO, dichas tablas se han obtenido del documento oficial de Faster R-CNN [33]. Véanse las figuras 5.11 y 5.12, con los resultados obtenidos en el reentrenamiento, en el cual se han aplicado modificaciones a la configuración del modelo original con el fin de adaptarlo al problema planteado. (Tabla 5.3)

method	# box	data	mAP	areo	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
SS	2000	12	65.7	80.3	74.7	66.9	46.9	37.7	73.9	68.6	87.7	41.7	71.1	51.1	86.0	77.8	79.8	69.8	32.1	65.5	63.8	76.4	61.7
SS	2000	07++12	68.4	82.3	78.4	70.8	52.3	38.7	77.8	71.6	89.3	44.2	73.0	55.0	**87.5**	80.5	80.8	72.0	35.1	68.3	**65.7**	80.4	64.2
RPN	300	12	67.0	82.3	76.4	71.0	48.4	45.2	72.1	72.3	87.3	42.2	73.7	50.0	86.8	78.7	78.4	77.4	34.5	70.1	57.1	77.1	58.9
RPN	300	07++12	70.4	84.9	79.8	74.3	53.9	49.8	77.5	75.9	88.5	45.6	77.1	55.3	86.9	81.7	80.9	79.6	40.1	72.6	60.9	81.2	61.5
RPN	300	COCO+07++12	**75.9**	**87.4**	**83.6**	**76.8**	**62.9**	**59.6**	**81.9**	**82.0**	**91.3**	**54.9**	**82.6**	**59.0**	**89.0**	**85.5**	**84.7**	**84.1**	**52.2**	**78.9**	65.5	**85.4**	**70.2**

Figura 5.11 *Resultados del entrenamiento de Faster R-CNN original aplicado a los datos PASCAL VOC [33]*

method	proposals	training data	COCO val		COCO test-dev	
			mAP@.5	mAP@[.5, .95]	mAP@.5	mAP@[.5, .95]
Fast R-CNN [2]	SS, 2000	COCO train	-	-	35.9	19.7
Fast R-CNN [impl. in this paper]	SS, 2000	COCO train	38.6	18.9	39.3	19.3
Faster R-CNN	RPN, 300	COCO train	41.5	21.2	42.1	21.5
Faster R-CNN	RPN, 300	COCO trainval	-	-	**42.7**	**21.9**

Figura 5.12 *Resultados del entrenamiento de Faster R-CNN original aplicado a los datos MS COCO [33]*

Modelo	**Conjunto de Datos**	**mAP**
Faster R-CNN original	PASCAL VOC	75.9
Faster R-CNN original	MS COCO	42.7
Faster R-CNN reentrenado	Problema actual	85.3

Tabla 5.3 *Comparación de resultados del modelo Faster R-CNN*

Cabe destacar que el modelo desarrollado para la detección de moléculas de ADN transformados en triángulos únicamente se ha entrenado para detectar esta clase. Sin embargo, el entrenamiento del modelo original corresponde a la media de las precisiones de las distintas clases. En la figura 5.11 se puede comprobar las precisiones de cada una, que tal y como se observa, según el tipo de etiqueta a buscar el modelo puede funcionar mejor o peor.

Se comprueba en las siguientes imágenes (5.13-5.27) los resultados. Con el fin de analizar los falsos positivos y negativos, se muestra en rojo, las posiciones reales, y, en verde, las predichas por el modelo. Además, en la siguiente tabla 5.4 se comprueba la precisión media de cada figura junto con el porcentaje de falsos positivos (FP) y falsos negativos (FN).

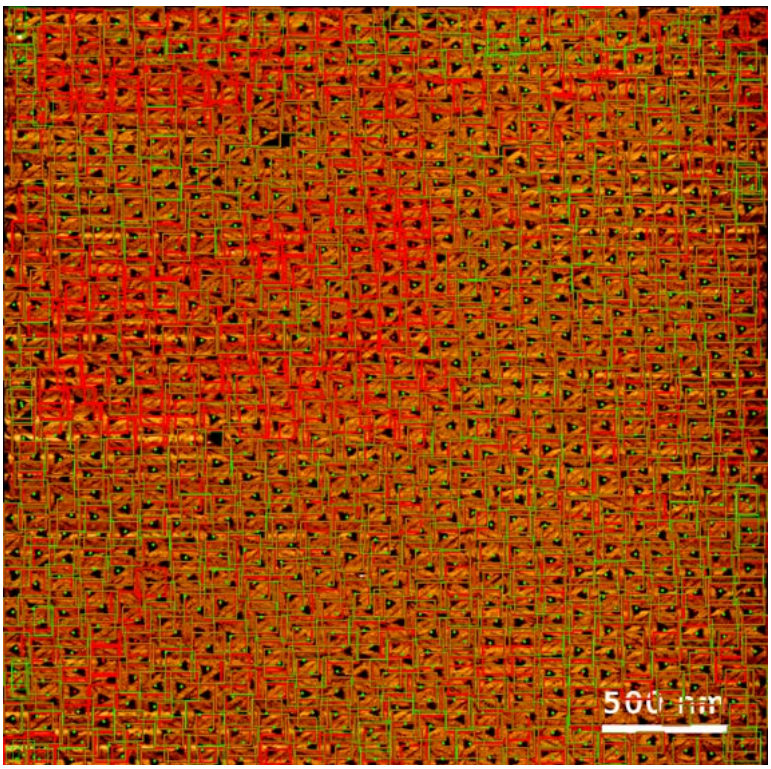


Figura 5.13. *Resultados reales y predichos por el modelo Faster R-CNN*

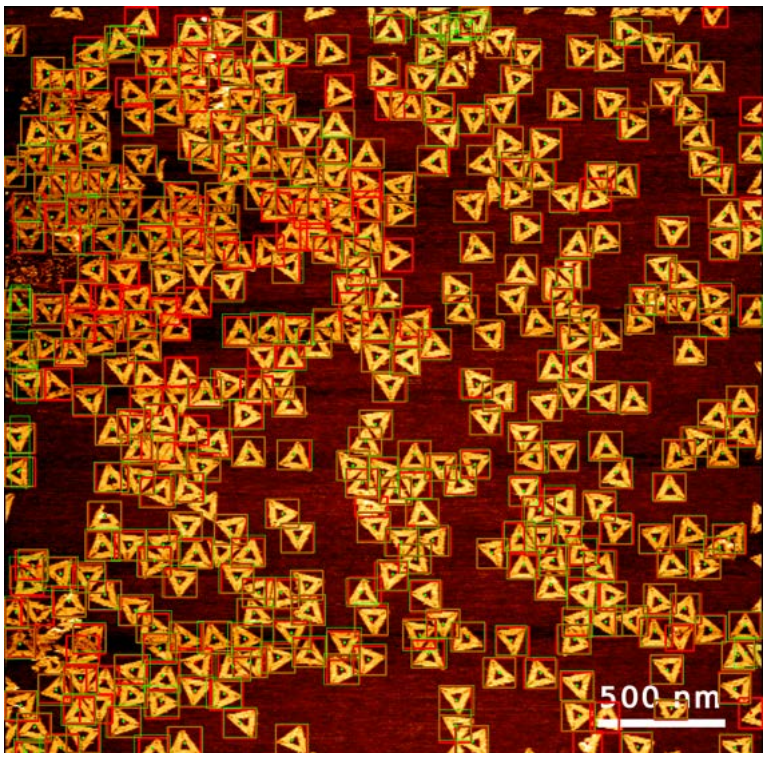


Figura 5.14 *Resultados reales y predichos por el modelo Faster R-CNN*

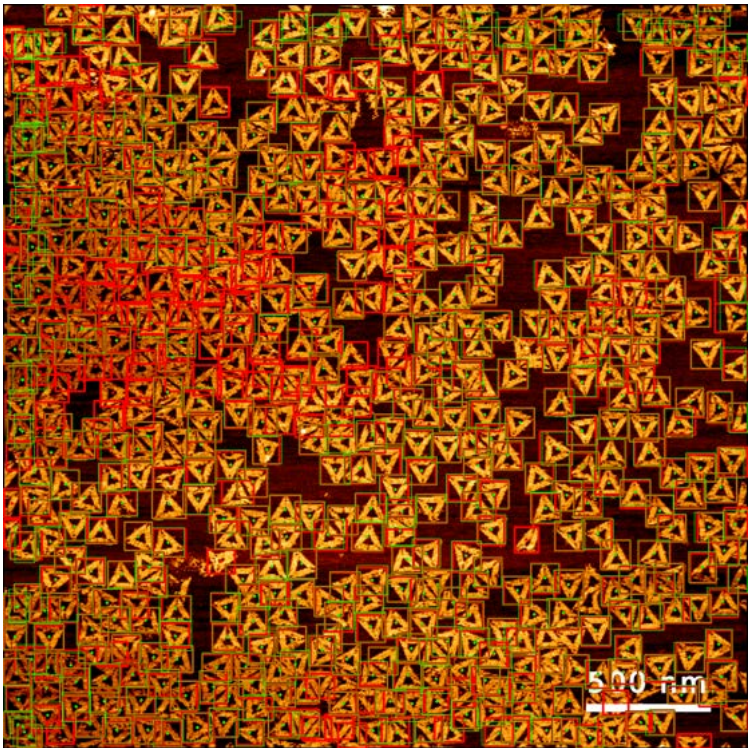


Figura 5.15 *Resultados reales y predichos por el modelo Faster R-CNN*

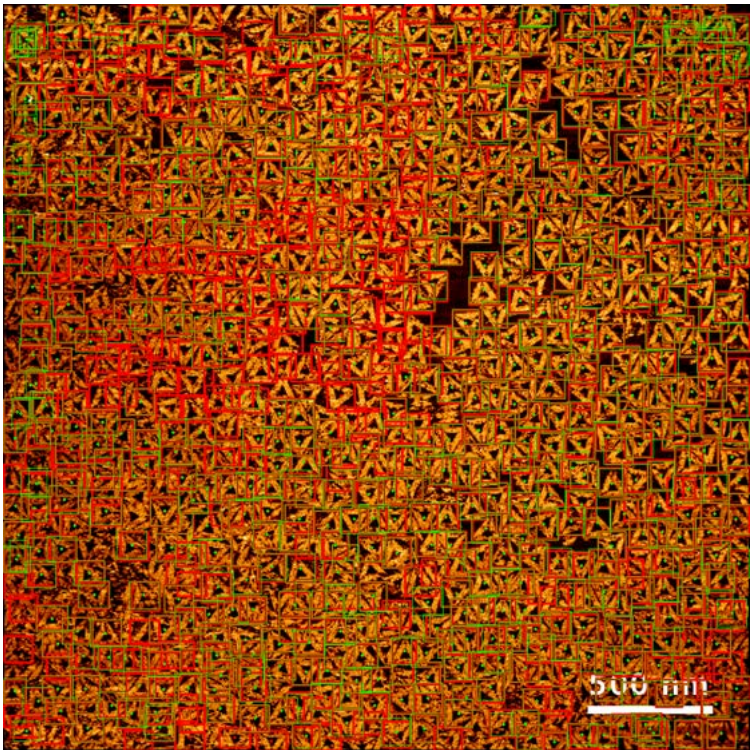


Figura 5.16 *Resultados reales y predichos por el modelo Faster R-CNN*

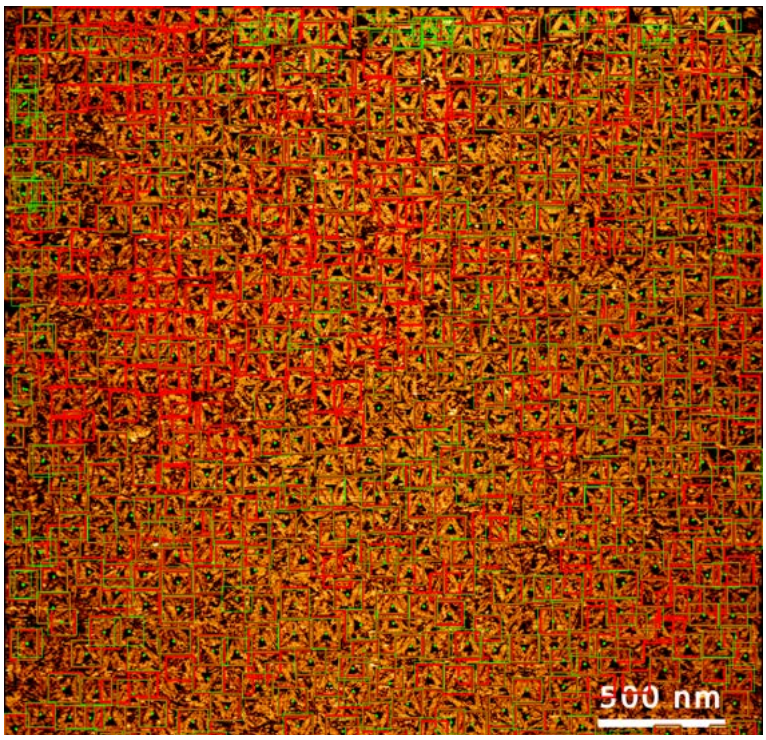


Figura 5.18 *Resultados reales y predichos por el modelo Faster R-CNN*

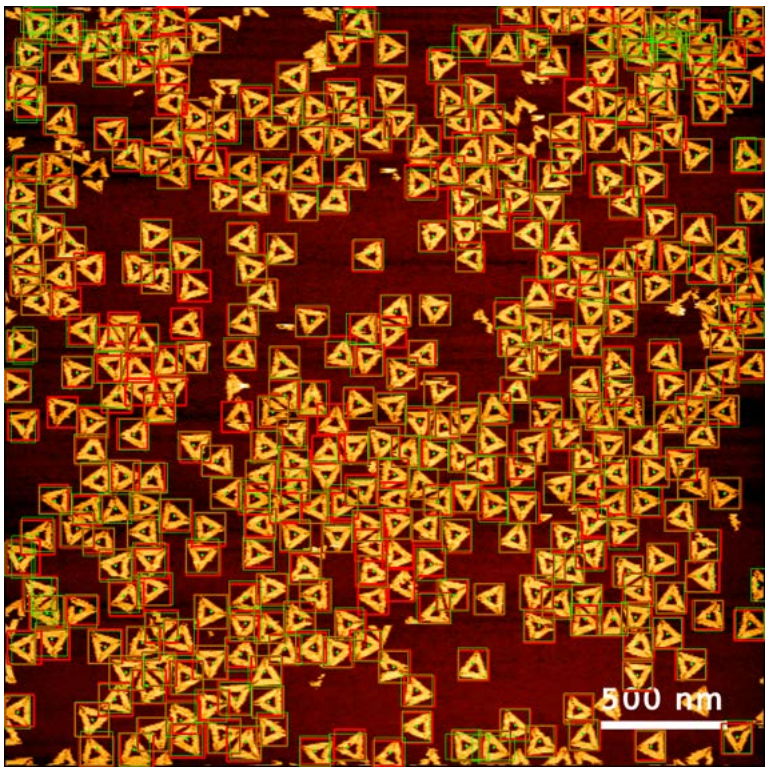


Figura 5.17 *Resultados reales y predichos por el modelo Faster R-CNN*

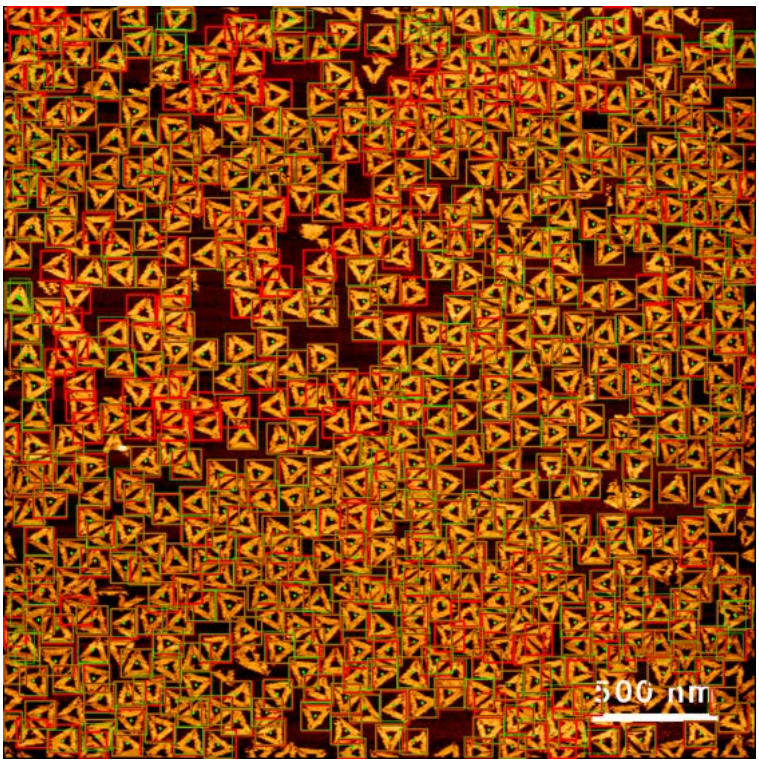


Figura 5.19 *Resultados reales y predichos por el modelo Faster R-CNN*

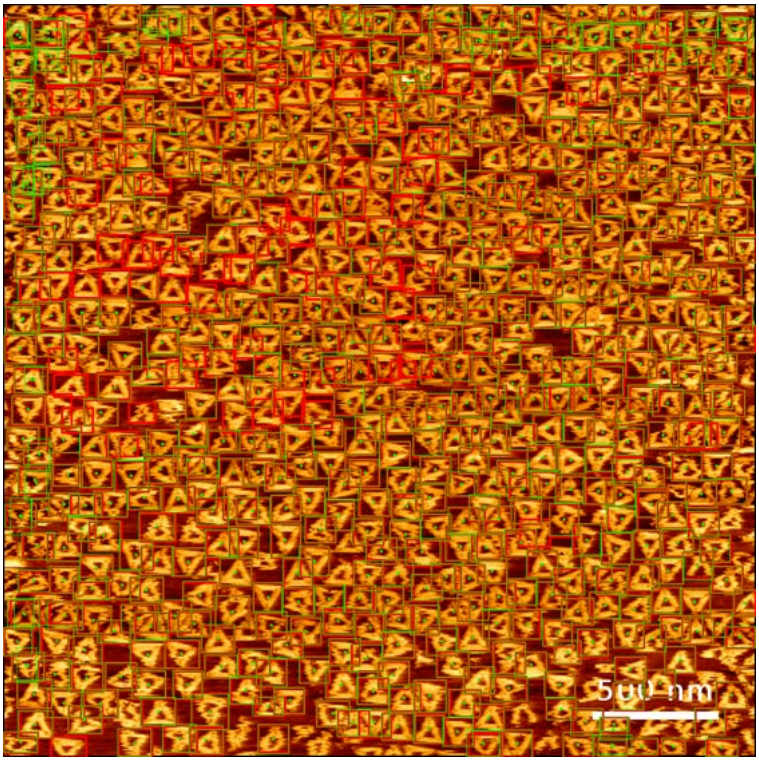


Figura 5.20 *Resultados reales y predichos por el modelo Faster R-CNN*

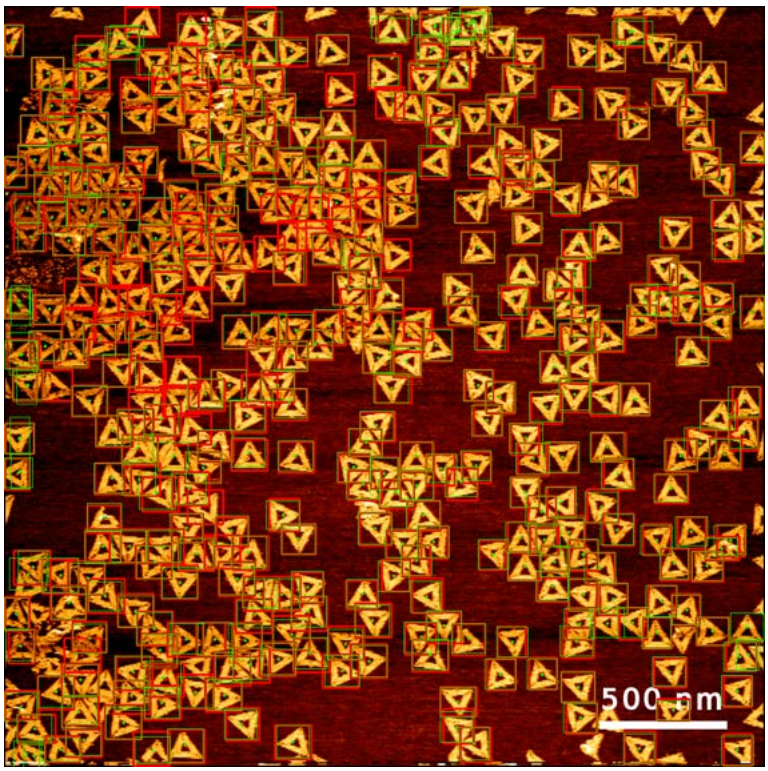


Figura 5.21 *Resultados reales y predichos por el modelo Faster R-CNN*

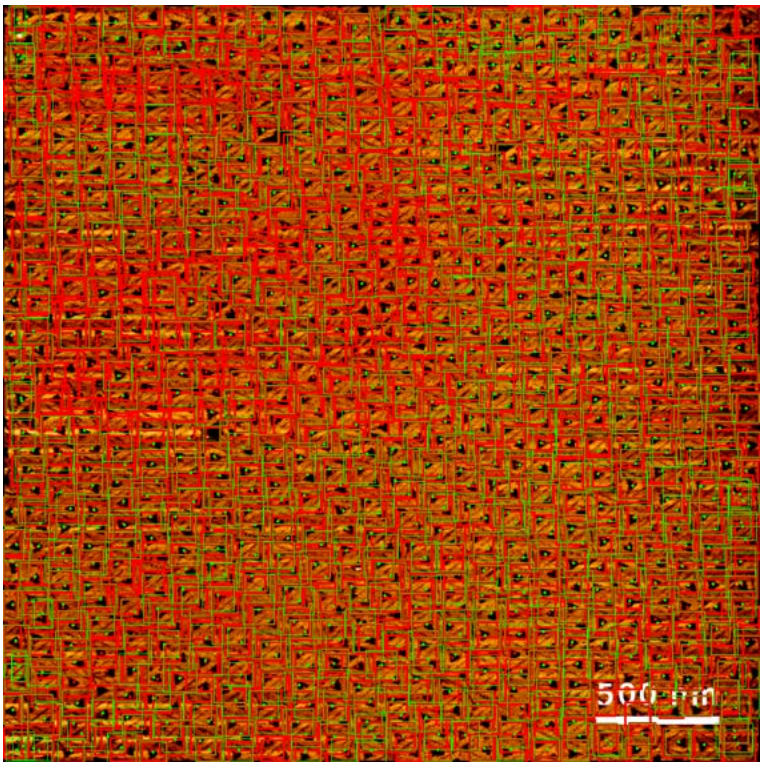


Figura 5.22 *Resultados reales y predichos por el modelo Faster R-CNN*

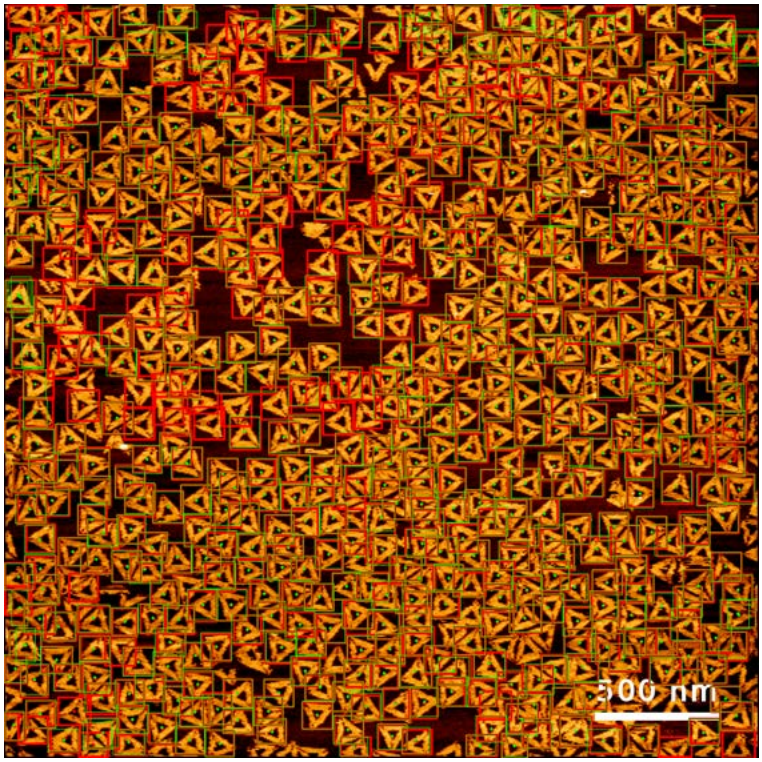


Figura 5.23 *Resultados reales y predichos por el modelo Faster R-CNN*

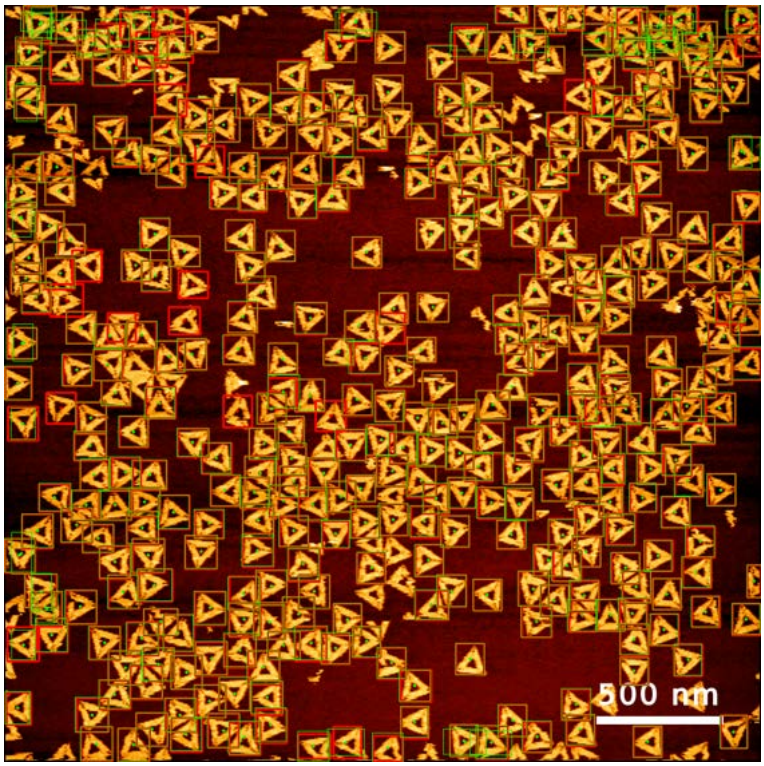


Figura 5.24 *Resultados reales y predichos por el modelo Faster R-CNN*

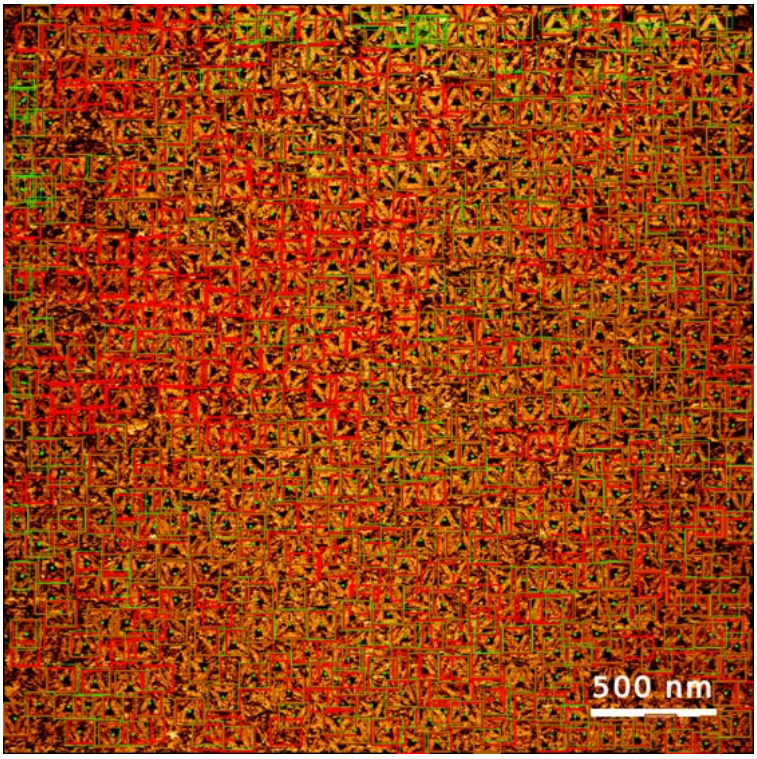


Figura 5.25 *Resultados reales y predichos por el modelo Faster R-CNN*

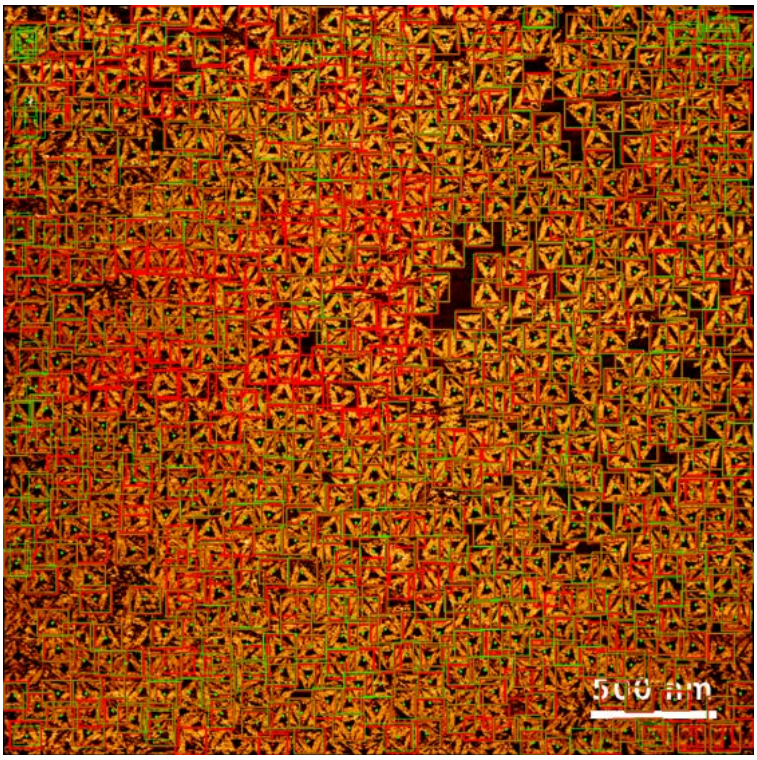


Figura 5.26 *Resultados reales y predichos por el modelo Faster R-CNN*

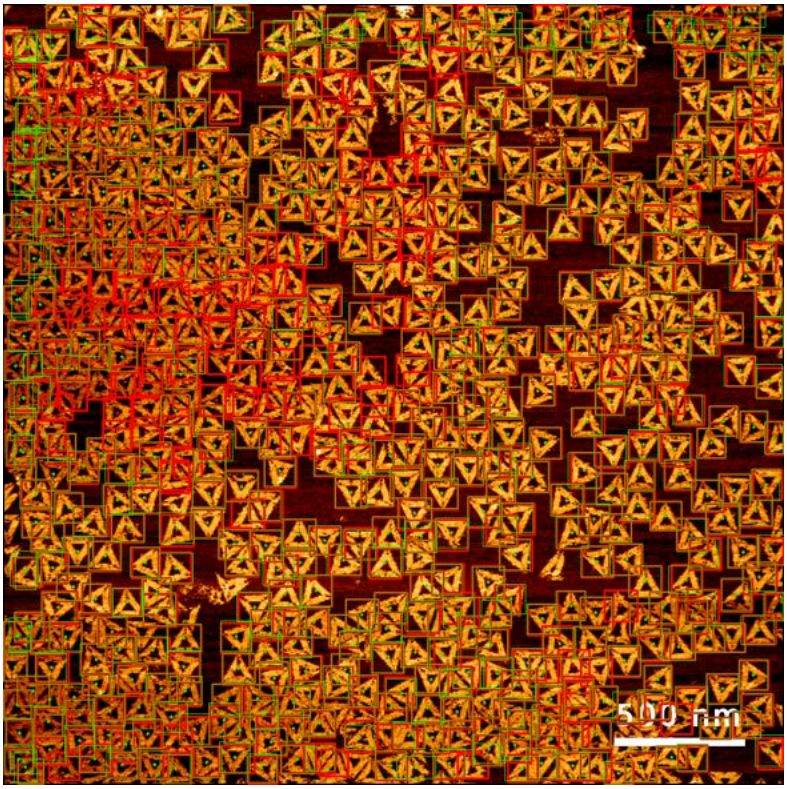


Figura 5.27 *Resultados reales y predichos por el modelo Faster R-CNN*

Figuras	Precisión Media - AP (%)	Porcentaje FN (%)	Porcentaje FP (%)
5.13	79.6	10.1	3.5
5.14	92.1	9.3	2.5
5.15	86.7	9.2	3.1
5.16	84.5	13.7	2.3
5.17	80.4	11.5	3.4
5.18	95	9.2	2.1
5.19	91	9.4	2.6
5.20	89.2	10.3	3.8
5.21	90.9	9.8	2.2
5.22	75.1	13.3	3.9
5.23	91.1	9.2	3.2
5.24	96.1	8.9	2.3
5.25	79.3	14.2	4.1
5.26	84.7	11.4	2.8
5.27	87	10.2	2.4

Tabla 5.4 *Porcentaje de los resultados por imagen*

Tal y como se puede comprobar, las imágenes con mayor AP son aquellas las cuales los triángulos se encuentran más distanciados, esto es debido al funcionamiento del modelo.

Recuérdese que Faster R-CNN propone una cantidad de regiones tras pasar por una red reentrenada. Este valor se asigna en la fase de entrenamiento, para este problema se han configurado un máximo de 1100 propuestas de regiones, de las cuales únicamente puede haber 1000 por clase. De estas, pueden encontrarse distintos tamaños y superposiciones entre ellas, además de posiciones en las que no se encuentre ninguna clase, por ello, aquellas imágenes con un mayor número de moléculas de ADN son las más difíciles para el modelo.

Una de las desventajas de añadir un mayor número de propuestas es el tiempo, tanto de entrenamiento, como de procesamiento por imagen, otro problema es la superposición de distintas regiones rodeando a un mismo objeto.

Se muestra a continuación las propuestas de regiones del modelo Faster R-CNN sin pasar por el clasificador en una de las imágenes del conjunto de datos, figura 5.28.

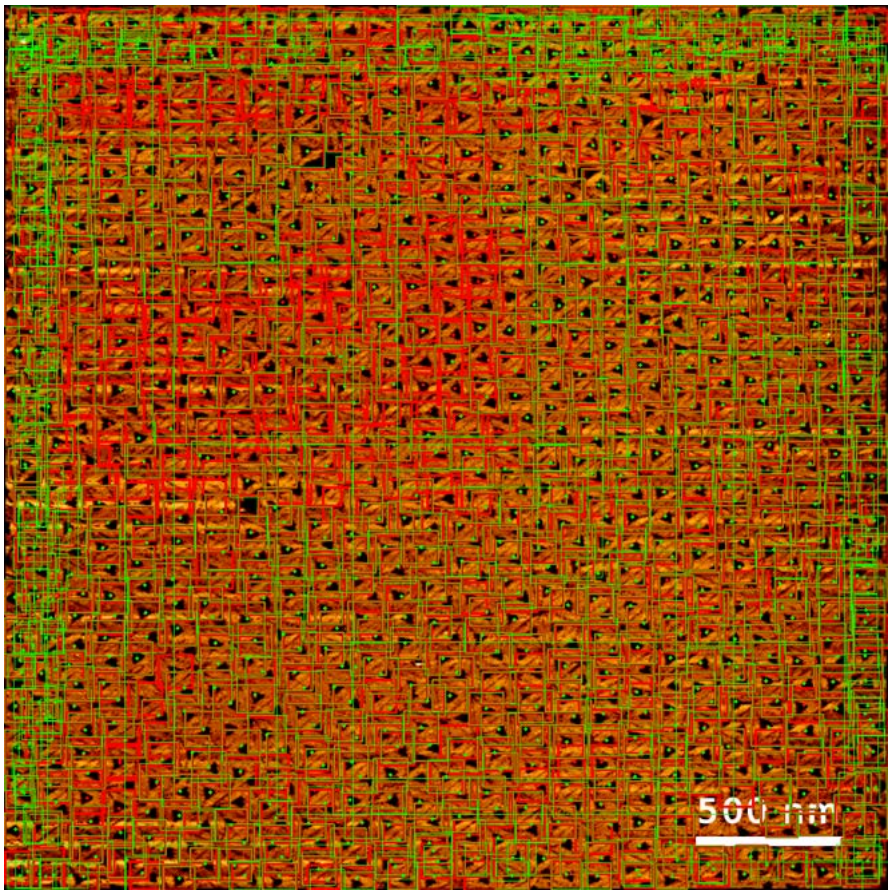


Figura 5.28 *Faster R-CNN como detector de regiones*

Se observa como los falsos negativos, los triángulos no identificados, se debe a la primera fase del modelo, el cual no identifica como objeto los distintos triángulos.

Se analiza a continuación los fallos del modelo, para ello se selecciona una de las imágenes del conjunto de datos y se observan distintas zonas para comprender los errores del modelo. Figura 5.30, en el que se encuentran zonas con falsos positivos, y figura 5.29, en el que se encuentran zonas con falsos negativos.

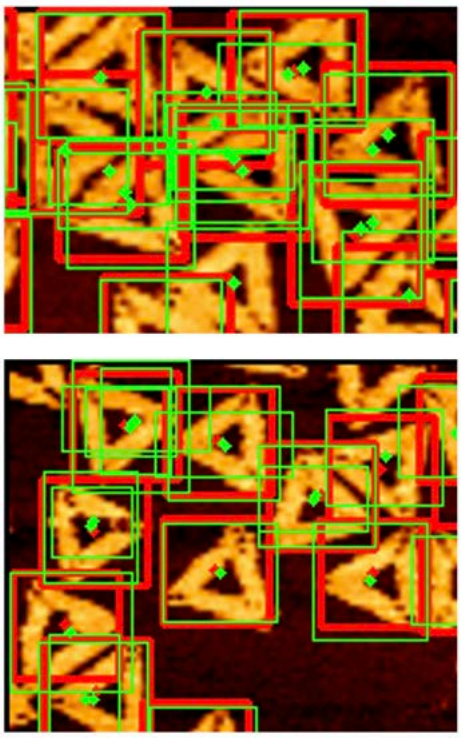


Figura 5.30 *Falsos Positivos*

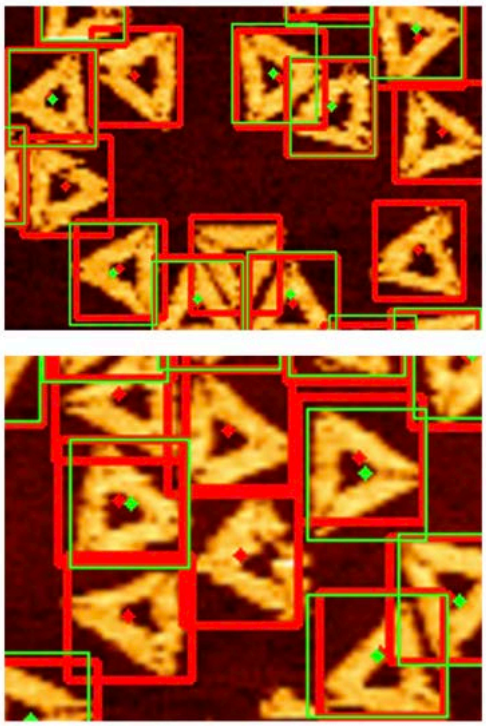


Figura 5.29 *Falsos Negativos*

En conclusión, los falsos positivos, zonas en las que no se encuentra un triángulo y el modelo predice lo contrario, son en su gran mayoría superposiciones de regiones que rodean al mismo triángulo. Y, los falsos negativos se deben a lo explicado anteriormente, no se han propuesto como regiones a clasificar

Además, los falsos negativos también son triángulos con ruido, es decir, que no queda clara la forma, tal y como se comprueba en la figura 5.29.

5.3 RESULTADOS DEL CONJUNTO DE AMBOS MODELOS

Otra manera de dar solución al problema es fusionar ambos modelos, tanto la red reentrenada como el clasificador. Para ello se utiliza la red de Faster R-CNN únicamente como generador de propuestas de regiones, y, una vez entrenada, se aplica el clasificador en las coordenadas de la imagen sacadas por el modelo de clasificación creado, en vez de utilizar las predicciones que salen del modelo original.

En conclusión, se utiliza el RPN del modelo sin hacer uso del clasificador, tal y como se muestra en la siguiente figura 5.13.

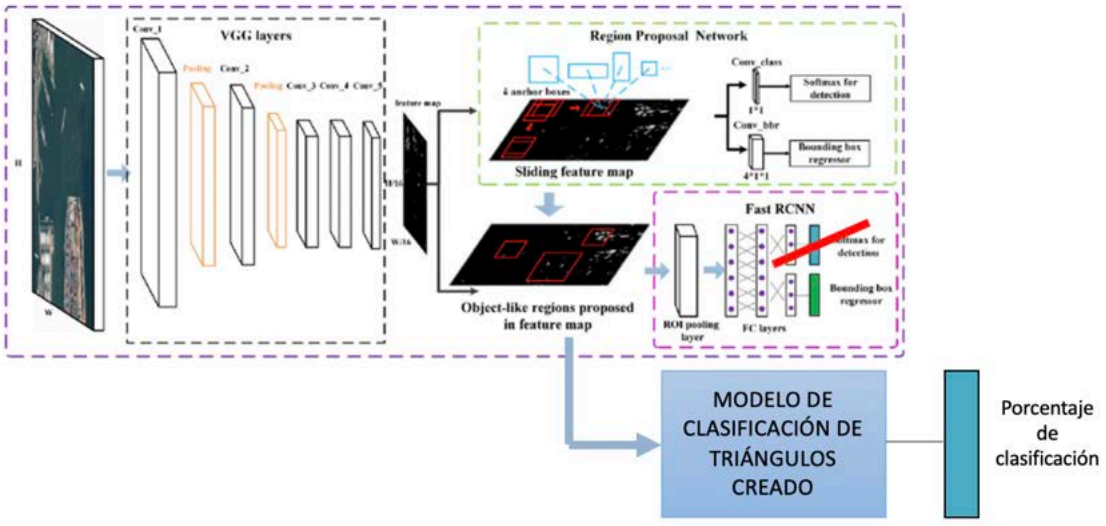


Figura 5.31 *Arquitectura de Faster R-CNN junto con un clasificador de triángulos*

Los resultados obtenidos tras aplicar el modelo al conjunto global de los datos, junto con dos imagenes nunca vistas por el modelo, se muestra a continuación en la siguiente tabla 5.4.

Datos	**mAP**	**Precisión Media (AP)**	**Recall**
Global	0.703	0.703	0.744
Imagen 1	0.842	0.842	0.874
Imagen 2	0.766	0.766	0.877

Tabla 5.5 *Resultados del conjunto de ambos modelos*

Se comprueba un ejemplo de este modelo aplicado a una imagen del conjunto de datos en la siguiente figura 5.14.

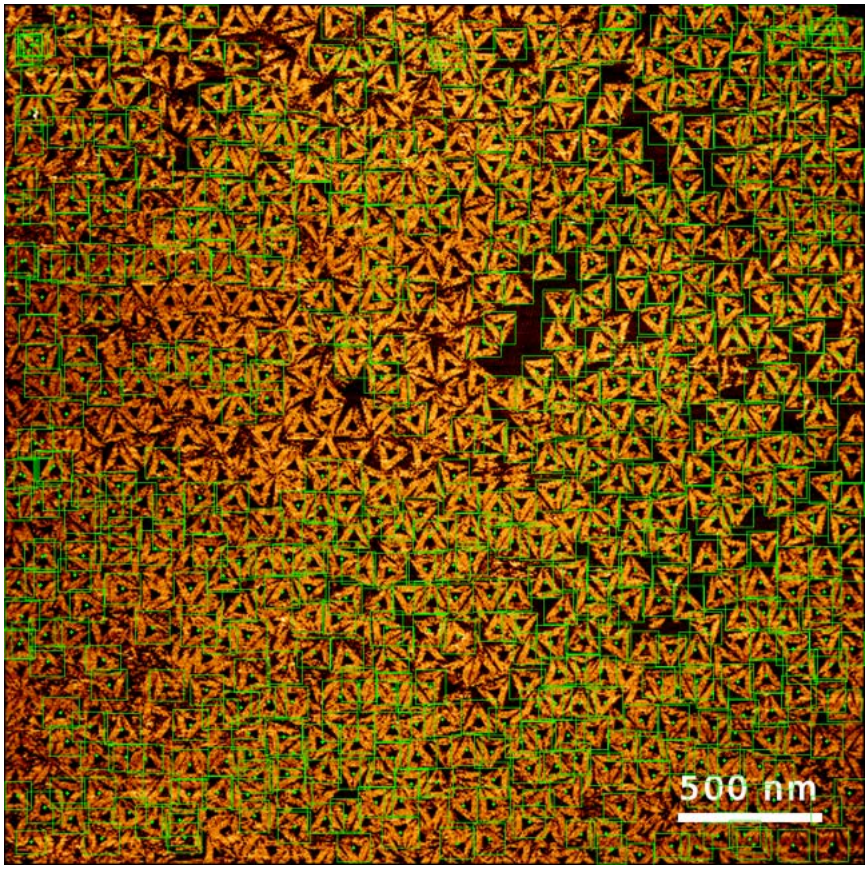


Figura 5.32 *Aplicación de ambos modelos*

Una vez visto este tipo de solución, puede aplicarse, en la arquitectura definida en la figura 5.13, distintos modelos de clasificación, como los ya creados en capítulos anteriores.

Además del modelo de clasificación utilizado, en el apartado 4.2.3 se mostraron una serie de redes diferentes creadas, en las cuales se obtenían altos porcentajes de precisión.

En la siguiente tabla 5.5 se observan los resultados obtenidos de estos modelos como clasificadores, y, además, utilizados como detectores de objetos haciendo uso de la arquitectura definida en la figura 5.13.

Modelo	**Precisión como clasificador**	**Precisión Media (AP)**
Modelo 3	0.897	0.682
Modelo 4	0.999	0.795
Modelo 5	0.997	0.799

Tabla 5.6 *Resultados para el modelo de Faster R-CNN junto con el modelo 4*

Se comprueban los resultados de distintas imágenes aplicadas a estos modelos en las siguientes figuras 5.15, modelo 3, 5.16, modelo 4 y 5.17, modelo 5.

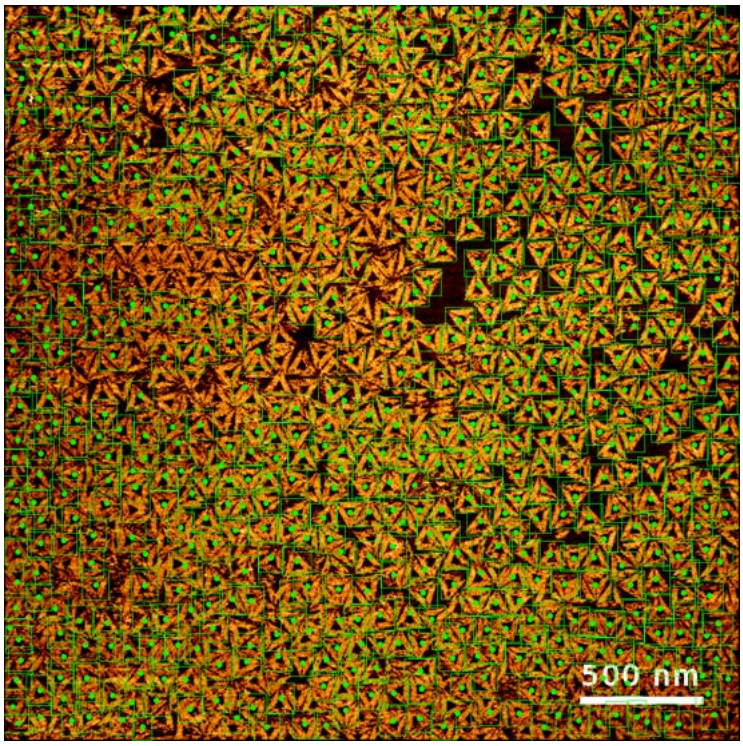


Figura 5.34 *Aplicación del modelo 3 junto con Faster R-CNN*

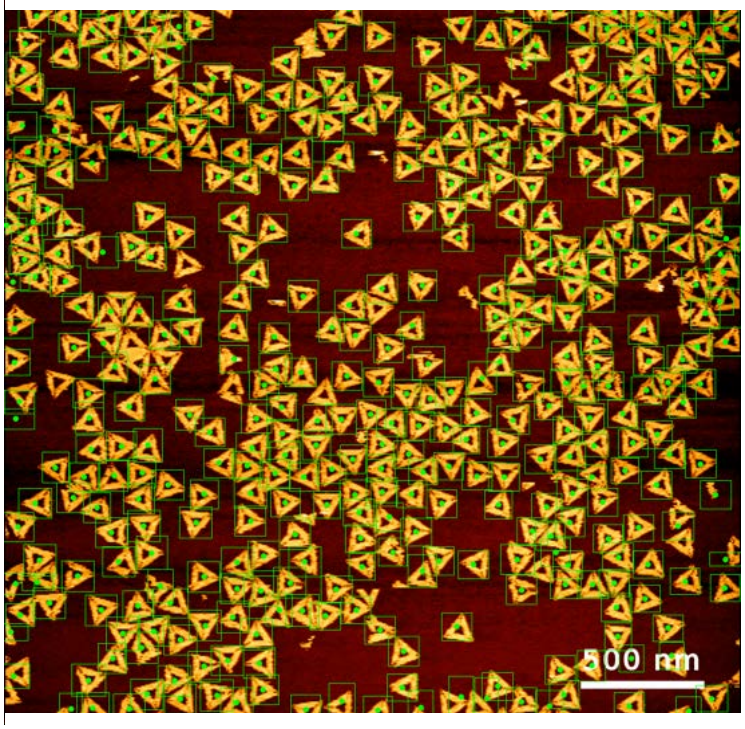


Figura 5.33 *Aplicación del modelo 4 junto con Faster R-CNN*

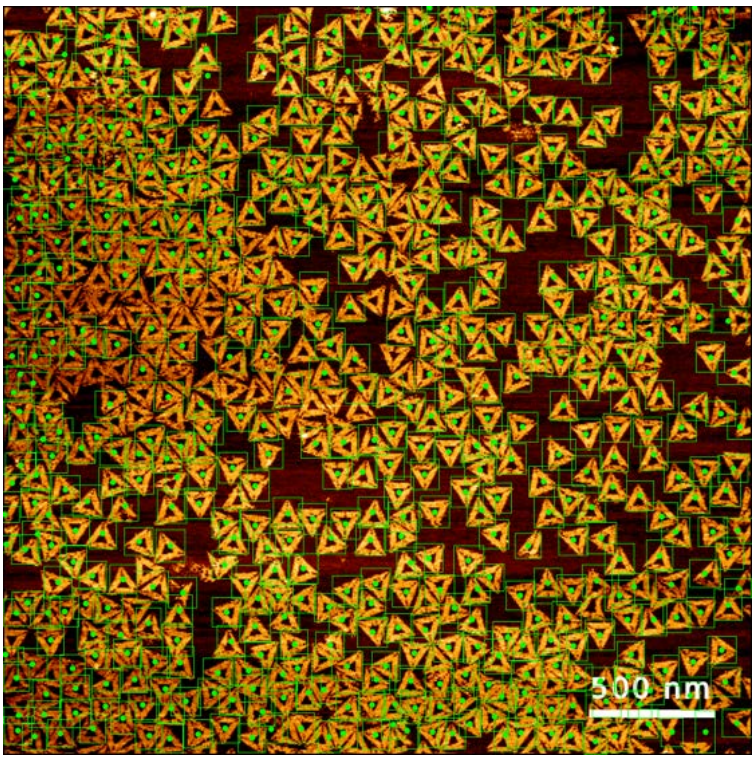


Figura 5.35 *Aplicación del modelo 5 junto con Faster R-CNN*

Tal y como se ha podido comprobar, a pesar de que el modelo escogido en un primer lugar obtuviese mejores resultados como clasificador, como detector de objetos el se obtienen con el modelo 5.

Además, cabe recordar que, en el capítulo 3, se reentrenó la red de clasificación vgg16 modificado para la detección de triángulos. Aplicado a la arquitectura definida en la figura 5.13, se obtienen los siguientes resultados, detallados en la tabla 5.6

Datos	**mAP**	**Precisión Media (AP)**	**Recall**
Global	0.679	0.679	0.873
Imagen 1	0.557	0.557	0.752
Imagen 2	0.771	0.771	0.852

Tabla 5.7 *Resultados del modelo de Faster R-CNN junto con VGG-16*

Un ejemplo de la aplicación del modelo a una imagen se muestra en la siguiente figura 5.18

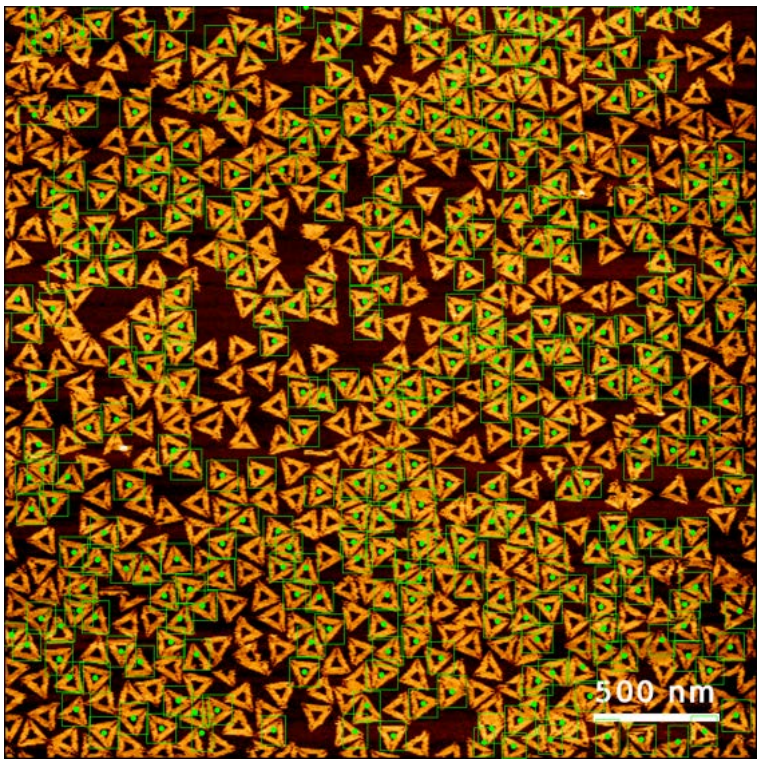


Figura 5.36 *Aplicación del modelo de Faster R-CNN junto con VGG-16*

Capítulo 6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 CONCLUSIONES

Tras analizar los resultados, se puede comprobar que utilizar un modelo de clasificación como detector de objetos no es una forma adecuada de dar solución a este problema, a pesar de hacer uso de distintos algoritmos que ayudan a mejorar los resultados.

Sin embargo, una manera más adecuada de dar solución al proyecto es el reentrenamiento de modelos ya creados para este tipo de problemas. Su uso ya ha sido implementado y, modificando ciertos parámetros del modelo, pueden dan resultados óptimos para cualquier tipo de detección de objetos, tal y como se ha podido comprobar durante el desarrollo del proyecto.

A continuación, en la tabla 6.1 se compararán los siete métodos utilizados y analizados.

Modelo	**mAP (%)**	**Recall (%)**
Clasificación + Algoritmos	10.5	25
Reentrenamiento Faster R-CNN	85.3	89
Faster R-CNN + Clasificación	70.3	74.4
Faster R-CNN + Modelo 3	68.2	88.6
Faster R-CNN + Modelo 4	79.5	85.3
Faster R-CNN + Modelo 5	79.9	86.1
Faster R-CNN + VGG16	67.9	87.3

Tabla 6.1 *Resultados de los modelos desarrollados*

En la siguiente tabla se comprueba como el modelo que obtiene los mejores resultados es el reentrenamiento de Faster R-CNN. Sin embargo, como modelo de detección de objetos, el conjunto de Faster R-CNN, utilizado como propuesta de regiones, junto con modelos de clasificación puede considerarse eficiente.

A continuación, en la tabla 6.2 se muestran los resultados obtenidos por los modelos de detección de objetos más conocidos en comparación a los obtenidos en el desarrollo del proyecto. Dichos datos se han obtenido de la documentación oficial de cada modelo, YOLO [32], Single Shot MultiBox Detector (SSD) [34], Fast R-CNN[36] y Faster R-CNN [33].

Modelo	**mAP (%)**
Clasificación + Algoritmos	10.5
Reentrenamiento Faster R-CNN	85.3
Faster R-CNN + Clasificación	70.3
Faster R-CNN + Modelo 3	68.2
Faster R-CNN + Modelo 4	79.5
Faster R-CNN + Modelo 5	79.9
Faster R-CNN + VGG16	67.9
Fast R-CNN	70
Faster R-CNN original	75.9
SSD	76.8
YOLO	66.4

***Tabla 6.2**. Comparación de distintos modelos de detección de objetos*

Tal y como se puede comprobar, el reentrenamiento de Faster R-CNN de detección de triángulos es el modelo con mejores resultados. Esto es debido a que el resto de los modelos se entrenan para un conjunto de clases y no únicamente para una clase. El tamaño del conjunto de datos necesarios para obtener mejores resultados es mayor.

A continuación, se comprueba la precisión media (AP) para cada clase entrenada del modelo de Faster R-CNN, el cual se ha escogido para comparar sus resultados con respecto a los obtenidos, debido a que es el mismo modelo utilizado. En la comparación se añade la clase triángulo la cual hace referencia al modelo reentrenado durante el desarrollo del proyecto. Tabla 6.3. Se han subrayado aquellas clases que superen el AP de la clase triángulo

CLASE	**AP (%)**
Avión	87.4
Bicicleta	83.6
Pájaro	76.8
Barco	62.9
Botella	59.6
Bus	81.9
Coche	82.9
Gato	91.3
Silla	54.9
Vaca	82.6
Mesa	59.0
Perro	89.0
Caballo	85.5
Moto bicicleta	84.7
Persona	84.1
Planta	52.2

Oveja	78.9
Sofa	65.5
Tren	85.4
Televisión	70.2
Triángulo	85.3

***Tabla 6.3**. Comparación de las clases identificadas por el modelo Faster R-CNN*

Tal y como se ha podido comprobar el modelo desarrollado es una de las clases con los mejores resultados obtenidos por el modelo.

A pesar de que los triángulos sea una clase sencilla y existen otro tipo de clases que a priori pueden parecer más complicados, por ejemplo, la clase gato, obtienen mejores resultados. Esto es debido a que los resultados obtenidos para detectar dichas clases hacen uso de un conjunto mayor de imágenes.

Además, para este tipo de clases (gatos, perros, aviones, etc) se encuentran como mucho dos o tres por imagen. En el caso particular de las moléculas de ADN, se haya en una misma imagen hasta 900 triángulos y de un tamaño reducido, lo que supone para el modelo mayor tiempo de procesamiento y mayor dificultad a la hora de detectarlos.

6.2 TRABAJOS FUTUROS

Tal y como se comentó al inicio, este proyecto surge del ADN origami. Técnica utilizada para plegar las moléculas de ADN con el fin de convertirlas en distintas formas. Una manera de profundizar en el proyecto desarrollado es la de detectar distintos tipos de figuras, no únicamente los triángulos. Además, el modelo reentrenado, Faster R-CNN está creado para la detección de distintos tipos de clases, por lo que podría adaptarse a la detección de moléculas de ADN que presentan distintas formas.

Además, una vez obtenidos las posiciones de las figuras, pueden calcularse las distancias entre cada uno de los triángulos para comprobar el funcionamiento de la red según las distintas características entre las distintas imágenes. Ya que en el conjunto de datos utilizado se encuentran tanto imágenes con muchos triángulos y muy juntos como imágenes con menor número de figuras y más separados entre sí

Con el fin de mejorar los resultados, el buscar nuevas imágenes y entrenar de nuevo la red con más datos siempre es una manera de aumentar la precisión los modelos desarrollados.

Otra manera de avanzar con el proyecto sería implementar distintos detectores de objetos, tales como YOLO o Single Shot Detector (SSD) ya mencionados en anteriores capítulos.

Así se compararán los resultados de estos tres modelos y se podrá escoger la red reentrenada que haya obtenido los mejores resultados. Además, se podrá añadir la aplicación del uso de

la librería de OpenCV y, utilizar la ventaja de la similitud de color de los triángulos en las distintas imágenes para detectar los triángulos, también puede ser otro método de realización del proyecto.

Por último, otra manera de intentar mejorar los resultados es la creación de cero de un modelo de detector de objetos, basándose en las características y en las ventajas de cada uno de los modelos ya conocidos, YOLO, Faster-RCNN y SSD.

Capítulo 7. BIBLIOGRAFÍA

[1] François Chollet. Libro Deep Learning with Python. Capitulo 1 [Available: https://livebook.manning.com/book/deep-learning-with-python/chapter-1/] [Último acceso: Enero 2020]
[2] François Chollet. Libro Deep Learning with Python. Capitulo 1. Sección 1.1.2 [Available: https://livebook.manning.com/book/deep-learning-with-python/chapter-1/18] [Último acceso Marzo 2020]
[3] Geoffrey Hinton [Available: https://elpais.com/elpais/2017/06/16/opinion/1497602599_425102.html] [Último acceso: Enero 2020] [Último acceso Marzo 2020]
[4] Imagen - [Fuente: https://www.atriainnovation.com/que-son-las-redes-neuronales-y-sus-funciones/]
[5] Imagen - Fuente: Libro: Deep Learning for Vision Systems. Capitulo 2. Apartado 2.1.1
[6] Mohamed Elgendy. Libro: Deep Learning for Vision Systems. Capitulo 2. Apartado 2.1.1 [Último acceso: Marzo 2020]
[7] Imagen - Fuente: https://es.slideshare.net/victorbalanta/red-26655361
[8] Imagen - Fuente: Libro: Deep Learning for Vision Systems. Capitulo 2. Apartado 2.1.1
[9] Imagen - Fuente: https://livebook.manning.com/book/deep-learning-with-python/chapter-2/114
[10] Imagen - Fuente: https://www.ellaberintodefalken.com/2019/10/vision-artificial-redes-convolucionales-CNN.html
[11] Imagen - Fuente: https://www.machinecurve.com/index.php/2020/01/30/what-are-max-pooling-average-pooling-global-max-pooling-and-global-average-pooling/
[12] Imagen - Fuente: https://medium.com/@bootcampai/redes-neuronales-convolucionales-5e0ce960caf8
[13] François Chollet. Libro Deep Learning with Python. Capitulo 1. Apartado 1.1.5 [Available: https://livebook.manning.com/book/deep-learning-with-python/chapter-1/section-1-1-5?origin=product-toc] [Último acceso Abril 2020]
[14] Imagen - Fuente: https://www.i2tutorials.com/how-can-you-avoid-local-minima-to-achieve-the-minimized-loss-function/
[15] Imagen - Fuente: https://livebook.manning.com/book/deep-learning-with-python/chapter-1/55
[16] François Chollet. Libro Deep Learning with Python. Capitulo 4. Apartado 4.4 [Available: http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf] [Último acceso Abril 2020]
[17] Imagen - Fuente: https://towardsdatascience.com/improving-accuracy-on-mnist-using-data-augmentation-b5c38eb5a903

[18] Imagen - Fuente: https://medium.com/@vanessabogado/competencia-en-kaggle-google-ai-open-images-detección-de-objetos-4ffaa389cb1c
[19] Imagen - Fuente: https://observatorio-ia.com/facebook-libera-detectron
[20] Código- LabelImg- https://github.com/tzutalin/labelImg
[21] Imagen - Fuente: Libro: Deep Learning with Python : [Available: http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf] [Último acceso Mayo 2020]
[22] Imagen - Fuente: Libro: Deep Learning with Python : [Available: http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf] [Último acceso Mayo 2020]
[23] Imagen - Fuente: Libro: Hands on Machine Learning with Scikit-Learn and TensorFlow: [Available: http://index-of.es/Varios-2/Hands%20on%20Machine%20Learning%20with%20Scikit%20Learn%20and%20Tensorflow.pdf] [Último acceso Mayo 2020]
[24] Karen Simonyan y Andrew Zisserman, Visual Geometry Group, Department of Engineering Science, University of Oxford [Available: https://arxiv.org/pdf/1409.1556.pdf] [Último acceso Mayo 2020]
[25]J.R.R.R. Uijlings, K.E.A. van de Sande, T. Gevers and A.W.M. Smeulders Oficial Paper-Selective Search for Object Recognition. [Available: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf] [Último Acceso Mayo 2020]
[26] Página Oficial Scikit Learn [Available: https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html] [Último acceso Mayo 2020]
[27] Mohamed Elgendy - Libro Deep Learning for Vision Systems Capítulo 7. Apartado 1.4. [Último acceso Mayo 2020]
[28] Imagen - Fuente: Libro: Computer Vision with Python 3. [http://diggerdnepr.ddns.net/wp-content/uploads/2019/02/computer-vision-python-3.pdf]
[29] OpenCV página oficial - [Available: **https://docs.opencv.org/master/index.html**] [Último acceso Mayo 2020]
[30] François Chollet. Libro Deep Learning with Python. Capitulo 7. Apartado 7.3.1 [Available: http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep%20Learning%20with%20Python.pdf] [Último acceso Abril 2020]
[31] Página Oficial Keras [Available: https://keras.io/api/optimizers/] [Último Acceso Junio 2020]
[32] JosephRedmon, Santosh Divvala, Ross Girshick, Ali Farhadi - Oficial Paper: You Only Look Once [Available: https://arxiv.org/pdf/1506.02640.pdf] [Último Acceso Junio 2020]
[33] Shaoqing Ren, Kaiming He, Ross Girshick , Jian Sun - Oficial Paper: Faster R-CNN [Available: https://arxiv.org/pdf/1506.01497.pdf][Último Acceso Junio 2020]

[34] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szeged, Scott Reed, Cheng-Yang Fu, Alexander C. Berg - Oficial Paper: SSD: Single Shot MultiBox Detector [Available: https://arxiv.org/pdf/1512.02325.pdf] [Último Acceso Junio 2020]
[35] D. Chaves, S. Saikia, L. F-R, E. A. y M. T [Availabe: https://pdfs.semanticscholar.org/3346/3ed1f39185347ba0b46e7e6b1a8734e0b79d.pdf][Último Acceso Mayo 2020]
[36] Ross Girshick, Microsoft Reseach Fast R-CNN [Available: https://arxiv.org/pdf/1504.08083.pdf] [Último Acceso Mayo 2020]
[37] Karen Simonyan y Andrew Zisserman [Available: https://arxiv.org/pdf/1409.1556.pdf] [Último Acceso Junio 2020]
[38] Imagen - Fuente: Mohamed Elgendy - Libro Deep Learning for Vision Systems Capítulo 7. Apartado 7.2.3 [Último acceso Mayo 2020]
[39] Imagen. Fuente: https://mohitjainweb.files.wordpress.com/2018/03/smoothl1loss.pdf
[40] faster_rcnn_resnet101_coco [Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md] [Último Acceso: Marzo 2020]
[41] T-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J.Hays, P. Perona, D. Remanan, C. L. Zitnick, P. Dollár - Oficial Paper: Microsoft COCO [Available: https://arxiv.org/pdf/1405.0312.pdf][Último Acceso Mayo 2020]
[42] TensorFlow - página oficial [Available: https://www.tensorflow.org/tutorials/load_data/tfrecord#tfrecord_files_in_python] [Último Acceso Mayo 2020]
[43] Código - [Available: https://gist.github.com/rotemtam/a1f92ab7f3444cf61da305539db4997b] [Último Acceso Abril 2020]
[44] Código Api TensorFlow [Available: https://github.com/tensorflow/models/tree/master/research/object_detection] [Último Acceso Junio 2020]
[45] Código Api TensorFlow [Available: https://github.com/tensorflow/models/blob/master/research/object_detection/legacy/train.py] [Último Acceso Junio 2020]
[46] Código Api TensorFlow [Available: https://github.com/tensorflow/models/blob/master/research/object_detection/export_inference_graph.py] [Último Acceso Junio 2020]
[47] Código GitHub [Available: https://github.com/bourdakos1/Core-ML-Custom-Object-Detection/blob/master/object_detection/object_detection_runner.py] [Último Acceso Junio 2020]
[48] Código GitHub Api TensorFlow [Available: https://github.com/tensorflow/models/blob/master/research/object_detection/object_detection_tutorial.ipynb] [Último Acceso Junio 2020]

[49] Código GitHub Api TensorFlow
[Available:https://github.com/tensorflow/models/blob/master/research/object_detection/utils/object_detection_evaluation.py] [Último Acceso Junio 2020]

ANEXO I: ALINEACIÓN DEL PROYECTO CON LOS ODS

Los objetivos de desarrollo sostenible (ODS) surgieron con el fin de concluir con la pobreza, garantizar la paz para todos y conseguir la protección del planeta. Los objetivos en total son 17, uno de ellos, específicamente el 3, trata el tema de salud y bienestar.

Las aplicaciones de machine learning cada vez son más utilizadas para aplicaciones médicas. Un ejemplo ya utilizado es el uso de las redes neuronales para detectar, con modelos de clasificación, si un paciente padece o no un cáncer.

En la siguiente tabla observamos los distintos ODS en los cuales puede aportar el trabajo desarrollado

Dimesión ODS	**Identificación del ODS**	**Papel**	**Objetivo**
Biosfera	-	-	-
Sociedad	ODS3: En el objetivo de salud y bienestar se promueve la garantía de una vida sana y encontrar el bienestar para todos en todas las edades.	Primario	En este trabajo se trabaja con moléculas de ADN, que tal y como se comprueba a lo largo del proyecto, las distintas moléculas, transformadas en triángulos, según la imagen se distribuyen en diferentes posiciones, unas más alejadas y otras más cercanas, esto es debido a que según las necesidades del paciente la aplicación de esta técnica es diferente. El estudio de estas características ayuda a la investigación de esta técnica para el uso de medicamentos según ciertas enfermedades
Economía	ODS8: En el objetivo de trabajo decente y crecimiento económico, busca la manera de que crezca la economía sostenida, inclusiva y sostenible, además de el empleo pleno y productivo y el trabajo decente para todos	Secundario	En este proyecto se analizan distintos modelos de inteligencia artificial, dichos modelos pueden adaptarse a otros problemas que puedan ayudar en el trabajo decente. El uso de cámaras con programas de inteligencia artificial que detecte personas y su comportamiento puede controlar, desde las horas de trabajo cumplidas, con el fin de que no se excedan más de lo debido, hasta el analisis la edad de la persona, para que los menores no sean explotados.

En la siguiente figura 1 se puede comprobar el impacto de la inteligencia artificial en los objetivos de desarrollo sostenible [1]

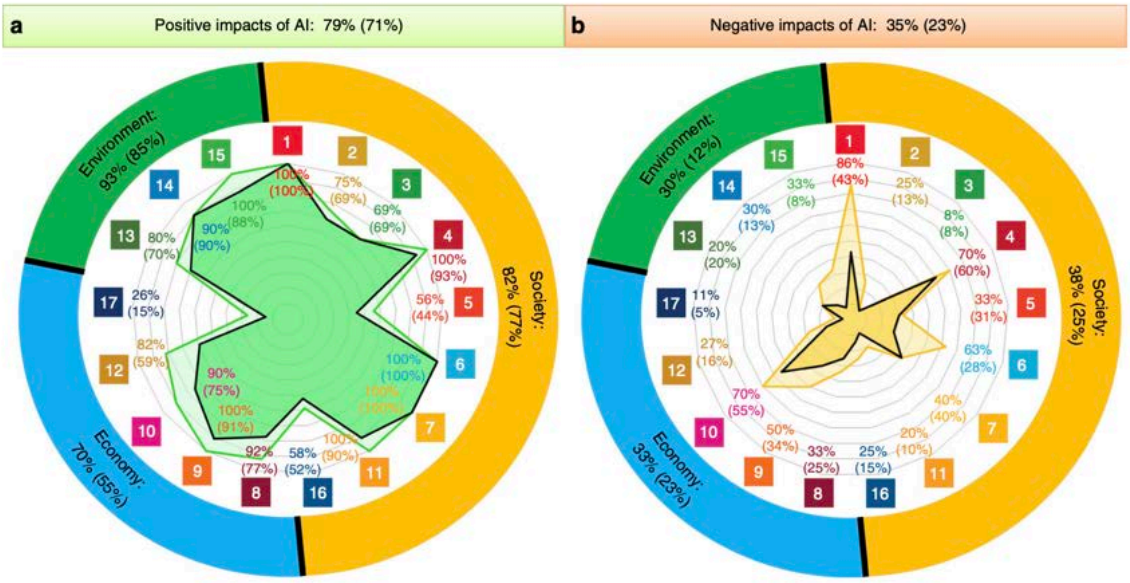


***Figura 1**. Resumen del impacto positivo y negativo que supone la inteligencia artificial en los ODS [1]*

[1] The role of artifical intelligene in achieving the Sustainable Development Goals – R.V, H. A, I. L, M.N, V.D, S.D, A.F, S.D.L, M.T & F.F.N [Available: https://www.nature.com/articles/s41467-019-14108-y.pdf]

ANEXO II: CÓDIGO

XMLToCSVpy

```
import os
import pandas as pd
import xml.etree.ElementTree as ET
import sys

def xml_to_csv(path):
    #Creamos un diccionario con una lista donde se almacenaran los valores
    d={
    'filename' : [],
    'width': [],
    'height': [],
    'class': [],
    'xmin':  [],
    'ymin': [],
    'xmax': [],
    'ymax': []
    }
    #Recorremos unicamente los archivos xml
    for file in os.listdir(path):
        if '.xml' in file: #Recorremos únicamente los archivos xml
            tree = ET.parse(file) #Abrimos el xml
            for e in tree.iter('object'):
                d['filename'].append(tree.getroot().find('filename').text)

d['width'].append(int(tree.getroot().find('size').find('width').text))

d['height'].append(int(tree.getroot().find('size').find('height').text))
                d['xmin'].append(int(e.find('bndbox').find('xmin').text))
                d['ymin'].append(int(e.find('bndbox').find('ymin').text))
                d['xmax'].append(int(e.find('bndbox').find('xmax').text))
                d['ymax'].append(int(e.find('bndbox').find('ymax').text))
    return pd.DataFrame(d)


def main():
    #Definirmos la ruta donde se encuentran los xml de entrenamiento
    img_en = os.path.join(os.getcwd(),'img_entrenamiento')
    df_e = xml_to_csv(img_en) #Llamamos a la función que devuelve el df con los
datos importantes
    df_e.to_csv(os.path.join(os.getcwd(),'CSV','entrenamiento.csv'), index=None)
    #Definirmos la ruta donde se encuentran los xml para el test
    img_t = os.path.join(os.getcwd(),'img_test')
    df_t = xml_to_csv(img_t) #Llamamos a la función que devuelve el df con los
datos importantes
    df_t.to_csv(os.path.join(os.getcwd(),'CSV','test.csv'), index=None)
```

```
if __name__=='__main__':
    main()
```

CrearDatos.py

```
# Importamos librerias necesarias
import pandas as pd
import csv
import cv2

# Tenemos las coordenadas en las imagenes de los triangulos en el archivo csv
# sacados del fichero XMLToCSV.py
# leemos los csv y lso unimos
df=pd.concat([pd.read_csv('./csv/entrenamiento.csv'),pd.read_csv('./csv/test.csv'
)],axis=0)

# Definimos la ruta donde se encuentran las imagenes
path='./csv/'

# Nos quedamis unicamente los valores que nos interesan de la tabla
df=df[['filename','xmin','ymin','xmax','ymax']]

# Almacenamos en una lista los valores de cada fila
# en cada fila tenemos:
#      1. La imagen a la que pertenece el triangulo -> valor en 0
#      2. El valor de la coordenada xmin dento de la imagen -> valor en 1
#      3. El valor de la coordenada ymin dento de la imagen -> valor en 2
#      4. El valor de la coordenada xmax dento de la imagen -> valor en 3
#      4. El valor de la coordenada ymax dento de la imagen -> valor en 4

lista = [list(l) for l in df.values]
# Definimos un iterador para almacenar las imagenes recortadas
# sin que se superpongan por coincidir en nombre al guardarlas
i=0

# Definimos donde se van a almacenar los triangulos recortados
path2='./CNN/positive_images/'

# Creamos un bucle que itere sobre los valores de cada fila de la tabla
# en la cual tenemos todos los valores necesarios para recortar todos los
triangulos
# de cada imagen

for vect in lista:
    img=cv2.imread(path + vect[0]) #Leemos la imagen
    crop_img=img[vect[2]:vect[4],vect[1]:vect[3]] #la recortamos
    # recordamos que estamos recortando la imagen img[ymin:ymax,xmin:xmax]
    cv2.imwrite(path2+str(i) + '.png',crop_img) # Guardamos la imagen recortada
    i=i+1 #Aumentamos en uno el iterador para que no se superpongan las imagenes
al guardarlas
```

```
# Para crear las imagenes negativas definimos una nueva ruta donde almacenarlas
path2='./CNN/negative_images/'
# Iterador lo inicializamos a 0
i=0

# Creamos una lista con los nombres de las imagenes a recortar
filename=[str(i+1) + '.png' for i in range(15)]

# Definimos la anchura y la altura de la imagen recortada que sera la misma
(w,h)=(34,34)
# Definimos la anchura y la altura de la imagen a recortar
(w2,h2)=cv2.imread(path+filename[0]).shape[:2]

# Recorremos cada imagen
for file in filename:
      img=cv2.imread(path + file) # La leemos y la almacenamos en una variable
      # Iteramos sobre el eje y de la imagen (cada punto de y es un pixel)
    for y in range(0,w2-w,h): # Se itera de 0 a w2-w(ya que se va a recortar
(1026-34=992)) de 34 en 34 pixeles
        for x in range(0,w2-w,w):
            crop_img=img[y:y+h,x:x+w] #Recortamos la imagen (cuando y e x sean
igual a 992 img[992:1026,992:1026] asi no excede el tamaño)
            cv2.imwrite(path2+str(i) + '.png',crop_img) # Guardamos la imagen
recortada
            i=i+1 # Aumentamos en 1 el iterador
```

Clasificacion.py

```
import keras
import os, shutil
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
import os , shutil
from keras import layers
from keras import models
from keras import optimizers


# Para empezar definimos las rutas donde se encuentran las imagenes positivas ->
los triangulos
# y las imagenes negativas -> los no triangulos
path_pos='./CNN/positive_images'
path_neg='./CNN/negative_images'


# A continuación crearemos tres carpetas
```

```
train_dir='./CNN/train/' #En la primera se encontraran las imagenes con las que
se entrenara la red
test_dir='./CNN/test/' #En la segunda se encontraran las imagenes con las que se
realizara el test
# Para ver como funciona la red para imagenes que no ha visto
validation_dir='./CNN/validation/' # Finalmente una carpeta donde se guardaran
# las imagenes de validacion para probar la red en el proceso de entrenamiento

# Existen dos clases
#      1. Imagenes positivas (pos) -> contienen triangulos -> se alamacenaran en
una carpeta llamada pos
#             dentro de cada carpeta de entrenamiento, validacion y test
#      2. Imagenes negativas (neg) -> No contienen triangulos

# Creamos una carpeta de la clase pos y neg en la carpeta de entrenamiento
train_pos_dir=os.path.join(train_dir + 'pos')
train_neg_dir=os.path.join(train_dir + 'neg')
# En la carpeta de test
test_pos_dir=os.path.join(test_dir + 'pos')
test_neg_dir=os.path.join(test_dir + 'neg')
# Y en la carpeta de validacion
validation_pos_dir=os.path.join(validation_dir + 'pos')
validation_neg_dir=os.path.join(validation_dir + 'neg')


# Las imagenes se llaman por numeros + la extensión de la imagen
# la manera de guardarlas con nuemeros facilita el codgio
# Guardamos en una lista el nombre del 80% de las imagenes positivas
filenames=['{}.png'.format(i) for i in range(7560)]
for file in filenames: #Recorremos los ficheros
       # Copiamos las imagenes hasta la 7560 (el 80% del total) en la carpeta
creada
       # para las imagenes positivas del entrenamiento
    shutil.copyfile(os.path.join(path_pos,file),os.path.join(train_pos_dir,file))

# Hacemos lo mismo para el resto de carpetas de test y validacion
# y para las imagenes negativas tambien
filenames=['{}.png'.format(i) for i in range(7560,8505)]
for file in filenames:
    shutil.copyfile(os.path.join(path_pos,file),os.path.join(test_pos_dir,file))

filenames=['{}.png'.format(i) for i in range(8505,9449)]
for file in filenames:

shutil.copyfile(os.path.join(path_pos,file),os.path.join(validation_pos_dir,file)
)


filenames=['{}.png'.format(i) for i in range(11532)]
for file in filenames:
    try: # Debido a que se borraron las imagenes que contenian triangulos en la
carpeta de negativos
    # algunas imagenes no existen y se han de saltar
```

```
shutil.copyfile(os.path.join(path_neg,file),os.path.join(train_neg_dir,file))
    except:
        continue

filenames=['{}.png'.format(i) for i in range(11532,12973)]
for file in filenames:
    try:

shutil.copyfile(os.path.join(path_neg,file),os.path.join(test_neg_dir,file))
    except:
        continue

filenames=['{}.png'.format(i) for i in range(12973,14415)]
for file in filenames:
    try:

shutil.copyfile(os.path.join(path_neg,file),os.path.join(validation_neg_dir,file)
)
    except:
        continue


# PREPARACIÓN DE LOS DATOS -> Pasamos las imagenes a tensores

# Los reescalamos para que los valores de los pixeles estén entre el 0 y el 1 ->
formato float
traindatagen= ImageDataGenerator(rescale=1./255)
testdatagen=ImageDataGenerator(rescale=1./255)
test_datagen=ImageDataGenerator(rescale=1./255)

# Guardamos las imagenes como tensores de tipo float.
# Para ello se hace uso de la función flow_from_directory
# ya que los datos se encuentran en el directorio
# Les decimos donde se encuentran las imagenes
# El tamaño que han de poseer las imagenes para entrar a la red
# Clase es de modo binario -> 1 si es un triangulo y 0 si no
# Batch size son la cantidad de datos con los que aprenderá la red
# en cada paso -> en este caso 64
train_generator=traindatagen.flow_from_directory(train_dir,target_size=(34,34),ba
tch_size=64,class_mode='binary')
validation_generator=testdatagen.flow_from_directory(validation_dir,target_size=(
34,34),batch_size=64,class_mode='binary')
test_generator=test_datagen.flow_from_directory(test_dir,target_size=(34,34),batc
h_size=64,class_mode='binary')

# Creamos el modelo

model=models.Sequential()
#BLOQUE1
model7.add(layers.SeparableConv2D(8,(3,3),activation='relu',input_shape=(34,34,3)
,padding='same'))
model7.add(layers.BatchNormalization())
```

```
model7.add(layers.MaxPooling2D((2,2)))
#BLOQUE2
model7.add(layers.SeparableConv2D(16,(3,3),activation='relu'))
model7.add(layers.BatchNormalization())
model7.add(layers.MaxPooling2D((2,2)))
#BLOQUE3
model7.add(layers.SeparableConv2D(32,(3,3),activation='relu'))
model7.add(layers.BatchNormalization())
model7.add(layers.MaxPooling2D((2,2)))
# BLOQUE4
model7.add(layers.Flatten())
model7.add(layers.Dense(512, activation='relu'))
model7.add(layers.BatchNormalization())
model7.add(layers.Dense(1,activation='sigmoid'))

# Un callback es un objeto que implementa objetos específicos
# Es llamado por el modelo en determinados puntos de entrenamiento
# Tiene acceso a los datos y sobre el estado del modelo y su ejecucion

# Para pasar los callbacks al modelo se pasan por un argumento callback en el fit
# Recibe una lista de callbacks, pueden pasarse cualquier cantidad de callbacks

callbacks_list = [
    # Interrumpe el modelo cuando deja de aprender
    # Se define la precisión para medir el aprendizaje de la red
    # Interrumpe el modelo cuando la precisión deja de mejorar para un número
mayor de (patience=9) iteración
    keras.callbacks.EarlyStopping(monitor='val_binary_accuracy',patience=9),
    # Guarda los pesos en cada iteración
    keras.callbacks.ModelCheckpoint(filepath='Model7.h5', #donde se guarda el
modelo
                                    monitor='val_loss', #no se sobreescribe el
modelo a no ser
                                     # que la pérdida en los datos de validación
no haya mejorado
                                    save_best_only=True),
    #Reduce la tasa de aprendizaje cuando deja de mejorar el aprendizaje
    # manera de evitar el local minima
    # La metrica que monitoriza la reducción es la perdida de validacion
    # por cada 10 epochs que ha dejado de aprender, reduce la tasa de aprendizaje
entre 10
    keras.callbacks.ReduceLROnPlateau(monitor='val_loss',factor=0.1,patience=5),

    #Creamos un callback TensorBoard
    # Una manera de ver como funciona el modelo y su arquitectura
    # como aprende durante el entrenamiento
    keras.callbacks.TensorBoard(
    log_dir='my_log_dir_1',
    histogram_freq=1, #Registra los histogramas de activación cada epoch
    write_graph=True,
    write_images=True,
    update_freq='epoch',
    profile_batch=20,
```

```
    embeddings_freq=0
    )
]



#Definimos los parámetros
model7.compile(optimizer=optimizers.Adam(learning_rate=0.01),loss='binary_crossen
tropy',metrics=['binary_accuracy'])

#steps_per-epoch=295 para que en cada paso procese 64(definido anteriormente)
imagenes y analice todo el conjunto de datos
# 64x295=18890
model7.fit_generator(train_generator,steps_per_epoch=295,

epochs=50,callbacks=callbacks_list,validation_data=validation_generator,
                    validation_steps=29)
```

DataAugmentationForFasterRCNN.py

```
import os
import scipy.misc
import cv2
from xml.dom import minidom
import xml.etree.cElementTree as ET
import numpy as np

#Ruta a la carpeta en la que se encuentran las imagenes junto con sus respectivos
xmls
src='./img_xml/'

def girarCoordenadas(x):
    return 1026 - int(x)

def generarNuevoDato(img,d,tree,x):
    annotation=ET.Element('annotation')


    folder=ET.SubElement(annotation,'folder')
    folder.text='triangulo'


    filename=ET.SubElement(annotation,'filename')
    filename.text='.'.join([str(e) + '_t' for e in
str(tree.find('filename').text).split('.')])[:-2]
```

```
    path=ET.SubElement(annotation,'path')
    path.text='.'.join([str(e) + '_t' for e in
str(tree.find('path').text).split('.')])[:-2]


    source=ET.SubElement(annotation,'source')
    database=ET.SubElement(source,'database')
    database.text='Unknown'


    size=ET.SubElement(annotation,'size')
    width=ET.SubElement(size,'width')
    height=ET.SubElement(size,'height')
    depth=ET.SubElement(size,'depth')
    width.text='1026'
    height.text='1026'
    depth.text='3'


    segmented= ET.SubElement(annotation,'segmented')
    segmented.text='0'

    for i in range(len(d['xmin'])): #La cantidad de triangulos
        obj=ET.SubElement(annotation,'object')
        name=ET.SubElement(obj,'name')
        name.text='triangulo'

        pose=ET.SubElement(obj,'pose')
        pose.text='Unspecified'

        truncated=ET.SubElement(obj,'truncated')
        truncated.text='0'

        difficult=ET.SubElement(obj,'difficult')
        difficult.text='0'

        bndbox=ET.SubElement(obj,'bndbox')

        xmin=ET.SubElement(bndbox,'xmin')
        xmin.text=str(d['xmin'][i])
        ymin=ET.SubElement(bndbox,'ymin')
        ymin.text=str(d['ymin'][i])
        xmax=ET.SubElement(bndbox,'xmax')
        xmax.text=str(d['xmax'][i])
        ymax=ET.SubElement(bndbox,'ymax')
        ymax.text=str(d['ymax'][i])


    doc=ET.ElementTree(annotation)
    #Guardamos xml junto con la image
    src_save='./img_xml/' + '.'.join([str(e) + '_t'+ str(x) for e in
str(tree.find('filename').text).split('.')])[:-
(2+len(str(x)))].replace('.png','.xml')
```

```
    doc.write(src_save,xml_declaration=True,method='xml')
    cv2.imwrite(src_save.replace('.xml','.png'),img)

d=dict()
x=0
for file in os.listdir(src): #Recorremos todos los archivos xml
#Primero comprobamos si se trata de un xml
    if '.xml' in file and 't' not in file:
        #Definimos un diccionario con todos los nombres del archivo junto con los
nuevos valores de las coordenadas
        d[src+file]={
            'xmin':[],
            'xmax':[],
            'ymin':[],
            'ymax':[]
        }
        #Si el fichero es un xml lo leemos
        tree=ET.parse(src+file)
        #leemos también la imagen -> se llama igual que su fichero xml
correspondiente
        #Reemplazamos las extensiones
        img=cv2.imread(src+file.replace('xml','png'))
        #Aplicamos distintas transformaciones
        #1. Giramos la imagen horizontalmente y verticalmente
        img_v=cv2.flip(img,-1)
        #Al darle la vuelta a las imagenes es necesario cambiar los valores de
las coordenadas de x e y

        #Para ello es necesario cambiar todos los valores de las coordenadas
dentro del fichero xml
        for e in tree.iter('object'):
        # En el xml la etiqueta xmin se encunetra dentro de bndbox que se
encuentra dentro de object
            xmin=e.find('bndbox').find('xmin').text
            xmax=e.find('bndbox').find('xmax').text
            ymin=e.find('bndbox').find('ymin').text
            ymax=e.find('bndbox').find('ymax').text
        # El nuevo valor de y será y'=Ymax-y. Siendo y' el nuevo valor.
Ymax=1026(altura de la imagen) e y el valor de y de la imagen original
        #Se aplica la misma función para la x. Para llevar a cabo este proceso se
llama a la función girarCoordenadas
            d[src+file]['xmin'].append(girarCoordenadas(xmin))
            d[src+file]['xmax'].append(girarCoordenadas(xmax))
            d[src+file]['ymin'].append(girarCoordenadas(ymin))
            d[src+file]['ymax'].append(girarCoordenadas(ymax))

        generarNuevoDato(img_v,d[src+file],tree,x)

        #Generamos ahora únicamente una vuelta horizontal -> Únicamente varían
las coordenadas del eje x
        img_v2=cv2.flip(img,1)
        #Vaciamos la lista del diccionario
        d[src+file]['xmin']=[]
```

```
        d[src+file]['xmax']=[]
        d[src+file]['ymin']=[]
        d[src+file]['ymax']=[]

        for e in tree.iter('object'):
        # En el xml la etiqueta xmin se encunetra dentro de bndbox que se
encuentra dentro de object
            xmin=e.find('bndbox').find('xmin').text
            xmax=e.find('bndbox').find('xmax').text
            ymin=e.find('bndbox').find('ymin').text
            ymax=e.find('bndbox').find('ymax').text
        # El nuevo valor de y será y'=Ymax-y. Siendo y' el nuevo valor.
Ymax=1026(altura de la imagen) e y el valor de y de la imagen original
        #Se aplica la misma función para la x. Para llevar a cabo este proceso se
llama a la función girarCoordenadas
            d[src+file]['xmin'].append(girarCoordenadas(xmin))
            d[src+file]['xmax'].append(girarCoordenadas(xmax))
            d[src+file]['ymin'].append(ymin)
            d[src+file]['ymax'].append(ymax)

        generarNuevoDato(img_v2,d[src+file],tree,x+1)

        #Generamos únicamente una vuelta vertical -> únicamente varían las
coordenadas del eje y
        img_v3=cv2.flip(img,0)
        #Vaciamos la lista del diccionario
        d[src+file]['xmin']=[]
        d[src+file]['xmax']=[]
        d[src+file]['ymin']=[]
        d[src+file]['ymax']=[]

        for e in tree.iter('object'):
        # En el xml la etiqueta xmin se encunetra dentro de bndbox que se
encuentra dentro de object
            xmin=e.find('bndbox').find('xmin').text
            xmax=e.find('bndbox').find('xmax').text
            ymin=e.find('bndbox').find('ymin').text
            ymax=e.find('bndbox').find('ymax').text
        # El nuevo valor de y será y'=Ymax-y. Siendo y' el nuevo valor.
Ymax=1026(altura de la imagen) e y el valor de y de la imagen original
        #Se aplica la misma función para la x. Para llevar a cabo este proceso se
llama a la función girarCoordenadas
            d[src+file]['xmin'].append(xmin)
            d[src+file]['xmax'].append(xmax)
            d[src+file]['ymin'].append(girarCoordenadas(ymin))
            d[src+file]['ymax'].append(girarCoordenadas(ymax))

        generarNuevoDato(img_v3,d[src+file],tree,x+2)


        #Cambiamos el color de la imagen
        #Las coordenadas en este caso se mantienen
        #Definimos parametros de manipulación de imagenes
```

```
        phi = 0.8
        theta = 0.7
        max_int=255.0
        img_v4=np.array((max_int/phi)*(img/(max_int/theta))**0.5,dtype='uint8')

        #Vaciamos la lista del diccionario
        d[src+file]['xmin']=[]
        d[src+file]['xmax']=[]
        d[src+file]['ymin']=[]
        d[src+file]['ymax']=[]

        for e in tree.iter('object'):
        # En el xml la etiqueta xmin se encunetra dentro de bndbox que se
encuentra dentro de object
            xmin=e.find('bndbox').find('xmin').text
            xmax=e.find('bndbox').find('xmax').text
            ymin=e.find('bndbox').find('ymin').text
            ymax=e.find('bndbox').find('ymax').text
        # El nuevo valor de y será y'=Ymax-y. Siendo y' el nuevo valor.
Ymax=1026(altura de la imagen) e y el valor de y de la imagen original
        #Se aplica la misma función para la x. Para llevar a cabo este proceso se
llama a la función girarCoordenadas
            d[src+file]['xmin'].append(xmin)
            d[src+file]['xmax'].append(xmax)
            d[src+file]['ymin'].append(ymin)
            d[src+file]['ymax'].append(ymax)

        generarNuevoDato(img_v4,d[src+file],tree,x+3)
```

Faster rcnn resnet101 coco.config

```
# Faster R-CNN with Resnet-101 (v1) configuration for MSCOCO Dataset.
# Users should configure the fine_tune_checkpoint field in the train config as
# well as the label_map_path and input_path fields in the train_input_reader and
# eval_input_reader. Search for "PATH_TO_BE_CONFIGURED" to find the fields that
# should be configured.

model {
  faster_rcnn {
#Número de clases
    num_classes: 1
    image_resizer {
 #Dimensiones de las imágenes
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1026
#Cambiamos el valor debido a que nuestras imágenes son de un tamaño 1026x1026
      }
```

```
    }
    feature_extractor {
      type: 'faster_rcnn_resnet101'
      first_stage_features_stride: 16
    }
    first_stage_anchor_generator {
       #Características de las cajas de anclas explicadas en el capitulo 4
      grid_anchor_generator {
       #escala de las imágenes (tamaños)
        scales: [0.25, 0.5, 1.0, 2.0]
       # Relaciones de aspectos de la caja delimitada
        aspect_ratios: [0.5, 1.0, 2.0]
       #Pasos que da la caja ancla de altura y anchura
        height_stride: 16
        width_stride: 16
      }
    }
    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
       #Truncated normal selecciona números de una distribución normal
       # en el cual la media es cercana al 0 y sus valores también, con una
       # desviación típica de 0.01. Se denomina truncated decido a que se
eliminan las colas de la distribución normal
        truncated_normal_initializer {
          stddev: 0.01
        }
      }
    }

    first_stage_nms_score_threshold: 0.0
# El umbral de objecteness score es 0.6, si sobrepasa este valor se considera un
posible objeto a pasar por el clasificador
    first_stage_nms_iou_threshold: 0.6
#En el primer componente como máximo pueden definirse 1100 regiones de posibles
objetos
    first_stage_max_proposals: 1100
#Peso de perdida de localización 2.0 en la primera etapa y 1 para la pérdida de
si es un objeto o no
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
    initial_crop_size: 20
# Matriz de maxpooling de 2x2 y saltos de 2 en dos pixeles
    maxpool_kernel_size: 2
    maxpool_stride: 2
# Pasamos a definir el segundo componente
    second_stage_box_predictor {
      mask_rcnn_box_predictor {
```

```
# Dropout es una técnica de pérdida de información para no caer en sobreajuste
        use_dropout: false
        dropout_keep_probability: 1.0
        fc_hyperparams {
          op: FC
# Regularización es otra manera de no caer en sobreajuste, existen dos tipos l1 y
l2
          regularizer {
#Se añade un coste para forzar a los pesos a tener valores pequeños, hace las
distribuciones de los valores de los pesos más regular. Se consigue añadiendo a
la función de pérdida de la red un coste asociado a tener pesos de un valor alto
            l2_regularizer {
# no se utiliza
              weight: 0.0
            }
          }
          initializer {
            variance_scaling_initializer {
              factor: 1.0
              uniform: true
              mode: FAN_AVG
            }
          }
        }
      }
    }
# Pasamos a la configuración de los resultados
    second_stage_post_processing {
# Como se ha explicado en el capítulo 3, el NonMax-Suppression es una técnica
para que una clase no esté rodeada por el mismo cuadrado delimitador
      batch_non_max_suppression {
        score_threshold: 0.0
# Dos cuadrados rodean al mismo objeto si la operación de iou, explicada en el
capítulo 3, de ambos superan el umbral de 0.5
        iou_threshold: 0.5
#Como máximo se pueden detectar 1000 triángulos en una imagen
        max_detections_per_class: 1000
#Mismo valor debido a que únicamente se ha definido una clase
        max_total_detections: 1000
      }
#Función de activación
      score_converter: SOFTMAX
    }
    second_stage_localization_loss_weight: 2.0
    second_stage_classification_loss_weight: 1.0
  }
}
# Parámetros del proceso de entrenamiento
train_config: {
# Se procesa únicamente una imagen por entrenamiento, es suficiente ya que una
imagen puede poseer hasta una aproximación de 1000 triangulos
  batch_size: 1
  optimizer {
```

```
    momentum_optimizer: {
      learning_rate: {
# Tasa de aprendizaje, basándose en el gradiente
        manual_step_learning_rate {
          initial_learning_rate: 0.0003
          schedule {
            step: 0
            learning_rate: .0003
          }
          schedule {
            step: 900000
            learning_rate: .00003
          }
          schedule {
            step: 1200000
            learning_rate: .000003
          }
        }
      }
# El momento es la rapidez a la que aprende el modelo, tal y como se explico en
el inicio de la memoria
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  gradient_clipping_by_norm: 10.0
#Donde se guarda el modelo
  fine_tune_checkpoint: "modelo/model.ckpt"
  from_detection_checkpoint: true
  # Note: The below line limits the training process to 200K steps, which we
  # empirically found to be sufficient enough to train the pets dataset. This
  # effectively bypasses the learning rate schedule (the learning rate will
  # never decay). Remove the below line to train indefinitely.
#Número de pasos máximo que puede dar el entrenamiento, se ha reducido debido a
que únicamente se entrena para una clase
  num_steps: 100000
  data_augmentation_options {
# Técnicas de aumento de datos
    random_horizontal_flip {
    }
  }
}

train_input_reader: {
#Donde se encuentran los datos de entrenamiento
  tf_record_input_reader {
    input_path: "TFRecords/entrenamiento.record"
  }
  label_map_path: "configuracion/label_map.pbtxt"
}

eval_config: {
  num_examples: 8000
```

```
  # Note: The below line limits the evaluation process to 10 evaluations.
  # Remove the below line to evaluate indefinitely.
  max_evals: 10
}

eval_input_reader: {
# Donde se encuentran los datos de validación
  tf_record_input_reader {
    input_path: "TFRecords/test.record"
  }
  label_map_path: "configuracion/label_map.pbtxt"
  shuffle: false
  num_readers: 1
  num_epochs: 1
}
```

FasterRCNN.py

```
from tkinter import *
from tkinter import filedialog
import os
import numpy as np
import sys
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

import json
from PIL import Image
import matplotlib.pyplot as plt


from utils import visualization_utils as vis_util
from utils import label_map_util



#Pedimos al usuario que escoga la imagen a procesar por el modelo
Tk().withdraw()
PATH_IMG=filedialog.askopenfilename(initialdir
="~/Desktop/deteccion_objetos3/",title="Seleccione una imagen",filetypes=(('jpeg
files','.jpg'),('png files','.png')))



#Donde se encuntran las clases configuradas
PATH_LABELS = 'configuracion/label_map.pbtxt'

#Genera un mapa de etiquetas
label_map = label_map_util.load_labelmap(PATH_LABELS)
```

```
# Devuelve una lista de diccionarios, de cada clase, con las siguientes etiquetas
# ID el identificador de la clase. Un uno para los triangulos
# name el nombre de la clase -> Triangulo
categories=
label_map_util.convert_label_map_to_categories(label_map,max_num_classes=1000,use
_display_name=True)
# Devuelve un diccionario en el que esta vez la clave de las clases es el
identificador
CATEGORY_INDEX=label_map_util.create_category_index(categories)

# Ruta donde se encuentra el modelo congelado
PATH_MODEL='modelo_congelado/frozen_inference_graph.pb'

# Función que convierte la imagen en un array
def load_image_into_numpy_array(img):
      #Primero cogemos los valores de su altura y anchura
      (w,h) = img.size
      #Lo pasamos a un array con la misma de forma (hxwx3)
      return np.array(img.getdata()).reshape((h,w,3)).astype(np.uint8)

def detect_objects(img_path):
      #Cargamos la imagen
      img=Image.open(img_path)
      #La pasamos a un array
      img_np=load_image_into_numpy_array(img)
      # Preparamos los datos de entrada al modelo, tipo -> image_tensor
      img_np_expand=np.expand_dims(img_np,axis=0)

      #Sacamos las predicciones pasan
      # Con sess.run estamos ejecutando el modelo
      (boxes,scores,classes,num)=sess.run([detection_boxes,detection_scores,dete
ction_classes,num_detections], feed_dict={image_tensor: img_np_expand})

      #Función que pinta en la imagen las cajas delimitadoras con sus
porcentajes y etiquetas de clase
      # No devuelve nada ya que varia la imagen en si
      vis_util.visualize_boxes_and_labels_on_image_array(
            img_np,
            np.squeeze(boxes), #Todas las dimensiones dentro del array de tamaño
1 se eliminan
            np.squeeze(classes).astype(np.int32),
            np.squeeze(scores),
            CATEGORY_INDEX,
            #Minima probabilidad de la imagen para que se pinte
            min_score_thresh=0.8,
            use_normalized_coordinates=True,
            line_thickness=2
            )

      #Para guardar la imagen
      fig=plt.figure(figsize=(1026,1026))
      fig.set_size_inches(16,9)
      ax=plt.Axes(fig,[0.,0.,1.,1.])
```

```
        ax.set_axis_off()
        fig.add_axes(ax)
        plt.imshow(img_np,aspect='auto')
        plt.savefig('output/{}'.format(img_path[-5]),dpi=62)
        plt.close(fig)
    # Una vez guardada se acaba el programa


print('Cargamndo el modelo')
#Creamos el grafo
# Los programas de tensorflow primero se construyen y luego se ejecutan
# Para la ejecución es necesario el uso de una sesión
# El grafo se utiliza para representar la red neuronal en la fase de construcción
# Para ejecutarlo hace uso de lo que se denomina como ops entrenadas, las cuales
salen
# del archivo congelado
detection_graph=tf.Graph()
#Implica que es por defecto el grafo principal
with detection_graph.as_default():
        od_graph_def=tf.GraphDef() #Crea un grafo desde un fichero GraphDef
        with tf.gfile.GFile(PATH_MODEL,'rb') as fid: #Leemos el archivo donde se
almacena el modelo
                od_graph_def.ParseFromString(fid.read()) # fid.read devuelve los
contenidos del fichero como string
                tf.import_graph_def(od_graph_def,name='') # Se carga el modelo en el
grafo

with detection_graph.as_default(): #Volvemos a procesar el grafo pero esta vez
lleva el modelo en su interior
        with tf.Session(graph=detection_graph) as sess: #Creamos una sesión para
poder hacer uso del modelo
        #En TensorFlow se trabaja con tensores, estructura de datos que representa
cualquier tipo de dato
        # Un tensor es como una lista o un array de n-dimensiones
        # Los tensores pasan por las operaciones del grafo y pueden variar
        # Los parametros del modelo estan definidos como tensores
        #Primero el tensor que se encarga de la imagen de entrada al modelo del
tipo image_tensor
                image_tensor=detection_graph.get_tensor_by_name('image_tensor:0')
                #El tensor que se encarga de la predicción de cajas delimitadoras

        detection_boxes=detection_graph.get_tensor_by_name('detection_boxes:0')
                # Tensor encargado de las probabilidades de clasificación

        detection_scores=detection_graph.get_tensor_by_name('detection_scores:0')
                # Tensor encargado de definir la clase de lo que rodea la caja

        detection_classes=detection_graph.get_tensor_by_name('detection_classes:0'
)
                # Numero total de triangulos encontrados en la imagen

        num_detections=detection_graph.get_tensor_by_name('num_detections:0')
```

```
            #Llamamos a la función encargada de predecir y dibujar los cuadrado
alrededor de los triangulos
            detect_objects(PATH_IMG)
```

ResultadosClasificador.py

```
# Importamos librerias
import numpy as np
import sys
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import json
from PIL import Image
from nms import nms
from keras.models import load_model
import cv2
import pandas as pd
import skimage.io

from object_detection.utils import object_detection_evaluation as od

# Leemos el modelo del clasificador
model=load_model('Model7.h5')

# Necesitamos los csv con los valores reales
df=pd.read_csv('CSV/entrenamiento.csv')
df2=pd.read_csv('CSV/test.csv')

# Unimos ambos
df=pd.concat([df,df2],axis=0)

df['scores']=1

# Creamos una función que itere la imagen
def iteroImg(path):
    t=34 # altura y anchura del las imágenes que entran en el clasificador
    img=cv2.imread(path) # leemos la imagen a recortar
    lista_bb=[] #creamos una lista que almacene las coordenadas
    scores=[] #creamos otra lista que almacene las predicciones
# Iteramos sobre el eje y de la imagen (cada punto de y es un pixel)
    for y in range(0,1026-int(t/2),int(13)): # Se itera la imagen de 13 pixeles
en 13 pixeles
        for x in range(0,1026-int(t/2),int(13)): # Lo mismo en el eje x
            crop_img=img[y:y+t,x:x+t] #Recortamos la imagen
            crop_img=np.expand_dims(crop_img,axis=0) # Antes de poder pasarla por
el clasificador se han de preparar los datos
            crop_img= crop_img.astype(float)/255 #la normalizamos
            if crop_img.shape[1]!=34 or crop_img.shape[2]!=34: # se ha de cumplir
el tamaño de entrada a la red
                continue
            pred=model.predict(crop_img)[0][0] #se pasa por el modelo
            if pred>=0.9:
```

```
                (cx,cy)=(x+t/2,y+t/2) #si es mayor a 0.9 se almacena en la lista
                lista_bb.append((cx,cy,34,34)) #guardamos el punto central con
respecto a las coordenadas de la imagen junto con su anchura y altura
                scores.append(pred) #Almacenamos las predicciones


    indices=nms.boxes(lista_bb,scores) # Aplicamos el algoritmo de Non-Maximum
Supression


    indices=np.array(indices).reshape(-1)
    draw_bb=np.array(lista_bb)[indices] # Nueva lista con los cuadrados no
superpuestos
    sc=np.array(scores)[indices] # nos quedamos unicamente las predicciones tras
aplicar el algoritmo

    bb_predict=[] #creamos la lista de los bounding box una vez aplicado el
algoritmo
    for (x1,y1,w1,h1) in draw_bb:
        bb_predict.append([(y1-17)/1026,(x1-17)/1026,(y1+17)/1026,(x1+17)/1026])
        # Almacenamos los valores como (ymin,xmin,ymax,xmax) normalizados

    return{
        'bb':bb_predict,
        'scores':sc
    }


# Creamos una lista que almacene los nombres de las imagenes a analizar
paths=[str(i+1) + '.png' for i in range(17) if i!=8 and i!=9]

# A continuación pasamos al análisis de resultados
# Para ello hacemos uso del fichero object_detection_evaluation.py
# sacado de la API de TensorFlow para modelos de detección de objetos

i=0 #Inicializamos un iterador para almacenar las imagenes en la clase
ObjectDetectionEvaluation y que no
# se superpongan
# Creamos la clase y específicamos que únicamente existe una clase
evaluate=od.ObjectDetectionEvaluation(num_groundtruth_classes=1)
for p in paths: # iteramos sobre la lista que almacena los nombres de las
imagenes
    # Nos quedamos con los datos de una única imagen
    df_imgx=df[df['filename']==p][['xmin','ymin','xmax','ymax','scores']]
    # Los normalizamos

df_imgx[['ymin','xmin','ymax','xmax']]=df_imgx[['ymin','xmin','ymax','xmax']]/102
6
    # Almacenamos en un numpy array los valores de las coordenadas de las cajas
en las imagenes reales
    gt_bb=np.squeeze(df_imgx[['ymin','xmin','ymax','xmax']].values)
    df_imgx['labels']=0 #Solo hay una clase y es definida como 0
```

```
    gt_labels=np.squeeze(df_imgx['labels'].values)
    # Lo añadimos a la clase como datos reales, para ello es necesario:
    #   1. Un identificador que lo relacione con los resultados predichos
    #   2. Las coordenadas normalizadas de los bounding boxes
    #   3. Las etiquetas de la clase a la que pertenecen
    evaluate.add_single_ground_truth_image_info(str(i) + p,gt_bb,gt_labels)


    # Llamamos a la funcion que predice los resultados
    p_m=iteroImg('images/'+p)
    #   Preparamos los datos
    bb=np.squeeze(p_m['bb'])
    scores=np.squeeze(p_m['scores'])
    labels=np.zeros(scores.shape)

    # Los añadimos a la clase como datos predichos
    # En esta parte es necesario pasar los valores de las predicciones
    evaluate.add_single_detected_image_info(str(i) + p,bb,scores,labels)
    i+=1 # Aumentamos en uno el iterados

# Una vez se hayan añadido todas las imagenes procedemos a llamar a la función
evaluate
# Esta función calcula el mAP , la preicision por clases,la cual es igual al mAP
debido a que únicamente
# se predice una clase. Y el Recall por clases.

res=evaluate.evaluate()
# Lo imprimimos por pantalla
print('mAP: ',res[0])
print('Precisiones por clases: ',res[1])

print('Recalls por clase',sum(res[2][0])/len(res[2][0]))
```

Resultados.py

```
import os
import numpy as np
import sys
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import pandas as pd
import skimage.io
from PIL import Image

from object_detection.utils import object_detection_evaluation as od

#Creamos una funcion que aplique el modelo a una imagen
def detect_objects(img_path='img_entrenamiento/6.png'):
    img=Image.open(img_path) #leemos la imagen
    (w,h)=img.size
    # La preparamos para entrar al modelo
```

```
img_np_expand=np.expand_dims(np.array(img.getdata()).reshape((h,w,3)).astype(np.u
int8),axis=0)
    # Aplicamos el modelo y guardamos los resultados en un diccionario
    (boxes,scores,classes,num) =
sess.run([detection_boxes,detection_scores,detection_classes,num_detections],feed
_dict={image_tensor:img_np_expand})
    d={
        'img':img_path,
        'bb': boxes,
        'scores':scores,
        'num': num
    }
    return d

# Ruta donde se encuentra el modelo congelado
PATH_MODEL='modelo_congelado/frozen_inference_graph.pb'

# Almacenamos en una lista los nombres de las imagenes
paths=[str(i+1) + '.png' for i in range(17) if i!=8 and i!=9]
# Creamos un grafo para leer el modelo
detection_graph=tf.Graph()
with detection_graph.as_default():
    od_graph_def=tf.GraphDef()
    with tf.gfile.GFile(PATH_MODEL,'rb') as fid:
        od_graph_def.ParseFromString(fid.read())
        tf.import_graph_def(od_graph_def,name='')

# Una vez leido necesitamos los tensores encargados de procesar los datos
with detection_graph.as_default():
    with tf.Session(graph=detection_graph) as sess:
        image_tensor=detection_graph.get_tensor_by_name('image_tensor:0')
        detection_boxes=detection_graph.get_tensor_by_name('detection_boxes:0')
        detection_scores=detection_graph.get_tensor_by_name('detection_scores:0')

detection_classes=detection_graph.get_tensor_by_name('detection_classes:0')
        num_detections=detection_graph.get_tensor_by_name('num_detections:0')
        d=[]  # creamos una lista vacia
        for p in paths: # recorremos todas las imagenes
            d.append(detect_objects('images/' + p)) #pedimos que nos saque los
resultados de las imagenes
           # los cuales se encontraran en orden en ua lista de diccionarios


# leemos los archivos en los que se almacenan las coordenadas reales de los
triangulos
df=pd.read_csv('CSV/entrenamiento.csv')
df2=pd.read_csv('CSV/test.csv')
df=pd.concat([df,df2],axis=0)



# Creamos la clase encargada de calcular la precisión y. el recall
```

```
evaluate=od.ObjectDetectionEvaluation(num_groundtruth_classes=1)

i=0 #Inicializamos un iterador
# Recorremos la lista de los nombres de todas las imagenes
for p in path:
       # Nos quedamos con los datos que nos interesan
    df_imgx=df[df['filename']==p][['xmin','ymin','xmax','ymax','scores']]
    #los normalizamos

df_imgx[['ymin','xmin','ymax','xmax']]=df_imgx[['ymin','xmin','ymax','xmax']]/102
6
    # Es necesario pasarlos a un array numpy
    gt_bb=np.squeeze(df_imgx[['ymin','xmin','ymax','xmax']].values)
    df_imgx['labels']=0
    gt_labels=np.squeeze(df_imgx['labels'].values)
    # Añadimos los valores a la clase como los reales
    evaluate.add_single_ground_truth_image_info(p,gt_bb,gt_labels)
    # Leemos el valor de los bounding boxes de la lista de diccionarios
    bb=np.squeeze(d[i]['bb'][0])
    scores=np.squeeze(d[i]['scores'][0]) # ademas de los resultados

    labels=np.zeros(scores.shape)
    evaluate.add_single_detected_image_info(p,bb,scores,labels)
    i+=1 # aumentamos en uno el iterador

res=evaluate.evaluate() # se calculan los resultados y se muestran por pantalla

print('mAP: ',res[0])
print('Precisiones por clase: ',res[1])
print('Recalls por clase',sum(res[2][0])/len(res[2][0]))
```