E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

Detección de objetos en carretera: comparativa entre técnicas de Machine Learning y Deep Learning

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor: Erick Martínez Fernández

Director: Mikel Sesma Sara

Pamplona, 09-2022

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

"Any sufficiently advanced technology is indistinguishable from magic"

- Arthur C. Clarke

Resumen

En este trabajo se va a abordar el problema de detección de objetos en carretera utilizando dos tipos de técnicas. Por un lado, utilizaremos técnicas de procesamiento de imagen para la extracción de características junto con distintos modelos de *Machine Learning* (Redes Neuronales, *Random Forest*, etc). Por otro, estudiaremos técnicas propias de *Deep Learning* basadas en redes neuronales convolucionales de distintos tipos (como YOLO, *You Only Look Once*, o SSD, *Single Shot Detector*). Compararemos los resultados obtenidos en ambas metodologías utilizando un *dataset* con 100.000 imágenes.

Palabras clave

Machine Learning, *Deep Learning*, Inteligencia Artificial, Visión Artificial, Red Neuronal, Red Neuronal Convolucional (CNN), YOLO (*You Only Look Once*), SSD (*Single Shot Detector*)

Abstract

In this project we are going to approach the problem of road object detection using two types of techniques. On the one hand, we will use image processing techniques for feature extraction with different models of Machine Learning (Neural Networks, Random Forest, etc). On the other hand, we will study Deep Learning techniques based on convolutional neural networks of different types (such as YOLO (You Only Look Once), or SSD (Single Shot Detector). We will compare the results obtained in both methodologies using a 100,000 images dataset.

Key Words

Machine Learning, Deep Learning, Artificial Intelligence, Computer vision, Neural Network, Convolutional Neural Network, YOLO (You Only Look Once), SSD (Single Shot Detector)

Índice

1. Introducción **8**

2. Preliminares **10**
2.1. *Machine Learning* . . . 10
2.1.1. Perceptrón Multicapa . . . 10
2.1.1.1. Proceso de aprendizaje . . . 13
2.1.2. *Random Forest* . . . 14
2.1.2.1. Ganancia de información . . . 16
2.1.2.2. Indice Gini . . . 16
2.1.2.3. Algoritmo de aprendizaje de un *Random Forest* . . . 17
2.1.3. *Ada Boost* . . . 17
2.1.4. Evaluación de clasificación de imágenes: *f1-score* . . . 18
2.2. Extracción de características: HOG (*Histogram Of Oriented Gradients*) . . 19
2.3. *Non-Max Suppression* (Supresión de no máximos) . . . 23
2.3.1. *Intersection Over Union* (IOU) . . . 23
2.3.2. Algoritmo *Non-Max Suppression* . . . 24
2.4. *Deep Learning* . . . 24
2.4.1. CNN (*Convolutional Neural Network*) . . . 24
2.4.1.1. Convoluciones . . . 25
2.4.1.2. *Padding* . . . 25
2.4.1.3. *Pooling* . . . 26
2.4.1.4. Capas totalmente conectadas (*fully connected layers*) . . . 27
2.4.1.5. Función de activación de la última capa (*softmax layer*) . . 28
2.5. Evaluación de detección de objetos: mAP (*Mean Average Precision*) 28

3. Algoritmos de detección de objetos basados en *Deep Learning* **30**
3.1. YOLO (*You Only Look Once*) . . . 30
3.1.0.1. Versiones de YOLO . . . 30
3.1.0.2. Codificación de las *Bounding Boxes* . . . 31
3.2. SSD (*Single Shot Detector*) . . . 34
3.2.1. Arquitectura de SSD . . . 34
3.2.1.1. VGG-16 . . . 34
3.2.1.2. Capas propias de SSD . . . 35
3.2.1.3. *Non-Maximum Suppression* . . . 35
3.2.2. Funcionamiento de SSD . . . 36
3.2.2.1. Extracción de características (*Feature Extraction*) 36
3.2.2.2. Fase de detección (*Detection Head*) . . . 36

4. Implementación de los modelos **39**
4.1. *Dataset* . . . 39
4.2. *Machine Learning* . . . 40
4.2.1. Perceptrón Multicapa . . . 41
4.2.2. *Random Forest* . . . 42
4.2.3. *Ada Boost* . . . 42
4.2.4. Métrica de error para la clasificación . . . 43
4.3. *Deep Learning* . . . 44
4.4. YOLOv5 . . . 44
4.4.0.1. Formato del *dataset* . . . 44

4.4.0.2. Entrenamiento de la red 44
4.4.0.3. *Testing* de la red 45

5. Resultados 46
5.1. Maching Learning 46
5.1.1. Un modelo por cada *Bounding Box* 46
5.1.1.1. Clasificación 46
5.1.1.2. Detección 47
5.1.2. Un modelo por cada clase 48
5.1.2.1. Clasificación 48
5.1.2.2. Detección 50
5.2. Deep Learning 53
5.2.1. YOLO 53
5.2.1.1. Casos específicos 54

6. Conclusiones 61

7. Lineas futuras 61

Referencias 62

Índice de figuras

1. Ejemplo de perceptrón multicapa 10
2. Ejemplo de Perceptrón 11
3. Función de activación ReLU 11
4. Función de activación Escalonada 12
5. Función de activación Tanh 12
6. Separabilidad del perceptrón 13
7. Ejemplo de perceptrón multicapa para un problema de 3 clases 13
8. Estructura de árbol de decisión 15
9. Ejemplo de árbol de decisión 15
10. Ejemplo de matriz de confusión 19
11. Ejemplo de gradiente horizontal 20
12. Ejemplo de gradiente vertical 20
13. Ejemplo de celdas para el HOG 21
14. Ejemplo de codificación del histograma de HOG. Este ejemplo ha sido creado con 9 *bins* y gradientes “sin signo”. En lo que respecta al color verde su dirección es 10º, como en el *array* del histograma no está representado el 10º como tal (está en 0º y el 20º) se repartirá la magnitud proporcionalmente entre cada uno de los dos lados, en este caso 10º está justo en el medio entre 0º y 20º, por tanto la magnitud 4 se divide entre 2 agregando un 2 a cada uno de los ángulos. En el caso del ejemplo en rojo existe un 80º dentro del *array* del histograma por tanto se agrega directamente el 2 (magnitud correspondiente a la dirección 80º) a la celda del histograma. Si el ángulo supera los 160º es similar a lo dicho anteriormente, la magnitud se reparte proporcionalmente entre ambos extremos (0º y 160º en este caso). Al finalizar con todos los píxeles de la celda se agregan todas las magnitudes que hay para cada ángulo para así formar el histograma (Figura 15) 22
15. Ejemplo de histograma para HOG 22

16. Ejemplo de HOG aplicado a una imagen de Señal de Stop 23
17. Descripción gráfica de la operación que se realiza en el cálculo del IoU . . . 23
18. Ejemplo de Supresión de No Máximos 24
19. Convolución con filtro de 1 Dimensión 25
20. Ejemplo de *Padding* 26
21. Ejemplo de tipos de *Padding* 26
22. Ejemplo de funcionamiento de *Pooling layers*. En el caso de la inzquierda tenemos un tamaño de filtro o ventana de 2 y un *stride* o paso de 2, empezando por arriba a la izquierda $max\{1, 3, 2, 9\} = 9$, desplazamos dos posiciones y $max\{2, 1, 1, 1\} = 2$ y así con toda la matriz de entrada. En el caso de la derecha tenemos $tamaño\ de\ filtro = 3$ y $stride = 1$, comenzando arriba a la izquierda con el recuadro verde tenemos $max\{1, 3, 2, 2, 9, 1, 1, 3, 2\} = 9$, con una ventana de 3x3 y un *stride* de 1 vamos avanzando, en el caso del recuadro morado $max\{9, 1, 1, 3, 2, 3, 3, 5, 1\} = 9$ y similar con el recuadro rojo inferior $max\{1, 3, 2, 8, 3, 5, 5, 6, 1\} = 8$ 27
23. Arquitectura de CNN 28
24. Prerequisitos para codificación YOLO 31
25. Explicación parámetros de codificación de YOLO 32
26. Ejemplo de codificación YOLO sin objeto 32
27. Ejemplo de codificación YOLO con más de 1 objeto 33
28. Arquitectura de la primera versión de YOLO 33
29. Arquitectura de VGG-16 35
30. Arquitectura de *Single Shot Detector* 36
31. Diferentes tipos de *Anchor Boxes* 37
32. *Anchor Boxes* centradas en un punto 37
33. Ventana deslizante con *Anchor Boxes* en SSD 37
34. Codificación de la detección en SSD 38
35. Arquitectura de *Single Shot Detector* por partes 38
36. Ejemplo de diversos *datasets* 39
37. Ejemplo de formato de datos para entrenamiento de YOLOv5 44
38. Ejemplo de falsos positivos en las detecciones del modelo con diferentes *Bounding Boxes* 47
39. Otro ejemplo de falsos positivos en las detecciones del modelo con diferentes *Bounding Boxes* 47
40. Otro ejemplo de falsos positivos en las detecciones del modelo con diferentes *Bounding Boxes* 48
41. Ejemplo de coincidencia parcial de señal de tráfico a la izquierda de la imagen 50
42. Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen 51
43. Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen 51
44. Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen 51
45. Ejemplo de coincidencia parcial de señal de tráfico a la izquierda de la imagen 51
46. Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen 52
47. Ejemplo de coincidencia parcial de señal de tráfico a la izquierda de la imagen 52
48. Ejemplo de falsos positivos en las detecciones 52
49. Ejemplo de resultados generales de YOLOv5 53
50. Otro ejemplo de resultados generales de YOLOv5 53
51. Ejemplo de resultados de noche de YOLOv5 54
52. Ejemplo de resultados con cielo nublado de YOLOv5 54
53. Ejemplo de resultados con el cielo despejado de YOLOv5 54

54. Ejemplo de no detección de señales de tráfico pese a su semejanza (carteles de la derecha) de YOLOv5 . . . 55
55. Ejemplo de detección de cartel publicitario como señal de tráfico de YOLOv5 55
56. Ejemplo de detección de señal de tráfico pese a la poca visibilidad (señales de la izquierda) de YOLOv5 . . . 55
57. Ejemplo de detección de dos señales (un falso positivo) y dos coches pese a la borrosidad de la imagen con YOLOv5 . . . 56
58. Ejemplo de detección de un coche y una señal de tráfico (falso positivo) pese a la borrosidad y ángulo de la imagen con YOLOv5 . . . 56
59. Ejemplo de detección de dos señales y dos coches (uno es un falso positivo) pese a la borrosidad de la imagen con YOLOv5 . . . 56
60. Ejemplo de detección erronea de un coche (parte derecha de la imagen) con YOLOv5 . . . 57
61. Ejemplo de detección erronea de un coche en un cristal mojado con YOLOv5 57
62. Ejemplo de detección erronea de dos personas en una pegatina de la luna con YOLOv5 . . . 57
63. Ejemplo de detección precisa multiclase con YOLOv5 . . . 58
64. Ejemplo de detección precisa multiclase (con bicicleta) con YOLOv5 . . . 58
65. Ejemplo de detección precisa multiclase con YOLOv5 . . . 58
66. Matriz de confusión del modelo de YOLOv5 . . . 59
67. Curva de *Precision-Recall* de YOLOv5 . . . 60

Índice de tablas

1. *F1-score macro* para *Bounding Box* Cuadradas . . . 46
2. *F1-score macro* para *Bounding Box* Rectangulares horizontales . . . 46
3. *F1-score macro* para *Bounding Box* Rectangulares verticales . . . 47
4. *F1-score macro* para modelo binario de *Traffic sign* . . . 48
5. *F1-score macro* para modelo binario de *Motor* . . . 49
6. *F1-score macro* para modelo binario de *Bike* . . . 49
7. *F1-score macro* para modelo binario de *Car* . . . 49
8. *F1-score macro* para modelo binario de *Person* . . . 50
9. *F1-score macro* para modelo binario de *Rider* . . . 50

1. Introducción

Las redes neuronales y la inteligencia artificial afectan cada vez más al día a día de nuestra sociedad, desde conducción autónoma de coches, predicción de precios dentro del sector inmobiliario, generación de imágenes y hasta filtros de instagram a través de visión artificial. En este último punto es en el que se va a centrar este proyecto, la visión artificial es la capacidad que tienen las maquinas de entender la visión humana del mundo a través de imágenes.
Concretamente el problema al que nos enfrentamos es la detección de objetos en carretera, comparando entre métodos que utilizan modelos de *Machine Learning* y métodos que utilizan modelos de *Deep Learning*. Específicamente para la detección de coches, bicicletas, motos, personas, señales de tráfico y *riders* (Conductores de bicicleta y de moto) en imágenes de carretera.

La importancia de este problema recae en que si el resultado de un modelo es fiable y eficiente en cuanto a detección de objetos y rapidez, este podría ser utilizado en un coche autónomo detectando en tiempo real todos los objetos que se encuentran a su alrededor con precisión (e.g este sistema está siendo aplicado actualmente en las pruebas de conducción autónoma de la marca de coches Tesla, Inc mediante una serie de cámaras y sensores alrededor del coche).

En cuanto a la parte de *Machine Learning* realizamos un estudio teniendo en cuenta diferentes características de los diferentes modelos, número de estimadores base y tipo de estimador para *AdaBoost*, el *solver* para la optimización de los pesos, el parámetro alfa y los tamaños de las capas para el Perceptrón Multicapa (*MLP*) además de número de estimadores (número de árboles en el bosque), el criterio de calidad, la profundidad máxima del árbol y el número máximo de características a considerar para el *Random Forest*.

En lo que a la parte de *Deep Learning* se refiere estudiamos dos principales métodos, en primer lugar YOLO (*You Only Look Once*), desarrollado por Joseph Redmon, Santosh Divvala, Ross Girshick y Ali Farhadi en 2015 [1]. Consiste en una única Red Neuronal profunda que predice *Bounding Boxes* y sus confianzas por cada clase desde las imágenes completas en un solo ciclo, gracias a que el proceso entero de detección se base únicamente en una sola red este se puede optimizar fácilmente de principio a fin basándose en el rendimiento de la propia detección.

El modelo básico de YOLO llega a procesar imágenes en tiempo real a 45 fotogramas por segundo, en cuanto al modelo de *Fast* YOLO procesa 155 fotogramas por segundo.

El segundo modelo que vamos a estudiar en la parte de *Deep Learning* es el SSD (*Single Shot Detector*), desarrollado por Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu y Alexander C. Berg en 2016 [2]. Tiene ciertas similitudes con el modelo de YOLO, consiste también en una única Red Neuronal profunda aplicada a una imagen completa una única vez (*Single Shot*) para crear así un mapa de características, después este se procesa una segunda vez para la obtención de las *Bounding Boxes* y de las confianzas para cada clase. El modelo de SSD300 llega a procesar imágenes a 59FPS y el modelo de SSD500 a 22FPS.

Con todo esto se realiza una comparación final entre resultados de ambos métodos tanto *Machine Learning* como *Deep Learning* a nivel teórico mediante los porcentajes de acierto y a nivel práctico los modelos aplicados a imágenes reales.

Dicho todo esto el objetivo principal de este trabajo es:

- Comparar entre una metodología basada en *Machine Learning* y otra en *Deep Learning* para detectar objetos en carretera

Para ello, planteamos los siguientes objetivos específicos:

- Evaluación de los diferentes métodos de *Machine Learning* ya citados anteriormente (*MLPClassifier* (*Multi-layer Perceptron Classifier*), *AdaBoost* y *RandomForest*) utilizando un modelo para cada tipo de *Bounding Box* (Cuadrada, Rectangular vertical y rectangular horizontal) modificando de cada uno de ellos los diferentes parámetros para que de esa forma se pueda obtener la combinación ideal para nuestro conjunto de datos.

- Evaluación de los diferentes métodos de *Machine Learning* utilizando un modelo para cada clase (coches, bicicletas, motos, personas, señales de tráfico y *riders* (Conductores de bicicleta y de moto)) modificando de cada uno de ellos los diferentes parámetros para que de esa forma se pueda obtener la combinación ideal para nuestro conjunto de datos.

- Estudio del método YOLO (You Only Look Once).

- Estudio del método SSD (Single Shot Detector).

- Evaluación del método YOLO (*You Only Look Once* (*Deep Learning*)) en nuestro *dataset*.

- Comparación entre los resultados de los modelos de *Machine Learning* (Modelos para cada *Bounding Box* y modelos para cada clase) y el modelo de *Deep Learning* (YOLO)

- Aplicación del mejor modelo a vídeos en carretera.

2. Preliminares

2.1. *Machine Learning*

Según Arthur Samuel (1959, IBM) “El área de estudio que da a los ordenadores la capacidad de aprender sin ser programados explícitamente” [3].
Según Robert E. Schapire (1990) “Aprender a hacer las cosas mejor en el futuro en base a las experiencias del pasado” [4].
El *Machine Learning* se puede definir de diversas formas, pero como idea general sería la capacidad que tiene un ordenador de aprender y crecer en función de cosas que ocurrieron en el pasado sin ser este explícitamente programado para ello. Es decir, se basan en los datos obtenidos para poder predecir nuevos datos. Esto a día de hoy se aplica a una gran variedad de campos, desde la sección de *spam* del email, los anuncios sugeridos en páginas en función de tus búsquedas recientes, hasta reconocimiento de imágenes a través de visión artificial.

2.1.1. Perceptrón Multicapa

Las redes neuronales originalmente se desarrollaron como una imitación al funcionamiento de las neuronas de un cerebro humano. Formalmente se trata de un conjunto de neuronas interconectadas entre sí mediante enlaces con pesos asignados, de forma que la salida de una neurona es la entrada de la siguiente neurona (Figura 1).

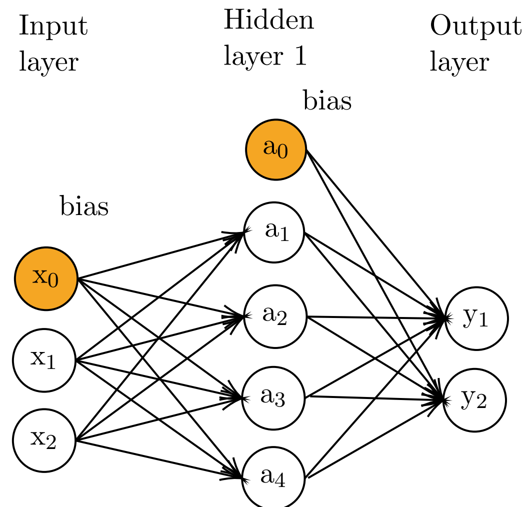


Figura 1: Ejemplo de perceptrón multicapa

Pero antes de introducirnos en los tecnicismos del perceptrón multicapa es necesario introducir y explicar el Perceptrón (Red neuronal con una única capa). Fue planteado y desarrollado por el psicólogo Frank Ronsenblant en 1958 [5] basado en una regla de aprendizaje a través del error. Este consiste en un conjunto de neuronas de entrada y en una única neurona de salida interconectadas entre ellas (Figura 2).

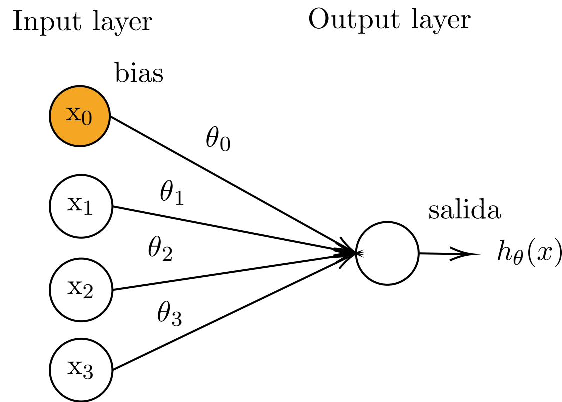


Figura 2: Ejemplo de Perceptrón

En este caso para el cálculo de la salida $h_\theta(x)$ necesitamos hacer una suma ponderada entre los pesos de cada enlace con la siguiente neurona y el valor de entrada (x):

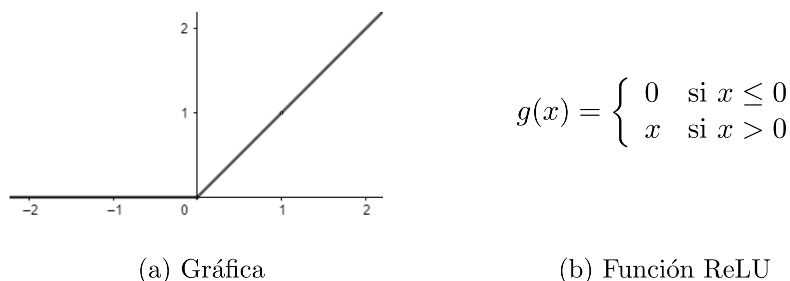
$$\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3. \tag{1}$$

Además a esto hay que añadirle que, para formar pequeños cambios no lineales en las salidas de las neuronas necesitamos una función de activación. Teniendo en cuenta que el valor de la entrada $x_0 = 1$ la fórmula nos quedaría:

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3), \tag{2}$$

siendo θ_0 el sesgo (bias) y g(z) la función de activación. Podemos elegir entre diversas funciones de activación según nos convenga, desde la Relu(*Rectified Linear Unit*), escalonada, tanh etc.

La función ReLU (*Rectified Linear Unit*) se caracteriza por tomar el valor 0 cuando la entrada es igual o menor que 0 y tomar el valor de entrada cuando esta es mayor que 0 (Figura 3)

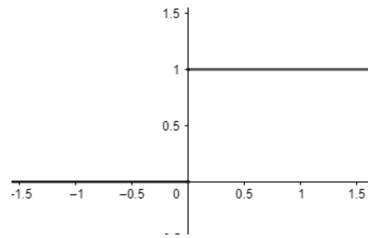


(a) Gráfica

(b) Función ReLU

Figura 3: Función de activación ReLU

La función escalonada se caracteriza porque cuando los valores de la x son menores e iguales que 0, la y = 0 y cuando son mayores que 0 y = 1



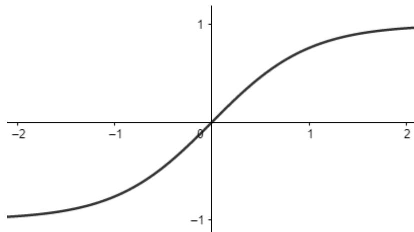
(a) Gráfica

$$g(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ 1 & \text{si } x > 0 \end{cases}$$

(b) Función escalonada

Figura 4: Función de activación Escalonada

En el caso de la función de tangente hiperbólica esta tiene dos asíntotas, en el 1 y en el -1, es decir que cuando x tiende a ∞ el valor tiende a 1 y cuando x tiende a $-\infty$ el valor tiende a -1 (Función simétrica respecto del punto de origen).



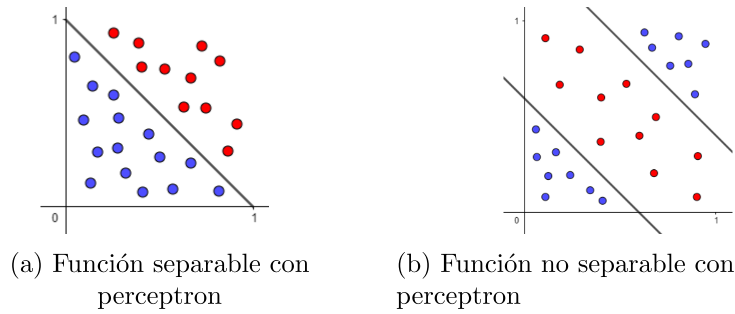
(a) Gráfica

$$g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

(b) Función Tanh

Figura 5: Función de activación Tanh

Sin embargo con un perceptrón únicamente pueden ser resueltos problemas con una frontera lineal, en cuanto el problema de clasificación requiere una frontera no lineal para ser solucionado es imposible ser resuelto con un solo perceptrón (Siempre y cuando la entrada del perceptrón sean las variables originales sin sufrir ningún tipo de transformación polinómica o de otro tipo), por ejemplo un perceptrón puede solucionar problemas de tipo puerta AND pero no puede solucionar una puerta XOR, ya que esta necesita al menos dos rectas para poder realizar bien la clasificación (Figura 6), teniendo en cuenta esto para solucionar problemas más complejos que una puerta AND es necesario pasar de un único perceptrón a un perceptrón multicapa (Añadiendo capas y neuronas al perceptrón), si el perceptrón multicapa tiene únicamente una salida podrá solo clasificar una clase (0 para negativo y 1 para positivo), sin embargo si este tiene más de una salida podrá realizar una clasificación multiclase, de forma que se puede ver en la Figura 7 que la salida de la red es en forma de array [0, 0, 1], referenciando cada uno de los números a las clases [Coche, Persona, Señal] tenemos que la salida de la red es un 1 para la clase Señal.



(a) Función separable con perceptron

(b) Función no separable con perceptron

Figura 6: Separabilidad del perceptrón

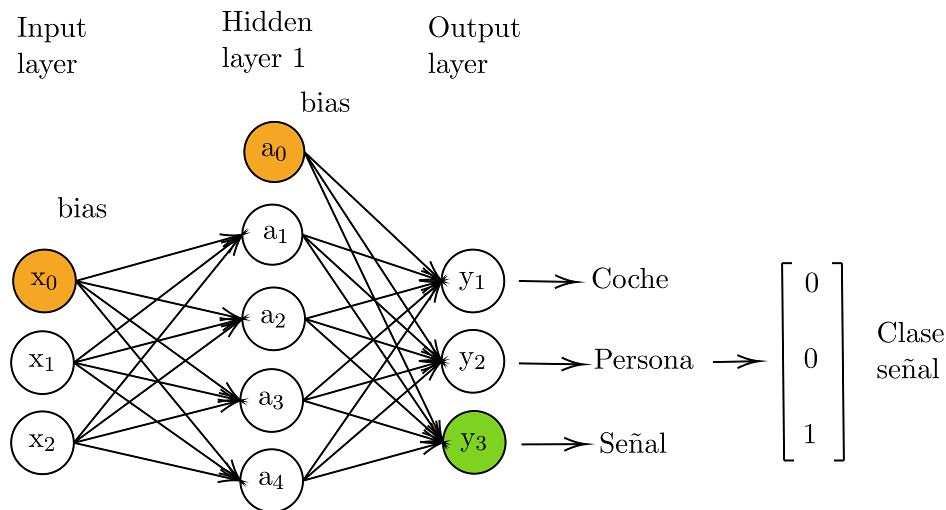


Figura 7: Ejemplo de perceptrón multicapa para un problema de 3 clases

Teniendo ya el perceptron multicapa explicado ahora nos vamos a centrar en cómo este se adapta al conjunto de entrada, cómo “aprende”de los datos para obtener buenas soluciones.

2.1.1.1. Proceso de aprendizaje

Primero de todo se inicializan los pesos de los enlaces con valores aleatorios, tras esto es necesario seleccionar un ejemplo de entrenamiento x^n compuesto por x_0, x_1, x_2,..., x_n y pasarlo por la red hasta obtener un resultado y^n, teniendo entonces los valores de entrada de la red y los valores de salida se puede calcular el coste para así comenzar a entrenar la red. Mediante este entrenamiento se buscará minimizar la función de coste de la red, siendo $E(h_\theta(x), y)$ la función de coste de la salida de la red $h_\theta(x)$ respecto de la clasificación del ejemplo real y.

Es necesario hacer el entrenamiento desde la última capa hacia la primera capa (*back-propagation*), ya que de esta forma se puede ver cómo afectan los pesos de los enlaces anteriores en la siguiente capa y así ir modificándolos para disminuir el error de la red [6]. Utilizando el descenso del gradiente para la minimización de costes en la red y a través de la regla de delta generalizada llegamos a

$$\theta^{n+1} = \theta^n - \alpha \frac{\partial E^n}{\partial \theta} \tag{3}$$

Tras realizar las derivaciones llegamos a la actualización de pesos de la última capa y del resto de capas (Ecuación 4, Ecuación 5).

$$\theta_{ji}^{C-1,n} = \theta_{ji}^{C-1,n-1} + \alpha\delta_i^{C,n} a_j^{C-1,n} \tag{4}$$

$$\theta_{kj}^{c} n + 1 = \theta_{kj}^{c,n} + \alpha\delta_j^{c+1} a_k^{c,n} \tag{5}$$

Siendo $\theta_{ji}^{C-1,n}$ el peso del enlace entre la neurona j y la neurona i para los enlaces entre la capa $C-1$ y C (entre la penúltima e última capa) y para la iteración n. Siendo α la constante de proporcionalidad entre el cambio del peso y el error cometido, de forma que si α es grande eso ayudara a su convergencia rápida ya que permite avanzar más rápidamente en la superficie del error, sin embargo puede existir el caso de que el método oscile alrededor del mínimo y nunca llegue a converger. En el caso de un α bajo la convergencia es más lenta pues los desplazamientos en la superficie del error son menores, sin embargo se asegura más su convergencia. En el caso de δ (delta generalizado) depende de la función de activación utilizada. Y por último la a es la activación de la capa (salida de la capa anterior). El parámetro δ_j^c es la derivada de la función de activación utilizada (Ecuación 6)

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \tag{6}$$

Este proceso de actualización de pesos se repite para cada uno de los ejemplos hasta finalizar con todos, y esto un número de iteraciones determinado o hasta que el error obtenido en la salida sea menor que un umbral dado.

2.1.2. *Random Forest*

Los *Random Forest* o bosques aleatorios son un tipo de ensemble, un ensemble es un clasificador o regresor (en este caso clasificador) formado por muchos clasificadores llamados clasificadores base. El principio esencial de estos ensembles es el de unir un grupo de modelos considerados débiles para formar uno fuerte y robusto. En el caso de los ensembles la diversidad entre los distintos clasificadores base es fundamental para garantizar un correcto funcionamiento. Los clasificadores base de un *Random Forest* son los árboles de decisión o *Decision Tree*. Para conseguir esta diversidad de la que hablábamos en los árboles de decisión es muy importante que cada uno de los árboles se entrene con un conjunto de instancias de entrada diferente, de forma que tras agregar todos los árboles para formar el *Random forest* el modelo generalice lo máximo posible y evite a toda costa el sobre entrenamiento, muy común en este tipo de ensembles, a esta selección aleatoria de instancias para el entrenamiento (”bolsas” de instancias) de cada uno de los clasificadores base se le llama Bootstrap.

Los árboles de decisión han sido utilizados en campos desde el de la inteligencia artificial como regresor o clasificador hasta en valoración de inversiones dentro del campo de la economía.

Se le denomina árbol de decisión por su semejanza con un árbol con muchas ramas. Este consiste en una estructura de diagrama de flujo formada por nodos (circulos) y ramas (flechas), dentro de los nodos existen el nodo raiz, los nodos internos (no hoja) y los nodos hoja (Figura 8).

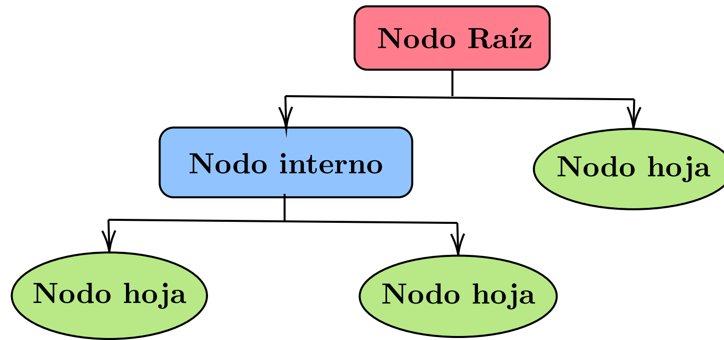


Figura 8: Estructura de árbol de decisión

En cada uno de los nodos se separan los datos en función de uno de los atributos, por ejemplo si una persona juega al tenis (atributo booleano si o no, 1 o 0), si es 1 todos los ejemplos que cumplan con esto (de los ejemplos que quedan en ese nodo) se irán hacia una rama del árbol (por ejemplo izquierda) y todos los ejemplos que no cumplan esto, es decir personas que no jueguen a tenis (de los ejemplos que quedan en ese nodo) irán por el otro lado (por la derecha). La idea es similar si el atributo es de tipo numérico, pero en ese caso se determinará un umbral para la separación (Atributo edad) (Figura 9).
En resumen se aplica la técnica divide y vencerás recursivamente hasta que en los nodos hoja se queden los datos separados por clases.

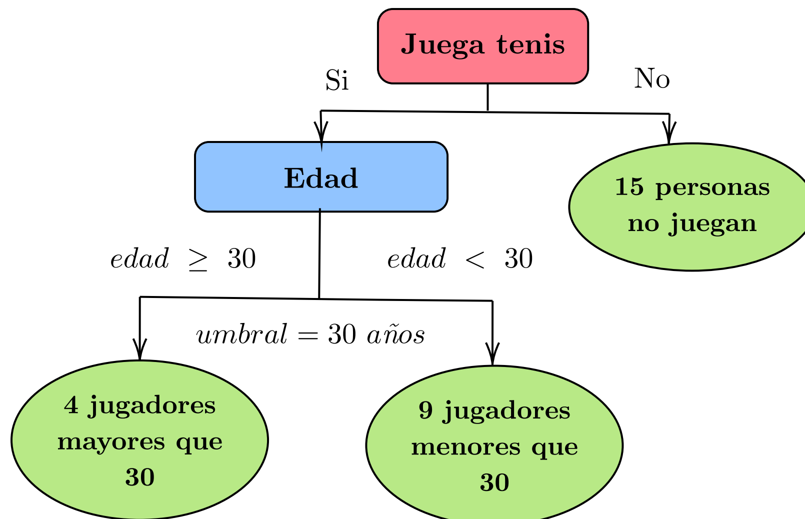


Figura 9: Ejemplo de árbol de decisión

Los algoritmos más comunes utilizados en árboles de decisión son CART (Índice Gini), ID3 (Ganancia de información), C4.5 (sucesor del ID3, Ratio de ganancia) y CHAID (Chi cuadrado), sin embargo aquí solo nos vamos a centrar en los métodos heuristicos utilizados en nuestro modelo, es decir en la ganancia de la información (*entropy*) y en el índice gini.

2.1.2.1. Ganancia de información

La ganancia de información es utilizada por ID3, para introducir la ganancia de información es necesario introducir primero la entropía. La entropía es la medida de impureza de un nodo de forma que, si la entropía es baja el grado de impureza es bajo y los ejemplos están muy separados en base a las clases, sin embargo si la entropía es alta la impureza será alta y no habrá separación de los ejemplos en base a las clases. Se define la entropía para un nodo t como:

$$E(t) = \sum_{j=1}^{Clases} -p(j|t)log_2(p(j|t)) \tag{7}$$

Teniendo en cuenta esto, la ganancia de información es la cantidad de información que se gana (entropía que se pierde) si se divide por el atributo A con respecto a predecir la clase C (Si los atributos son continuos habrá que discretizar previamente) Para un problema de clasificación de varias clases, la entropía del atributo a predecir C en un nodo T se calcula:

$$E(C) = \sum_{j=1}^{Clases} -\frac{n_j}{n}log_2\frac{n_j}{n} \tag{8}$$

siendo n el número de ejemplos y n_j el número de ejemplos de la clase j en el nodo T. La entropía media para un atributo A seleccionado en sus nodos hijos se calcula como:

$$E(A) = \sum_{j=1}^{Particiones} \frac{n_i}{n}E(nodo_i) \tag{9}$$

siendo n el número de ejemplos del nodo T (nodo dividido) y n_i el número de ejemplos del nodo hijo T_i.

Teniendo en cuenta lo anterior la ganancia de información se calcula como:

$$Gain(A) = E(C) - E(A) \tag{10}$$

Será elegido el atributo que maximice la ganancia de información.

2.1.2.2. Indice Gini

El indice Gini es el utilizado por CART (Classification and Regression Trees), considerada una de las 10 mejores técnicas de minería de datos. Para un nodo t el índice Gini es

$$Gini(t) = 1 - \sum_{j=1}^{C} p(j|t)^2 \tag{11}$$

siendo $p(j|t)^2$ la frecuencia relativa de la clase j en el nodo t. Cuando $Gini(t) = 0$ tenemos que todos los ejemplos pertenecen a una clase por tanto es una información valiosa ya que la clasificación es perfecta, en caso contrario los ejemplos estarán distribuidos homogéneamente por las clases.

2.1.2.3. Algoritmo de aprendizaje de un *Random Forest*

Como ya hemos dicho anteriormente un *Random Forest* es un ensemble formado por árboles de decisión. Para el aprendizaje del *Random Forest* es necesario:

1. Seleccionar el número de árboles
2. Para cada árbol:
   - Seleccionar aleatoriamente n ejemplos con reemplazamiento, siendo n $<$ Número de ejemplos totales (bootstrap)
   - Aprender el árbol con esa muestra (Eligiendo aleatoriamente m atributos en cada nodo, siendo m $\ll$ número de atributos del problema)
3. No se poda el árbol generado

Para la clasificación de un ejemplo nuevo hay que clasificar el ejemplo con todos los árboles del *Random Forest* y cada uno de estos vota por la clase que ha predicho, de forma que la clase más votada gana. Para la estimación del error en el *Random Forest* es necesario clasificar cada uno de los ejemplos con los clasificadores base que no hayan sido construidos con ese ejemplo.

2.1.3. *Ada Boost*

AdaBoost (***Ada****ptative* ***Boost****ing*) es, de nuevo, un tipo de ensemble (un clasificador formado por muchos clasificadores llamados clasificadores base), el cual fue originalmente creado para mejorar la eficiencia de clasificadores binarios. Intenta transformar clasificadores débiles (clasificador que funciona mejor que uno aleatorio, pero aun así se desempeña mal) en clasificadores fuertes asignando a las instancias más difíciles (instancias que han sido clasificadas mal por el clasificador anterior) un peso mayor, con esto se consigue reducir el *bias* y la varianza además de aumentar el margen de separación entre clases, sin embargo el problema de esto es que las instancias del *dataset* con ruido o los *outliers* van a adquirir un peso mayor [7].

El algoritmo de *Ada Boost* funciona de la siguiente forma:

- En primer lugar se crea el primer clasificador débil (e.g Árbol de decisión) y se le asigna el mismo peso a todas las muestras (ejemplos), ya que inicialmente no se sabe cuales deberían de tener más importancia:

$$w = \frac{1}{N} \tag{12}$$

  siendo N el número total de ejemplos.

- Repetir esto T veces (siendo T el número de clasificadores) o hasta que el error sea menor que un umbral:
  - Se clasifican los ejemplos con el clasificador débil
  - Se calcula la importancia o la influencia (peso) del clasificador al clasificar los ejemplos:

$$\alpha_t = \frac{1}{2} ln \frac{(1 - TotalError)}{TotalError} \tag{13}$$

    siendo α la influencia de este clasificador en la clasificación final y el $TotalError$ el número de clasificaciones erroneas entre el tamaño del conjunto de entrenamiento total
  - Se modifican los pesos de los ejemplos, en el caso de que el ejemplo esté correctamente clasificado se decrementa su peso y en el caso contrario (el ejemplo está mal clasificado) se aumenta el peso del mismo:

$$w_i = w_{i-1} \cdot e^{\pm\alpha} \tag{14}$$

    en el caso en el que α tenga simbolo negativo, es decir $e^{-\alpha}$ significa que se ha clasificado correctamente el ejemplo y por lo tanto hay que disminuir el peso w_i, sin embargo si α tiene signo positivo, es decir e^{α} significa que el ejemplo ha sido clasificado incorrectamente y que hay que aumentar el peso.
- Una vez finalizado el bucle hay que agregar todos los clasificadores débiles obtenidos junto con sus pesos para generar el clasificador fuerte que usaremos como modelo de la siguiente forma:

$$H_{final} = sgn(\sum_{t=0}^{T-1} \alpha_t g_t) \tag{15}$$

  siendo α_t las importancias o influencias de cada clasificador y g_t cada clasificador en sí, es decir se hace una suma ponderada de los clasificadores y las importancias que da como resultado el clasificador final [8].

En cuanto al error del clasificador ϵ_t es la suma de los pesos de todos los ejemplos mal clasificados. En nuestro caso el clasificador débil utilizado es el *Decision Tree* que ya ha sido explicado previamente.

2.1.4. Evaluación de clasificación de imágenes: *f1-score*

Para la evaluación de la clasificación de imágenes con los 3 modelos que se acaban de presentar hemos utilizado el F1-*score*. Antes de la explicación del *F1-score* es necesario introducir los siguientes conceptos teóricos:

1. Matriz de confusión: Permite la visualización del desempeño del modelo utilizado para la clasificación, esta necesita 4 atributos para su creación:
   - *True Positives* (TP): La etiqueta predicha por el modelo es positivo, similar que el valor real.
   - *True Negatives* (TN): La etiqueta predicha por el modelo es negativa, similar que el valor real.
   - *False Positives* (FP): La etiqueta predicha por el modelo es positiva, sin embargo el valor real es negativo.

- *False Negatives* (FN): La etiqueta predicha por el modelo es negativo, sin embargo el valor real es positivo.

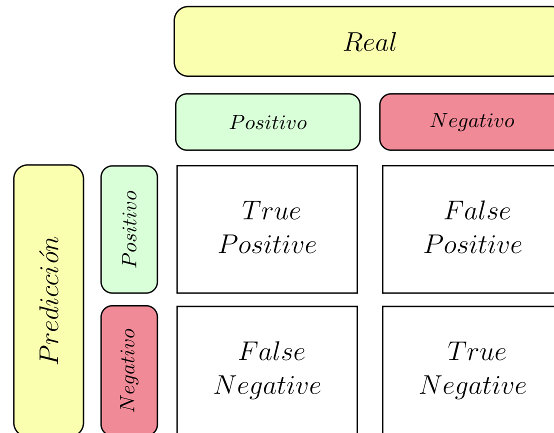


Figura 10: Ejemplo de matriz de confusión

2. *Recall*: Mide como de bien se predicen los verdaderos positivos o *True Positive* sobre el total de las predicciones:

$$Recall = \frac{TP}{TP + FN} \tag{16}$$

3. Precision: Mide como de bien se predicen los verdaderos positivos o *True Positive* sobre todas las predicciones positivas

$$Precision = \frac{TP}{TP + FP} \tag{17}$$

Conociendo ya las definiciones de *Precision* y *Recall*, *F1-score* se define como una media armónica de *precision* y *recall*, F1-*score* alcanza su mejor puntuación con 1 y su peor con 0. Es una métrica ideal ya que tiene en cuenta como de compensadas estén las diferentes clases al realizar la evaluación y puede proporcionar un dato más real a diferencia del *accuracy*. Se define como

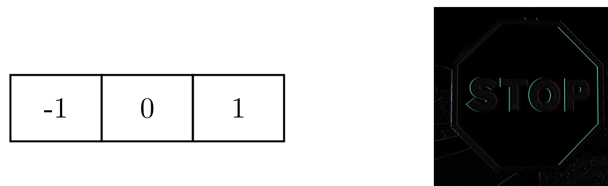
$$F1_score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{18}$$

2.2. Extracción de características: HOG (*Histogram Of Oriented Gradients*)

Hasta ahora todos los modelos que hemos visto de *Machine Learning* necesitaban un *array* o matriz de datos de entrada para realizar el entrenamiento, después con el entrenamiento ya hecho se le pasa un ejemplo para ser clasificado y así hallar el error. Sin embargo este proyecto está orientado al entrenamiento y la clasificación de imágenes, para ello necesitamos un método para transformar nuestros datos de entrada, en este caso imágenes a *arrays* o matrices con las que los modelos puedan operar. Para esta labor es utilizado el Histograma de Gradientes Orientados. El HOG es un descriptor de características muy utilizado en el mundo de la visión artificial para la detección de objetos, un descriptor de características es un modelo que consigue la representación de una imagen

lo más simplificada posible, de forma que se centre en las características útiles de la imagen y prescinda de las inútiles o superfluas, esto lo consigue mediante la codificación de una imagen de anchura x altura x 3 (considerando los canales RGB) en un *array* de n elementos (n dependiendo de las diferentes variables usadas durante la realización del HOG). El HOG fue planteado por primera vez (a pesar de que en aquel momento todavía no se llamaba HOG) por RK McConnell en 1986 [9], sin embargo el uso del HOG se generalizó en 2005 cuando fue utilizado para la detección de humanos por Navneet Dalal y Bill Triggs. El procesamiento de la imagen para obtener el vector de características funciona mediante los siguientes pasos [10]:

1. Preprocesamiento de la imagen: Redimensión de la imagen para poder utilizarla en el HOG
2. Calculo de gradientes: Es necesario realizar un cálculo de los gradientes de la imagen, tanto verticales como horizontales para resaltar de esta forma los diferentes tipos de bordes, esto se realiza filtrando toda la imagen con los núcleos de las Figuras 11a y 12a, dando como resultado las Figuras 11b y 12b



(a) Núcleos para el gradiente horizontal

(b) Señal de Stop con gradiente horizontal

Figura 11: Ejemplo de gradiente horizontal

(a) Núcleos para el gradiente vertical

(b) Señal de Stop con gradiente vertical

Figura 12: Ejemplo de gradiente vertical

3. Calculo de magnitud y dirección: Se calcula la magnitud

$$g = \sqrt{g_x^2 + g_y^2} \tag{19}$$

y la dirección

$$\theta = arctan\frac{g_y}{gx} \tag{20}$$

del gradiente para cada píxel de toda la imagen a partir de los gradientes horizontal (g_x) y vertical (g_y), tras esto nos quedan dos matrices de magnitudes y direcciones de tamaño ancho x alto (ancho y alto de la imagen). La magnitud y la dirección suelen ser representados mediante una flecha para cada píxel de la imagen, de forma

que la flecha muestra la dirección del degradado y su longitud muestra la magnitud, es decir la dirección de las flechas apunta a la dirección del cambio de intensidad y la magnitud muestra como de grande es la diferencia.

4. Separación en celdas: La imagen se separa en celdas, el tamaño de las celdas va en función del objeto que quieras detectar, ya que si el objeto es grande una celda muy pequeña no será suficientemente grande como para abarcar el objeto entero o una parte considerable de este y viceversa, en el caso de este ejemplo hemos elegido un tamaño de celda de 90 para que pudiese llegar a abarcar una letra entera o gran parte de esta además de segmentos grandes del borde de la señal (Figura 13).

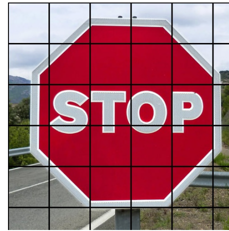


Figura 13: Ejemplo de celdas para el HOG

5. Creación de los histogramas: En este punto se codifican la magnitud y la dirección en un *array* para poder hacer el histograma para cada celda, cabe destacar que en este punto hay que elegir el número de *bins* de nuestro histograma, es decir, los elementos, o número de barras, del eje x del histograma, además de elegir si los gradientes van a tener “signo” o no, de forma que si el gradiente tiene “signo” van a ser representados ángulos desde 0º hasta 360º, pero si el gradiente va “sin signo” solo se representan ángulos entre 0º y 180º, por lo tanto las magnitudes de los ángulos que sobrepasen los 180º entraran dentro de su ángulo opuesto, por ejemplo si el ángulo es de 220º, la magnitud de ese ángulo se añade a la celda de 220º−180º= 40º. Teniendo ya definidos los *bins* y el signo para la codificación del HOG hay que separar esos 180º (en el caso de “sin signo”) o 360º (en el caso de “con signo”) en n *bins* e ir asignando las magnitudes de cada píxel a cada una de las celdas del histograma en función de su ángulo (Figura 14).

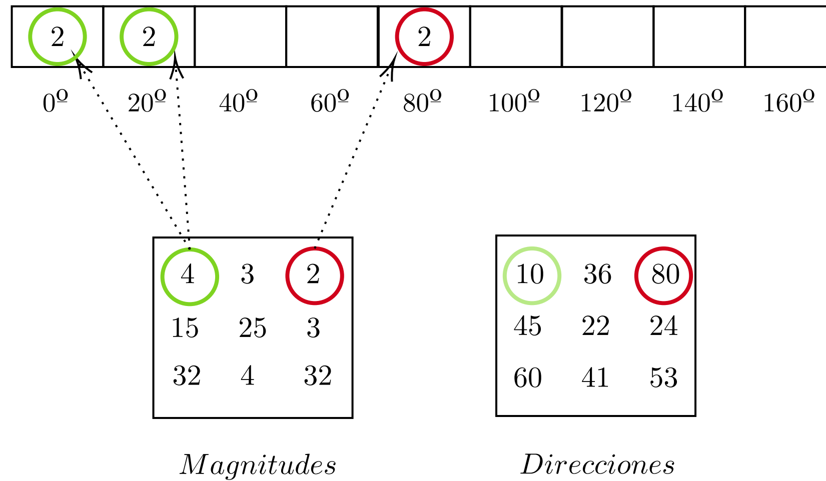


Figura 14: Ejemplo de codificación del histograma de HOG. Este ejemplo ha sido creado con 9 *bins* y gradientes "sin signo". En lo que respecta al color verde su dirección es 10º, como en el *array* del histograma no está representado el 10º como tal (está en 0º y el 20º) se repartirá la magnitud proporcionalmente entre cada uno de los dos lados, en este caso 10º está justo en el medio entre 0º y 20º, por tanto la magnitud 4 se divide entre 2 agregando un 2 a cada uno de los ángulos. En el caso del ejemplo en rojo existe un 80º dentro del *array* del histograma por tanto se agrega directamente el 2 (magnitud correspondiente a la dirección 80º) a la celda del histograma. Si el ángulo supera los 160º es similar a lo dicho anteriormente, la magnitud se reparte proporcionalmente entre ambos extremos (0º y 160º en este caso). Al finalizar con todos los píxeles de la celda se agregan todas las magnitudes que hay para cada ángulo para así formar el histograma (Figura 15)

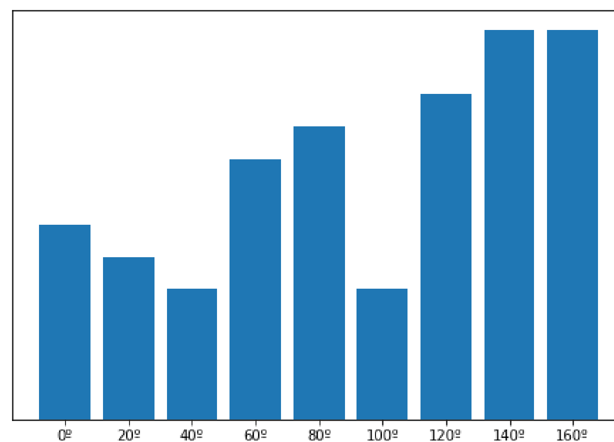


Figura 15: Ejemplo de histograma para HOG

6. Normalización y concatenación: Para evitar que variaciones de iluminación afecten considerablemente a los histogramas es recomendable normalizarlos, de esta forma es necesaria una ventana deslizante que vaya recorriendo la imagen entera y normalizando los diferentes histogramas (la normalización se puede realizar de 4 en 4 histogramas, de 9 en 9 histogramas y así sucesivamente), tras la normalización solo habrá que concatenar todos los *arrays* normalizados (histogramas individuales) para formar el Histograma de Gradientes Orientados.

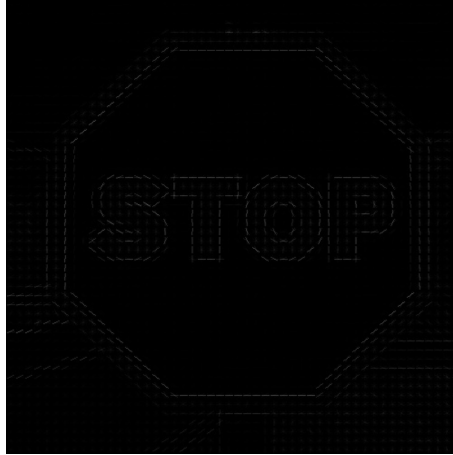


Figura 16: Ejemplo de HOG aplicado a una imagen de Señal de Stop

2.3. *Non-Max Suppression* (Supresión de no máximos)

Generalmente el modelo predice una gran cantidad de *Bounding Boxes* en su salida, de forma que sea necesario un proceso de filtración de estas para seleccionar entre todas las posibles candidatas para la detección de un objeto la que de verdad se adecúa a sus límites en función de la confianza de la predicción y del IOU (*Intersection Over Union*).

2.3.1. *Intersection Over Union* (IOU)

Es un término utilizado para determinar el grado de superposición (*overlap*) entre dos cajas o *Bounding Boxes*, es muy utilizado en la Visión Artificial y más concretamente en la detección de objetos mediante *Bounding Boxes*. El IOU se define de la siguiente forma:

$$IOU = \frac{\textit{Área de la intersección de las dos Bounding Boxes}}{\textit{Área de la unión de las dos Bounding Boxes}}, \tag{21}$$

que con una representación más gráfica se vería asi (Figura 17).

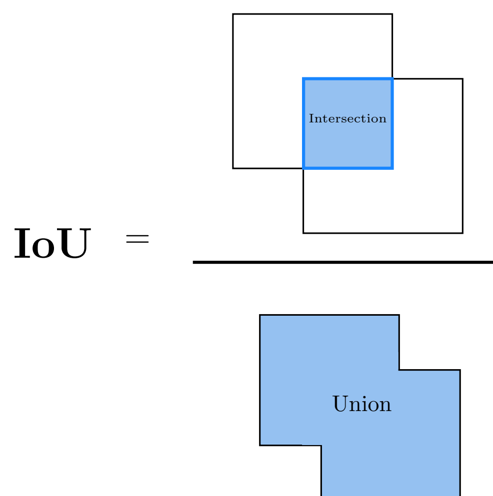


Figura 17: Descripción gráfica de la operación que se realiza en el cálculo del IoU

2.3.2. Algoritmo *Non-Max Suppression*

Mientras existan *Bounding Boxes* en la lista provisional:

1. Seleccionar de la lista de posibles *Bounding Boxes* (lista provisional) la que mayor confianza tenga y agregarla a la lista de *Bounning Boxes* final

2. Comparar la BB agregada a la lista final con todas las BB de la lista provisional, es decir calcular el IOU y si es superior que un umbral definido N (ambas *Bounding Boxes* abarcan prácticamente el mismo objeto) se eliminan de la lista de BB provisionales.

Figura 18: Ejemplo de Supresión de No Máximos

2.4. *Deep Learning*

El *Deep Learning* o Aprendizaje profundo es un subcampo del, ya mencionado, *Machine Learning*, está relacionado directamente con los algoritmos inspirados en la estructura y función del cerebro y sus neuronas, las redes neuronales (también explicadas en el apartado 2.1.1 Perceptrón Multicapa). El *Deep Learning* permite que los modelos computacionales que se componen de múltiples capas de procesamiento aprendan representaciones de datos con múltiples niveles de abstracción, gracias a esto se han obtenido avances en el reconocimiento de voz y texto mediante las redes recurrentes y en el reconocimiento y detección de objetos, gracias a las Redes Neuronales Convolucionales profundas, estos avances también se encuentran en dominios muy diversos como en el descubrimiento de un fármaco o en la genómica [11].

2.4.1. CNN (*Convolutional Neural Network*)

Las Redes Neuronales Convolucionales (*Convolutional Neural Networks* (CNN) en inglés) son un tipo de redes principalmente orientadas a trabajar directamente con imágenes, sin ser necesario, por tanto, una extracción de características como HOG antes de la aplicación de la red, ya que la propia red realiza la extracción de las características. Estas redes son ampliamente utilizadas para la detección de objetos en el campo de la Visión Artificial. Las CNN tienen similitudes con las Redes Neuronales convencionales, ya que, están formadas por neuronas, tienen pesos y sesgos, la red recibe un *input*, que en este caso es una imagen y calcula el output para esa clase, es decir, desde una imagen sin ningún tipo de procesamiento la red predice una clase o *output*, además de todavía conservar la función de coste de la red para la actualización de pesos. En el caso de las CNN's lo que se intenta con ellas es minimizar al máximo las dimensiones de la imagen conservando de esa forma las características más útiles de esta y suprimiendo las irrelevantes, así es más

fácil de procesar y los cálculos son más rápidos. En esta red las características se detectan progresivamente por capas, en las primeras capas se detectan rasgos generales como bordes, cambios de intensidad, etc, sin embargo cuanto más profunda sea la red empieza a detectar características más específicas, como ojos y cejas (en el caso de una cara), letras, etc, de forma que cuando la imagen pase por toda la red esta sea capaz de reconocer un objeto íntegro a partir de diversas características que se encuentran en el.

2.4.1.1. Convoluciones

Las convoluciones son usadas en las CNN entre otras cosas, para resaltar características de la entrada, como diferentes tipos de bordes (verticales, horizontales) o características concretas. La base es muy simple, contamos con un filtro (también llamado *kernel*) que actua de forma similar que los pesos de una Red Neuronal como la explicada anteriormente, es decir, van a ser modificados a lo largo del entrenamiento de la red para lograr unos mejores resultados. Estos *kernels* tienen un tamaño de z_h x z_w x z_c (en el primer ejemplo lo haremos con solo 1 canal, $z_c = 1$) y además contamos con una entrada de x_h x x_w x x_c, es importante que la profundidad, es decir x_c y z_c sean iguales, $x_c = z_c$ para que la convolución pueda ocurrir. Aplicando esto al ejemplo tenemos un filtro y una ventana que debe ir desplazándose a lo largo y ancho de la entrada s veces después de hacer cada operación. La operación en este caso es una suma de multiplicaciones, el valor resultante se pondrá en la posición que le corresponde de la salida (Figura 19)

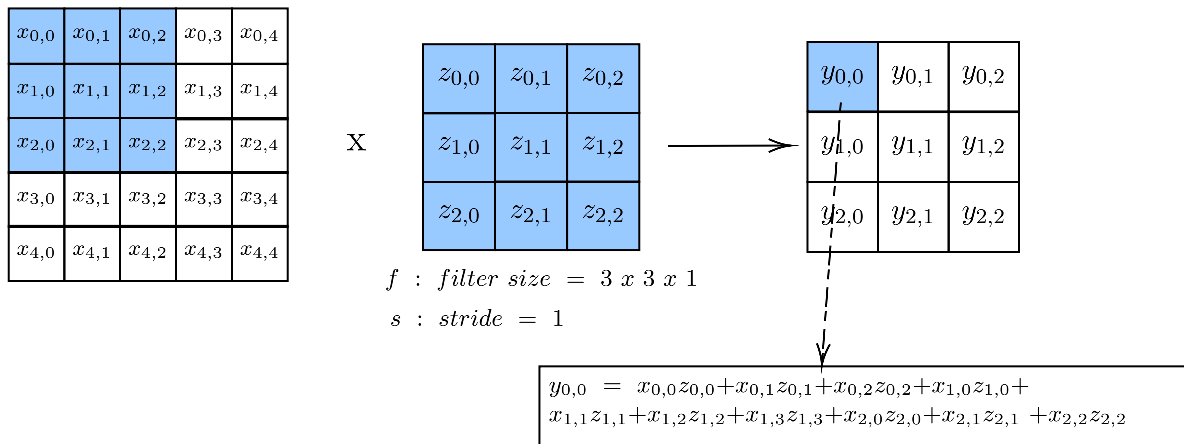


Figura 19: Convolución con filtro de 1 Dimensión

2.4.1.2. *Padding*

El *padding* o rellenado consiste en rellenar con valores los bordes de una matriz con el objetivo de que a la hora de realizar una convolución el tamaño de la matriz de entrada sea similar que el tamaño de la matriz de salida. Esto se logra creando bordes extra en la matriz para que al ser reducido el tamaño de la misma al realizar la convolución el tamaño no se vea afectado. En el ejemplo de la Figura 20 se ha utilizado un *kernel* de tamaño 2 x 2 y un *stride* de 1, es por eso que solo ha sido necesario una capa extra de rellenado, si el tamaño del *kernel* hubiese superado el 2 x 2 serían necesarias más capas de *padding* siguiendo la ecuación (en el caso de *kernels* con tamaños inpares, que son los

habitualmente usados)

$$N^o capas\ de\ Padding = \frac{shape(kernel) - 1}{2} \tag{22}$$

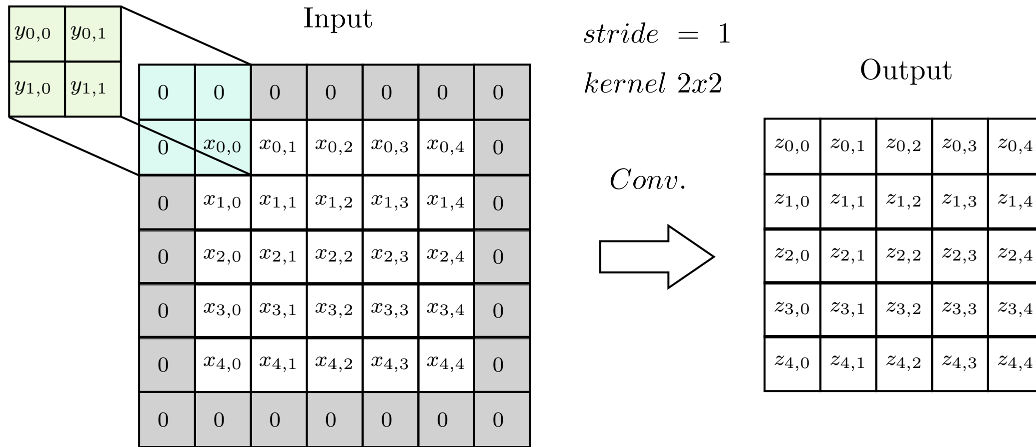


Figura 20: Ejemplo de *Padding*

Existen diferentes tipos de rellenado:

1. *Zero padding*: Consiste en rellenar con 0's.
2. *Reflection padding*: Consiste en, utilizando el propio borde de la matriz como si fuera un espejo, se copian al borde extra exterior los valores que están contiguos a este "espejo" en la parte interior.
3. *Replication padding*: Consiste en copiar los valores del borde exterior de la matriz en el borde o bordes extra del rellenado [12].

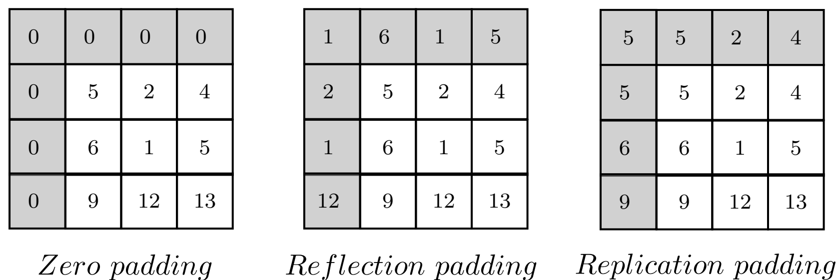


Figura 21: Ejemplo de tipos de *Padding*

2.4.1.3. *Pooling*

Las capas de *Pooling* (agrupación en castellano) se utilizan para reducir las dimensiones de los mapas de características. De este modo, se reduce el número de parámetros que hay que aprender y la cantidad de cálculos realizados en la red. Las capas de *Pooling* resumen las características presentes en una región del mapa de características generado por una capa de convolución.

El *Pooling* consiste en, teniendo un tamaño de filtro (*filter size*) especificado como f (ventana de fxf) y un paso (*stride*) especificado como s, ir recorriendo la matriz de entrada de izquierda a derecha y de arriba a abajo hallando los máximos de los valores de las ventanas de tamaño f y avanzando con un paso s (Figura 22).

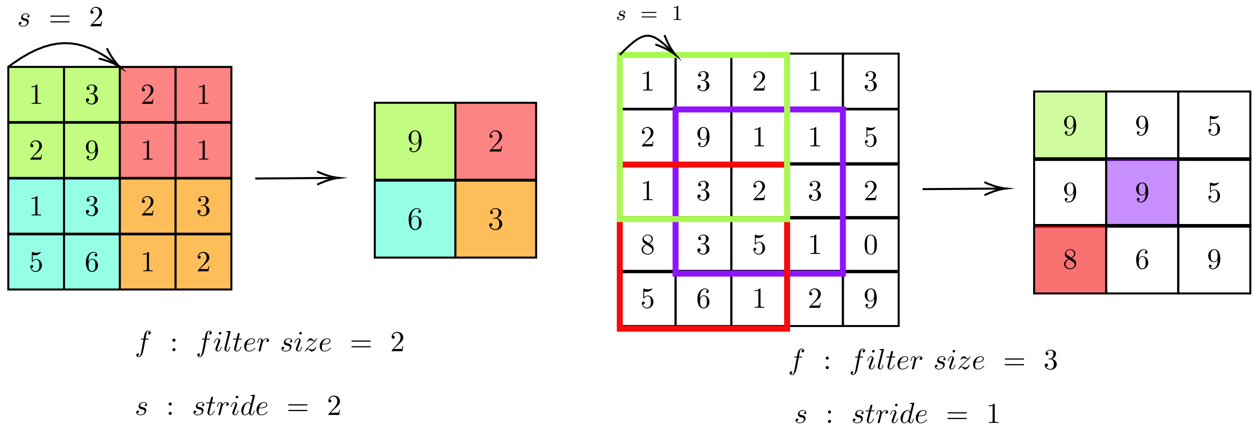


Figura 22: Ejemplo de funcionamiento de *Pooling layers*. En el caso de la inzquierda tenemos un tamaño de filtro o ventana de 2 y un *stride* o paso de 2, empezando por arriba a la izquierda $max\{1,3,2,9\} = 9$, desplazamos dos posiciones y $max\{2,1,1,1\} = 2$ y así con toda la matriz de entrada.
En el caso de la derecha tenemos $tamaño\ de\ filtro = 3$ y $stride = 1$, comenzando arriba a la izquierda con el recuadro verde tenemos $max\{1,3,2,2,9,1,1,3,2\} = 9$, con una ventana de 3x3 y un *stride* de 1 vamos avanzando, en el caso del recuadro morado $max\{9,1,1,3,2,3,3,5,1\} = 9$ y similar con el recuadro rojo inferior $max\{1,3,2,8,3,5,5,6,1\} = 8$

Es necesario recalcar que en el caso en el que la entrada fuese multicanal, es decir n_h x n_w x n_c (altura x anchura x numero de canales), las capas de *Max Pooling* se aplicarian canal a canal independientemente, de forma que si la entrada es 4x4x3, *filter size* = 2, *stride* = 2 la salida sera una matriz 2x2x3 aplicando el *Max Pooling* para cada uno de los canales de la entrada por separado.

2.4.1.4. Capas totalmente conectadas (*fully connected layers*)

Las capas totalmente conectadas o *fully connected layers* en inglés son, como el propio nombre indica, las capas que están totalmente conectadas a la siguiente capa mediante enlaces con sus respectivos pesos asignados. Esto ya ha sido utilizado antes en el apartado 2.1.1 Perceptron Multicapa y en este caso ejerce la misma función, ya que en las CNN el entrenamiento es similar a las NN clásicas pero con alguna diferencia. Se hace la propagación hacia adelante sobre la red para hallar la función de coste, tras esto se aplica el algoritmo de *backpropagation* para la modificación de pesos para la reducción de la función de coste, sin embargo en el caso de las CNN además de la modificación de pesos en el *backpropagation* también se da la modificación de los *kernels* de las convoluciones para así reducir la función de coste y obtener una mejor solución.

2.4.1.5. Función de activación de la última capa (*softmax layer*)

Ya hemos mencionado antes la labor de las funciones de activación, sin embargo en este caso es diferente, ya que en las CNN las funciones de activación de la ultima capa suelen ser diferentes al resto de capas, debido a que esta función de activación debe de ser seleccionada en función de tareas, es decir, dependiendo de si la CNN es Multiclase o no y si es multietiqueta o no. En nuestro caso como la CNN es multiclase y monoetiqueta (La red puede predecir entre diferentes clases, Coche, Señal, persona, etc, pero solo puede predecir uno a la vez, es decir un objeto no puede ser persona y señal a la vez) entonces la función de activación que hay que utilizar es *softmax*. La función de activación *softmax* convierte el vector de números que le llega de la capa anterior (penúltima capa) a un vector de probabilidades de cada una de las clases.

En la Figura 23 se muestra la unión de los 4 conceptos anteriores en forma de arquitectura de una Red Neuronal Convolucional.

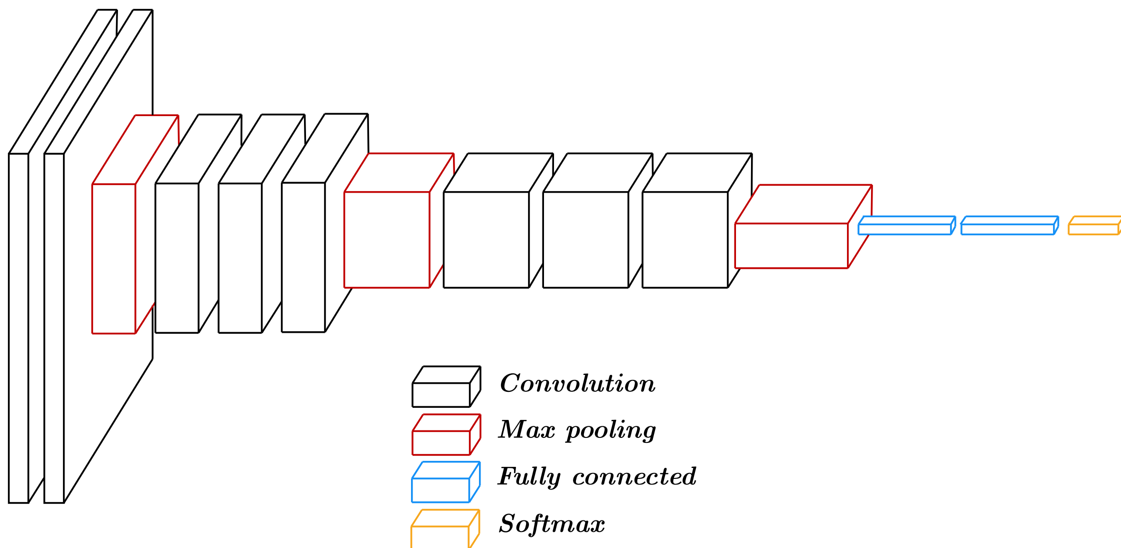


Figura 23: Arquitectura de CNN

2.5. Evaluación de detección de objetos: mAP (*Mean Average Precision*)

La media de las precisiones medias o *Mean Average Precision* (mAP) es una métrica ampliamente utilizada para la evaluación de modelos de detección de objetos, como YOLO, R-CNN, Fast R-CNN etc. Para el cálculo de la métrica mAP son necesarias una serie de submétricas que son las siguientes:

1. Matriz de confusión: Esta permite la visualización del desempeño del modelo utilizado para la clasificación (Explicada en el apartado 2.1.4 Evaluación de clasificación de imágenes: *f1-score*).

2. *Intersection Over Union* (IOU): Determina el grado de superposición entre dos objetos, ha sido explicado en el apartado 2.3.1 *Intersection Over Union (IOU).*

3. *Recall*: Mide como de bien se predicen los verdaderos positivos sobre el total de las predicciones (Explicado en el apartado 2.1.4 Evaluación de clasificación de imágenes: *f1-score*).

4. *Precision*: Mide como de bien se predicen los verdaderos positivos sobre todas las predicciones positivas (Explicado en el apartado 2.1.4 Evaluación de clasificación de imágenes: *f1-score*).

Con las métricas mencionadas ya calculadas es necesario hallar el área bajo la curva *Precision-Recall*, siendo este el AP. Al hacer la media entre los AP en función de las clases que se hayan utilizado (Conteniendo cada una de estas clases diferentes AP en función de umbrales que se hayan utilizado en el IOU) se obtiene la métrica mAP:

$$mAP = \frac{1}{n} \sum_{k=1}^{k=n} AP_k, \tag{23}$$

siendo AP_k el AP de la clase k y n el número de clases

3. Algoritmos de detección de objetos basados en *Deep Learning*

3.1. YOLO (*You Only Look Once*)

YOLO (*You Only Look Once*), desarrollado por Joseph Redmon, Santosh Divvala, Ross Girshick y Ali Farhadi en 2015 [1].

Consiste en una única Red Neuronal profunda que predice *Bounding Boxes* y sus confianzas por cada clase desde las imágenes completas en un solo ciclo, gracias a que el proceso entero de detección se base únicamente en una sola red este se puede optimizar fácilmente de principio a fin basándose en el rendimiento de la propia detección. YOLO utiliza características de toda la imagen para las predicciones de las *Bounding Boxes*. El modelo básico de YOLO llega a procesar imágenes en tiempo real a 45 fotogramas por segundo, en cuanto al modelo de *Fast* YOLO procesa 155 fotogramas por segundo.

3.1.0.1. Versiones de YOLO

1. YOLOv2: Publicada en 2017 por el mismo equipo que creo originalmente YOLO en 2016, entre los cuales se encuentra Joseph Redmon, se realizaron mejoras en la arquitectura, además de incluir en el modelo *Batch Normalization*, resoluciones mayores y la incorporación de las *Anchor Boxes* [13].

2. YOLOv3: Publicada en 2018 por el mismo equipo que YOLOv3, siendo esta la última versión oficial sacada por este grupo, ya que, de aquí en adelante las versiones de YOLO han sido publicadas por diferentes grupos de investigación con diferentes objetivos. En esta versión, entre otras cosas, se añadieron conexiones con la red troncal, además de hacer predicciones en tres niveles de granularidad para conseguir un mayor rendimiento en la predicción de objetos pequeños [14].

3. YOLOv4: Publicada en Abril de 2020 por Alexey Bochkovskiy que, como ya se ha comentado, fue la primera versión de YOLO que no fue lanzada por el equipo de Joseph Redmon. Entre otras cosas incluye mejoras como la implementación de *Bag of Freebies* o bolsa de regalos, gracias a *Data Augmentation* o aumento de datos, aumentando asi el conjunto de datos de entrenamiento. En esta versión de YOLO también se empezó a utilizar la función de activación de Mish [15].

4. YOLOv5: Lanzado en Junio de 2020 por Glenn Jocher, poco después que la anterior versión YOLOv4, primera versión de YOLO que no ha sido publicada con un *paper* asociado. En esta versión se hizo el cambio a la librería PyTorch, además de esto se continuó con el aumento de datos [16].

5. YOLOv6: Lanzado en Junio de 2022 por el equipo técnico de Meituan. Además de cambios en la arquitectura respecto a su versión anterior YOLOv5 para así adaptarla más al hardware esta versión implementa algunas mejoras en la canalización de la capacitación, como la capacitación *anchor free*, la asignación de etiquetas simOTA y la perdida de regresión del cuadrado SIoU [17].

6. YOLOv7: Lanzado en Julio de 2022 por WongKinYiu y Alexey Bochkovskiy. En esta versión se realizan una serie de cambios en la arquitectura, además de en las rutinas de entrenamiento, para asi lograr predecir *Bounding Boxes* con mayor precisión que otras versiones pero con una velocidad de inferencia similar [18].

Además de las versiones de YOLO secuenciales vistas anteriormente también existen versiones lanzadas paralelamente a las ya vistas o con bases ligeramente diferentes

7. PP-YOLO: Lanzado en agosto de 2020 por Baidu, las siglas "PP" significan "PaddlePaddle" que es el *framework* de *Deep Learning* de Baidu. En esta versión se logra aumentar el rendimiento de YOLOv4 gracias a la regularicación *DropBlock*, la matriz de *Non Max Supression* etc [19].

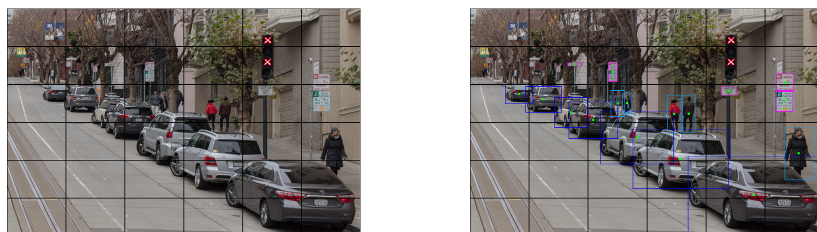
8. YOLOv4 escalado: Lanzado en noviembre de 2020 por Chien-Yao Wang, Alexey Bochkovskiy y Hong-Yuan Mark Liao. En esta versión se aumenta el tamaño de la red de YOLOv4 conservando su precisión y velocidad aprovechando las *Cross Stage Partial Networks* [20].

9. PP-YOLOv2: nuevamente creado por el equipo de Baidu y lanzado en Abril de 2021. En esta versión se realizaron ajustes menores para así lograr un mejor rendimiento respecto de su predecesora [21][22].

En nuestro caso utilizaremos la versión de YOLOv5 para nuestra comparación.

3.1.0.2. Codificación de las *Bounding Boxes*

Vamos a empezar con el método de codificación de las diferentes *Bounding Boxes* que utiliza YOLOv1. Primero de todo se empieza haciendo un *Grid*, es decir una malla encima de la imagen, en este caso hemos elegido una malla de 6x6 pero únicamente para el ejemplo, esto puede variar, teniendo esta malla y las *Bounding Boxes* junto con los puntos centrales de estas señaladas (Figura 24) podemos empezar la codificación.



(a) Malla de 7x7 (color negro) sobre la imagen original

(b) *Bounding Boxes* junto con los puntos centrales (verde) y la malla

Figura 24: Prerequisitos para codificación YOLO

Ahora sabiendo esto es necesario ir codificando en forma de *array* (Figura 25) cada una de las celdas del *Grid*, de forma que si ninguno de los centros de las *Bounding Boxes* se encuentra dentro de la celda del *Grid* el *array* mostrara que está vacío (0 en una de las posiciones del *array*, valores sin importancia en el resto), sin embargo si algún centro (uno o más de uno) se encuentra dentro de la celda del *Grid* esto se indicara en el *array* (1 en una de las posiciones del array), junto con las coordenadas del centro de la *Bounding Box*, la altura y anchura de la misma y la clase del objeto que está rodeando esa *Bounding Box*. Esto se realiza para cada una de las celdas del *Grid*. Es necesario recalcar que en el caso de los parámetros b_x y b_y tienen que estar entre 0 y 1, ya que las coordenadas de estos se toman en función de la celda, asignando $(0, 0)$ al punto superior izquierdo de la celda y $(1, 1)$ al punto inferior derecho.

$$\begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ \vdots \end{bmatrix} \rightarrow$$

p_c → *Indicará la probabilidad de detectar el centro de una BBox en la celda*

b_x → *Indicará la coordenada x del centro de la BBox detectada (respecto de la celda)*

b_y → *Indicará la coordenada y del centro de la BBox detectada (respecto de la celda)*

b_h → *Indicará la altura (height) de la Bounding Box detectada*

b_w → *Indicará la anchura (width) de la Bounding Box detectada*

c_1 → *Indicará la probabilidad de la BBox de pertenecer a la clase 1 (entre 0 y 1)*

c_2 → *Indicará la probabilidad de la BBox de pertenecer a la clase 2 (entre 0 y 1)*

c_3 → *Indicará la probabilidad de la BBox de pertenecer a la clase 3 (entre 0 y 1)*

c_n → *Esto se repite para todas las clases (n = número de clases)*

Figura 25: Explicación parámetros de codificación de YOLO

Vamos con algunos ejemplos, en este primer caso tenemos resaltado en rojo oscuro la celda en la que nos encontramos (la imagen es la misma que en la figura anterior sin embargo ha sido aumentada una zona para mayor claridad), se puede observar que parte de las *Bounding Boxes* de 2 de los coches de la parte superior se encuentran dentro de la celda, sin embargo ningún centro de los coches se encuentra dentro de la celda, por lo cual el parámetro $p_c = 0$ y el resto de parámetros del *array* no importan (Figura 26).

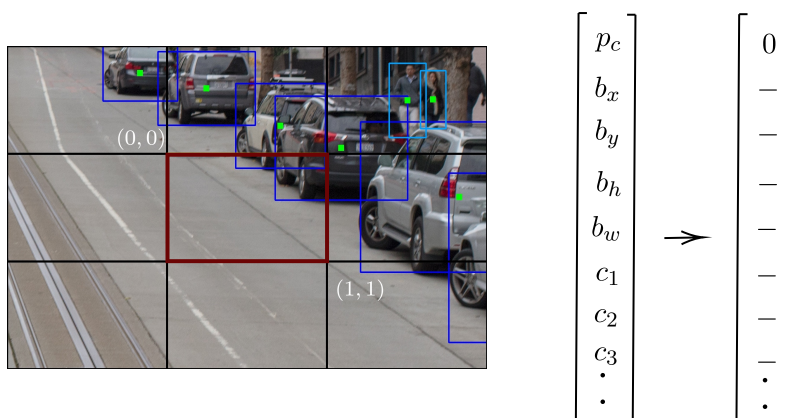


Figura 26: Ejemplo de codificación YOLO sin objeto

Como segundo ejemplo tenemos el siguiente, dentro de la misma celda se encuentran los centros de las *Bounding Boxes* de 2 personas y 1 coche, lo que tenemos que hacer en este caso es, en vez de codificarlo en forma de *array* de 8x1 (8 filas (en nuestro caso ya que tenemos 3 clases, si no serían más de 8) y 1 columna) lo codificaremos en forma de 16x1 o 24x1 ... Básicamente apilaremos en forma de *array* los diferentes *arrays* de los centros. En el ejemplo (Figura 27) las primero 8 posiciones del *array* pertenecen a la *Bounding Box* del coche (el parámetro $c_1 = 1$), las siguientes 8 posiciones (9-16) pertenecen a la *Bounding Box* de una persona (el parámetro $c_2 = 1$) y faltaría la representación de la segunda persona que aparece en la celda que iría más abajo, de forma que se formaría un *array* de tamaño 24x1 (24 filas (8 para el coche, 8 para 1 persona y 8 para la otra persona) y 1 columna).

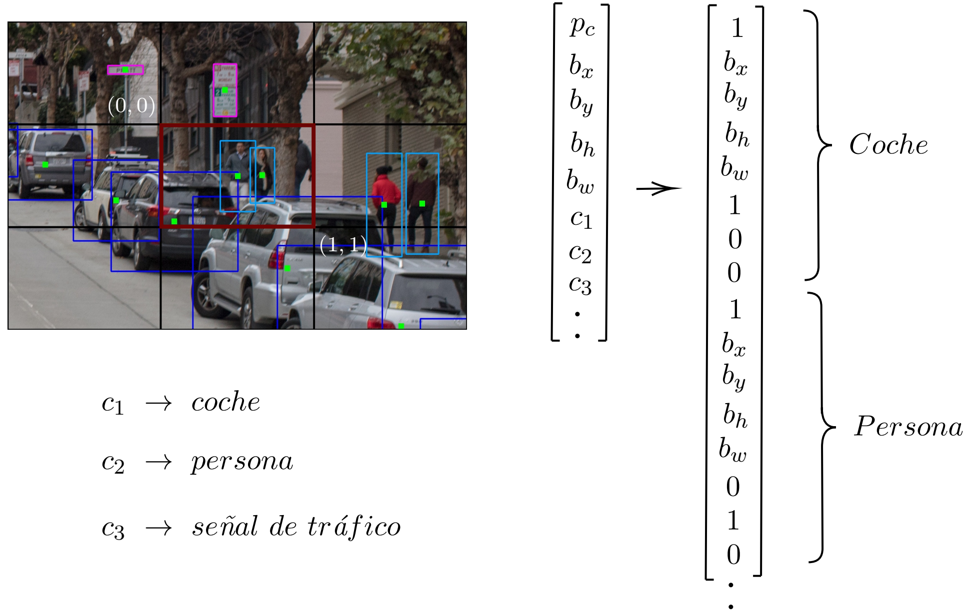


Figura 27: Ejemplo de codificación YOLO con más de 1 objeto

Toda esta codificación se realiza gracias a la arquitectura de la red de la primera versión de *You Only Look Once* (Figura 28). En esta están representados los diferentes tamaños por los que va pasando la codificación de la imagen, a la izquierda del todo tenemos una imagen de 448x448 píxeles en color (RGB), es decir 448 x 448 x 3 y a la derecha del todo tenemos la codificación de las *Bounding Boxes* en forma de matriz de *arrays* (7 x 7 x 30), en la parte inferior de la imagen se puede ver cada una de las capas convolucionales y de max *pooling* además de las capas *Fully Connected* por las que va pasando la imagen. En el primer caso tenemos una capa convolucional de 7 x 7 x 64-s-2, es decir que tenemos 64 kernels de 7 x 7 para hacer la convolución con un paso o *stride* de 2, después de esto se aplica un *Max Pooling* con un tamaño de filtro de 2 x 2 y un *stride* de 2 (conceptos explicados en el apartado 2.4.1 CNN (*Convolutional Neural Network*)).

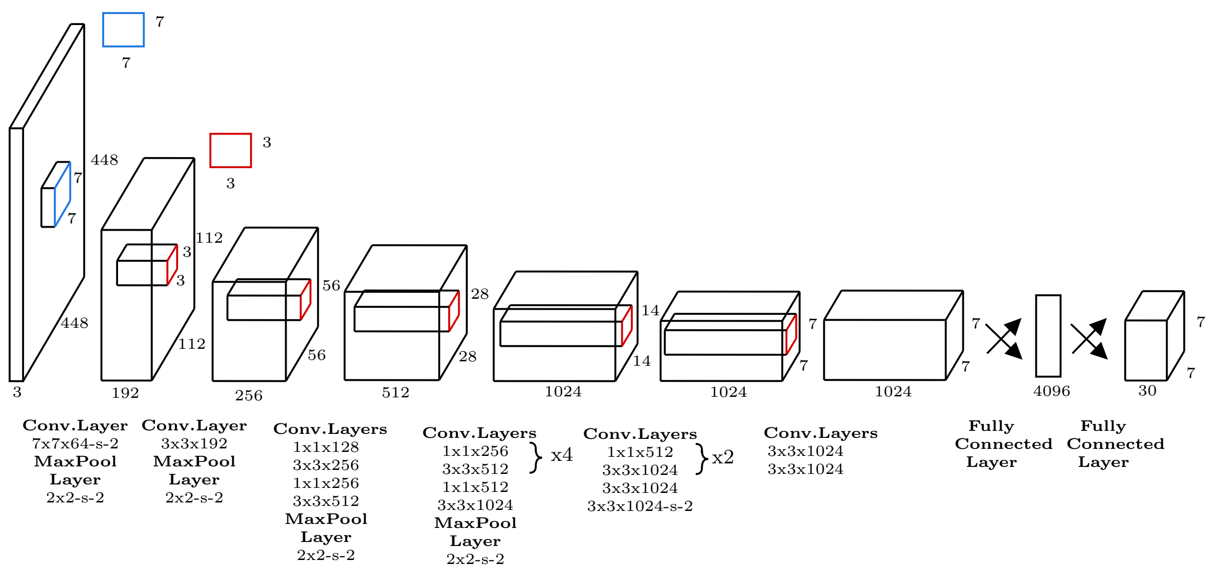


Figura 28: Arquitectura de la primera versión de YOLO

3.2. SSD (*Single Shot Detector*)

SSD (*Single Shot Detector* o *Single Shot Multibox Detector*) fue desarrollado por Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu y Alexander C. Berg en 2016 [2]. El SSD es un algoritmo de detección de objetos que, muy similar al anteriormente mencionado YOLO, realiza la predicción de *Bounding Boxes* y sus confianzas para cada clase de una sola vez (de ahí lo de *Single Shot* o único disparo). La arquitectura de esta Red Neuronal Convolucional se separa en dos partes, la primera que consiste en una modificación de la VGG-16 y una segunda parte que consiste en una serie de capas convolucionales que ayudan a SSD a la extracción de características a diferentes escalas y a la disminución del tamaño del *input*. El modelo básico de SSD llega a procesar 59 fotogramas por segundo.

3.2.1. Arquitectura de SSD

3.2.1.1. VGG-16

VGG-16 fue desarrollado por Karen Simonyan y Andrew Zisserman y lanzado en 2015 [23]. Las siglas se deben a, en primer lugar *Visual Geometry Group* (VGG) que es el grupo al que pertenecían y 16 es el número de capas con pesos que tiene la red. La VGG-16 es un tipo de Red Neuronal Convolucional considerada uno de los mejores modelos de visión artificial en el momento de su lanzamiento. VGG-16 es capaz lograr una precisión del 92,7 % utilizando las 14 millones de imágenes del *dataset* de *ImageNet* las cuales pertenecen a 1000 clases diferentes, además de esto es ampliamente utilizada ya que es muy sencillo realizar *Transfer Learning* con esta red [24]. La arquitectura consiste en 13 capas convolucionales (*convolutional layers*), 5 capas de *Max pooling* y 3 capas densas o *fully connected layers* que hacen un total de 21 capas, de las cuales solamente 16 tienen pesos entrenables. El input de esta red es una imagen de 224 x 224 x 3, es decir una imagen de 224 x 224 con los 3 canales RGB. Esta red utiliza siempre kernels de 3 x 3 con un *stride* de 1 en el caso de las capas convolucionales y un filtro de 2 x 2 con un *stride* de 2 para las capas de *Max Pooling*. Tras las capas convolucionales y de *Max Pool* nos encontramos con 2 capas totalmente conectadas (*Fully connected layers*) de 1 x 1 x 4096, seguidas de una capa de clasificación de 1 x 1 x 1000 continuando con la capa final o capa *softmax* de 1 x 1 x 1000 en la que cada entrada representa 1 de las 1000 clases que puede clasificar esta red (Figura 29). En el caso de la utilización de VGG-16 en el modelo de SSD se eliminan la última capa de *Max Pool*, las capas *fully connected* y la capa *softmax* para ser así sustituidas por el resto de capas convolucionales propias del modelo SSD. En el modelo SSD VGG-16 se utiliza para crear un mapa de características, el cual codifica información semántica en diferentes regiones de la imagen. Este mapa de características tienen un tamaño de 38 x 38 x 512.

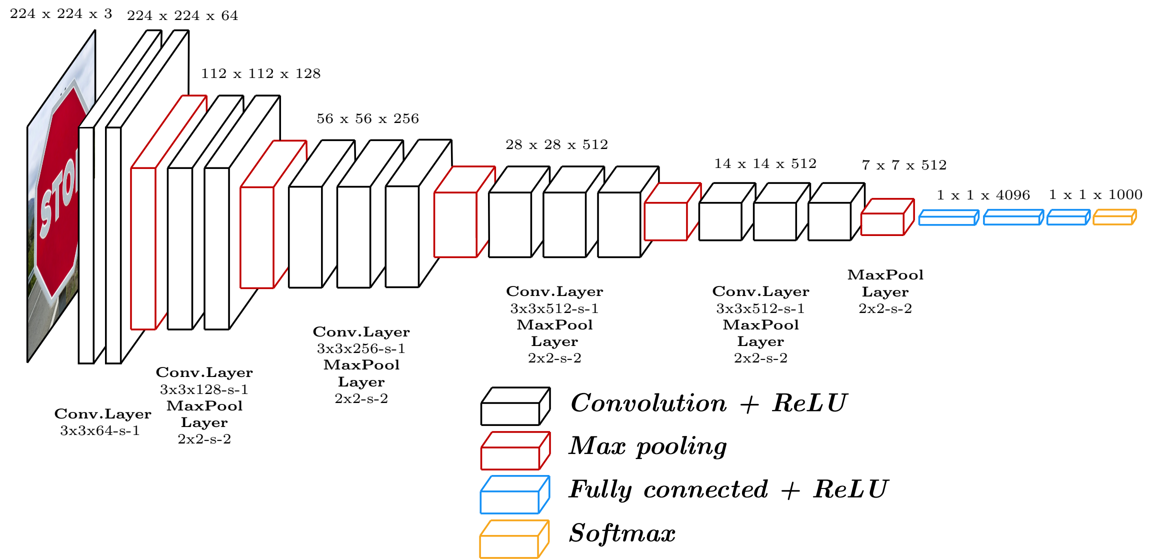


Figura 29: Arquitectura de VGG-16

3.2.1.2. Capas propias de SSD

Ahora vamos a explicar el segundo conjunto de capas del modelo SSD. Tras VGG-16 se añadieron una serie de capas convolucionales las cuales ayudan a la extracción de características a diferentes escalas además de a la disminución progresiva del tamaño, en este proceso se generan las *Bounding Boxes* y sus confianzas. Consiste en un conjunto de 5 capas convolucionales en las que su salida final es un conjunto de 8732 *Bounding Boxes* por cada clase con sus confianzas relacionadas con cada una de ellas.

3.2.1.3. *Non-Maximum Suppression*

Para finalizar con la detección y tras la VGG-16 y las capas convolucionales se realiza una Supresión de No Maximos para filtrar las ya mencionadas 8732 *Bounding Boxes* por cada clase, ya que es probable que haya muchas *bounding boxes* con confianzas muy bajas y otras en las que diversas *bounding boxes* señalen el mismo objeto, por tanto haya que seleccionar la que mejor lo detecte (La Supresión de No Máximos ha sido explicada en 2.3 *Non-Max Suppression* (Supresión de no máximos)). La arquitectura completa se muestra en la Figura 30.

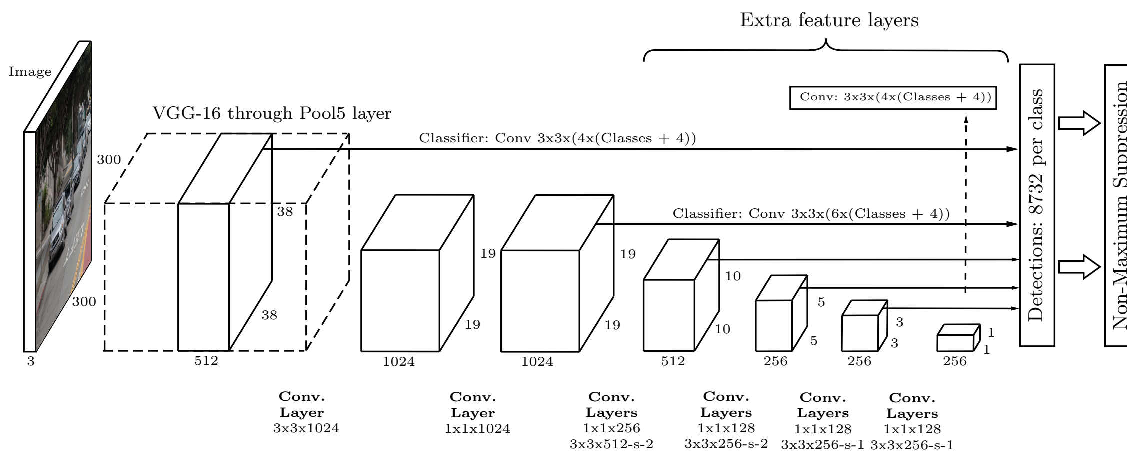


Figura 30: Arquitectura de *Single Shot Detector*

3.2.2. Funcionamiento de SSD

3.2.2.1. Extracción de características (*Feature Extraction*)

La entrada del modelo de SSD es una imagen de 300 x 300 x 3 (canal RGB), Tras pasar por las diversas capas convolucionales de la arquitectura de VGG-16 se obtiene un mapa de características de 38 x 38 x 512, sin embargo, dado que SSD tiene que lograr realizar detecciones de diversos tamaños se añadieron las capas convolucionales propias de SSD tras al arquitectura VGG-16, sin embargo la salida de VGG no solamente se debe pasar a través del resto de capas convolucionales, sino que cada vez que la entrada disminuya y se pase a través de una capa se guardará el tensor, de forma que al finalizar y llegar a la salida de la red que tiene un tamaño de 1 x 1 x 256 se hayan guardado los tensores de 38 x 38 x 512 (salida de VGG-16), 19 x 19 x 1024, 10 x 10 x 512 , 5 x 5 x 256 y 3 x 3 x 256 (representados en la Figura 35 en forma de pirámide en el lado izquierdo), de esta forma ahora tenemos varios mapas de características de diversos tamaños para así poder realizar la detección desde objetos grandes hasta muy pequeños.

3.2.2.2. Fase de detección (*Detection Head*)

Con los 5 mapas de características de diversos tamaños que hemos obtenido en la primera fase ahora es necesario realizar la detección y predicción de las *Bounding Boxes* y sus confianzas. El algoritmo de SSD funciona de la siguiente forma, se separa en celdas toda la imagen (generalmente son celdas pequeñas, en este caso para el ejemplo gráfico serán grandes), se recorre toda la imagen en forma de ventana deslizante con todas las posibles *anchor boxes* centradas en el punto central de la celda a lo largo de toda la imagen (Figura 33 (En este caso las *Anchor Boxes* no se encuentran en todas y cada una de las celdas pero únicamente está representado así para mayor claridad del dibujo)).

Anchor Boxes

Sabiendo que cada objeto tiene un tamaño concreto y no es posible abarcar los bordes de todos los objetos con una *Bounding Box* con un tamaño definido es necesario definir diferentes tamaños que se amolden a los diversos objetos (Figura 31), esto se conoce como *Anchor Boxes*, esto además también soluciona el problema de que varios objetos con distintas formas tengan su centro en el mismo punto (Figura 32).

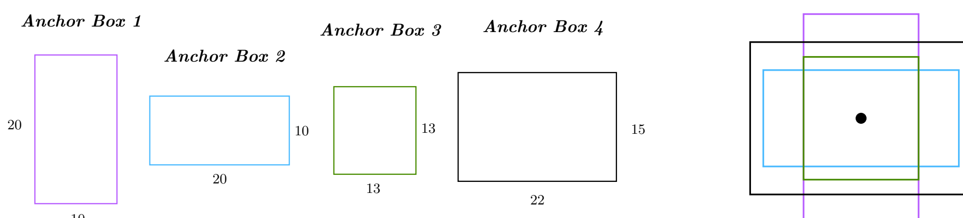


Figura 31: Diferentes tipos de *Anchor Boxes*

Figura 32: *Anchor Boxes* centradas en un punto

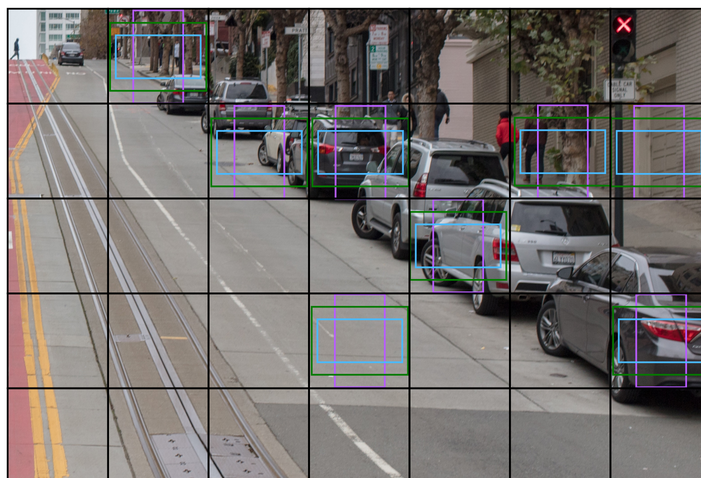


Figura 33: Ventana deslizante con *Anchor Boxes* en SSD

Cada uno de los recuadros de cada mapa de características se separa en dos partes (Figura 34, que son los mismos recuadros que los de la sección de *Detection Heads* de la Figura 35)), la primera en la que están las confianzas de las *Bounding Boxes* para cada uno de los tipos de *anchor box* y para cada clase, es decir tiene un tamaño de $S\ x\ S\ x\ (n_a \cdot n_c)$ (las variables y la notación están explicadas en la Figura 34) y la segunda que tiene las coordenadas de las *Bounding Boxes*, compuesta por la NN *Output*, que es el ajuste (predicción) que sale directamente de la red, la *Default Box*, que es una de las *anchor boxes* que coincide con un objeto, es decir la *Default Box* es la *Bounding Box* como tal y la salida de la red (*NN Output*) es cuanto hay que desplazar y modificar la *Default Box* para que se adapte al objeto, y la *Bounding Box* resultante es la *prediction* y se calcula como

$$\begin{aligned} b_x &= d_x + d_w \beta_x, \\ b_y &= d_y + d_h \beta_y, \\ b_w &= d_w e^{\beta_w}, \\ b_h &= d_h e^{\beta_h} \end{aligned} \quad (24)$$

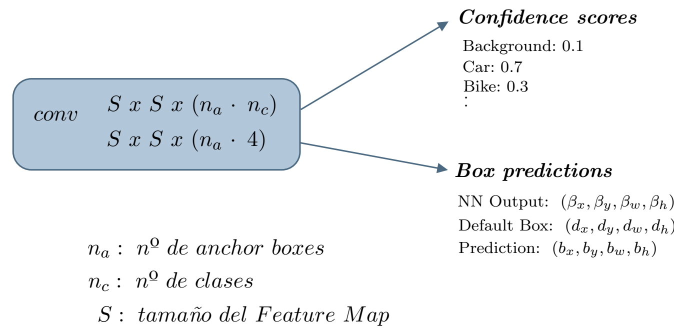


Figura 34: Codificación de la detección en SSD

Gracias a que hay 5 mapas de características diferentes la imagen es recorrida con el conjunto de todas las *Anchor Boxes* de 5 tamaños diferentes, esto ayuda a la detección de objetos grandes y pequeños, de forma que, si el mapa de características es más pequeño, es decir está más arriba en la pirámide (parte izquierda de la Figura 35) se detectarán con más facilidad los objetos más grandes y si el mapa de características es más grande, es decir utilizamos mapas de la base de la pirámide se detectarán más fácilmente los objetos pequeños.

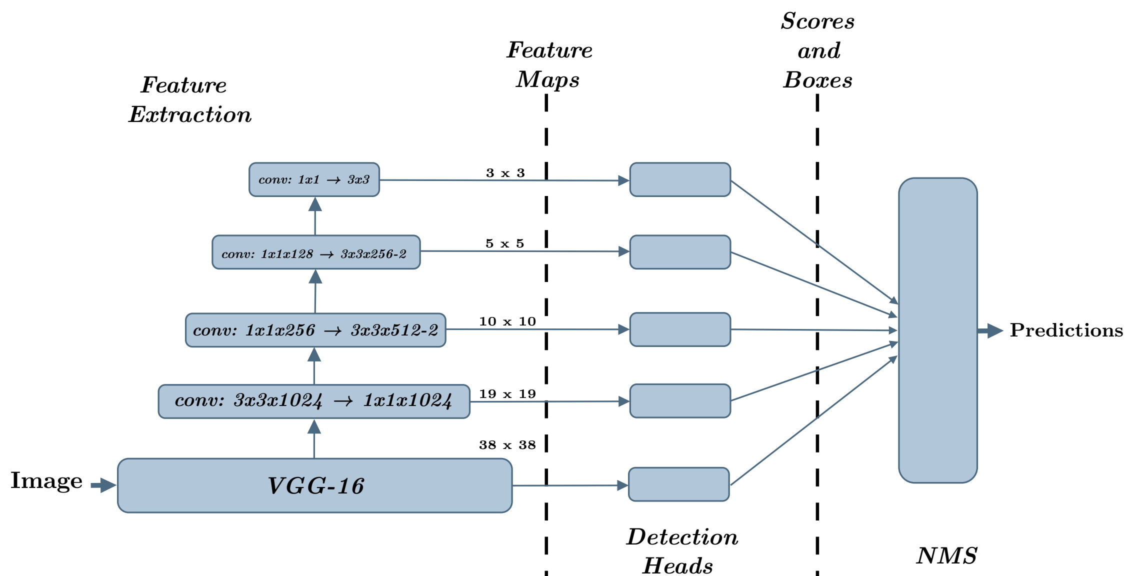


Figura 35: Arquitectura de *Single Shot Detector* por partes

4. Implementación de los modelos

En cuanto a la implementación de los modelos hemos decidido utilizar el lenguaje de programación *Python*, más concretamente en el software *Jupyter notebook*, ha sido así porque en este lenguaje es sencillo y practico realizar entrenamientos de modelos de *Machine Learning* con librerías determinadas, además de ser posible utilizar, gracias a otras librerías como Pytorch, la GPU del ordenador, de forma que sea posible entrenar más rápidamente modelos de *Deep Learning*.

4.1. *Dataset*

En cuanto al *dataset*, después de una larga búsqueda y de barajar diferentes ideas como la de utilizar diferentes *datasets* en función de la clase, por ejemplo el *dataset* de coches de *stanford* [25] o el *Specialized Cyclist Detection Dataset* [26], sin embargo, esta idea se acabó por desestimar porque era necesario buscar un *dataset* por clase y además era necesario buscar *datasets* que fuesen adecuados para el objetivo de este proyecto, la detección de objetos en carretera, ya que en el caso del *Specialized Cyclist Detection Dataset* si que son fotos tomadas con diversos climas, luces y ángulos, sin embargo en el caso del *dataset* de *stanford* las fotos son fotografías comerciales con muy poca variedad de ángulos y muy buena luz en todas (Figura 36).

(a) Fotografía de un coche del *Stanford dataset*

(b) Fotografía de una bici del *Specialized Cyclist Detection Dataset*

Figura 36: Ejemplo de diversos *datasets*

Teniendo esto en cuenta finalmente nos decidimos por utilizar un *dataset* de *berkeley Deep Drive* llamado BDD100k [27], en este caso utilizamos el conjunto de imágenes "100k" ya que dentro de este propio *dataset* también tienen vídeos separados en *frames* o diferentes *datasets* más pequeños. Nuestro *dataset* incluye 70000 imágenes de *train*, 10000 imágenes de *validation* y 20000 imágenes de *test*, los conjuntos de *train* y *val* incluyen además su correspondiente archivo de etiquetas (*labels*) en los que se almacenan datos de la imagen, como el tiempo (*weather*) o la escena en la que está sacada la foto (autopista, zona residencial etc) y información de las *Bounding Boxes* como la categoría, si está parcialmente tapado por otro objeto o no, y por supuesto, las coordenadas. En estas 100 mil imágenes de carretera que tiene el *dataset* existen un total de 274.591 señales de tráfico, 815.708 imágenes de coches, 8.217 imágenes de bicicletas, 104.609 imágenes de personas, 3.454 imágenes de motos y 5.166 imágenes de *riders* (conductores de moto o bici).

4.2. *Machine Learning*

En cuanto al apartado de *Machine Learning* se han utilizado una serie de librerías en función de las necesidades del proyecto que son las siguientes:

1. Pandas: utilizada para la lectura de los diferentes archivos (*train* y *val*) en formato json que incluían las características de las *Bounding Boxes.*.
2. Matplotlib.pyplot y cv2: Utilizadas para la lectura, escritura y muestra de imágenes.
3. Os: Utilizada para la lectura de las carpetas completas de imágenes a nivel de sistema operativo.
4. Numpy: Utilizada para el manejo de *arrays* y matrices en formato numpy.
5. Sklearnex y Sklearn: Usadas para el entrenamiento de los modelos de Perceptron multicapa, *Random Forest* y *Ada Boost*.
6. Time: Utilizada para medir los tiempos de entrenamiento de cada modelo.

Para todos los entrenamientos se han usado 3 conjuntos de datos diferentes:

1. Conjunto de *train*: Para el entrenamiento del modelo, 70 % del conjunto de datos totales.
2. Conjunto de *validation*: Para la modificación y ajuste de los parámetros, 20 % del conjunto de datos totales.
3. Conjunto de *test*: Para probar el modelo sobre un conjunto que no ha sido utilizado para su entrenamiento, 10 % del conjunto de datos totales.

Se han planteado dos propuestas para lograr el mejor detector posible:

- En el primer caso se realizan tres modelos generales, el primer modelo para *Bounding Boxes* cuadradas con las clases "*traffic sign*" , "*car*" , "*bike*" , "*motor*" y "otros" , incluyendo en este último parches realizados utilizando la ventana deslizante de objetos que no son parte del modelo, por ejemplo, la carretera, el cielo, tendidos eléctricos, personas etc. En el caso del segundo modelo son *Bounding Boxes* rectangulares horizontales formado por las clases "*car*" , "*bike*" , "*motor*" y "otros" , similar al anterior en esta última clase se incluyen personas, cielo, carretera, señales de tráfico etc. Por último el tercer modelo son *Bounding Boxes* rectangulares verticales formado por las clases "*person*" , "*rider*" , "*bike*" , "*motor*" y "otros" , la clase "otros" formada por coches, señales de tráfico, cielo, edificios etc. Estas clases han sido añadidas a los tres conjuntos de formas de *Bounding Box* (Rectangular horizontal, vertical y cuadrada) en función de sus características de tamaño, ya que en el caso de las señales de tráfico su forma de media se asemeja más a un cuadrado, sin embargo en el caso de los coches, como es posible verlos desde la parte trasera o de perfil se han incluido en el modelo de BB rectangulares horizontales y en el de cuadradas, y similar con las bicicletas, que en función del ángulo se pueden ver de las tres formas, rectangular vertical, rectangular horizontal o cuadrada.
- La segunda propuesta es realizar un modelo para cada clase, en el que cada uno de los modelos se entrene con dos clases, la clase principal y la clase otros, por ejemplo entrenar un *Random Forest* con la clase "coche" y la clase "otros" , en la clase "coche" están incluidas todas las imágenes de coches y en la clase "otros" todas las imágenes de NO coches, es decir, señales de tráfico, personas, motos, bicis, carretera etc.

En cualquiera de las dos propuestas se entrenan varios modelos (*Random Forest*, *Ada Boost* y Perceptrón Multicapa), realizando varias pruebas con diferente número de imágenes de entrenamiento para cada modelo, además de esto, gracias a la función “sklearn.model_selection.GridSearchCV” es posible hacer muchos entrenamientos variando ligeramente diferentes parámetros del modelo para así lograr la mejor combinación y con ello el mejor modelo. La fase de utilización de la ventana deslizante para la detección será utilizada en ambas propuestas tras el entrenamiento de los diversos modelos, en el caso de la primera metodología se recorrerá tres veces la imagen en función de la forma de la *Bounding Box* y en la segunda metodología se recorrerá una vez por cada modelo, es deci, el número de clases.

4.2.1. Perceptrón Multicapa

Para el Perceptrón Multicapa se utiliza la función “MLPClassifier” de “sklearn.neural_network” con parámetros fijos de un máximo de 1000 iteraciones ($max_iter = 1000$) y $random_state = 23$, además, gracias a la función “sklearn.model_selection.GridSearchCV” hemos entrenado el Perceptrón con una serie de hiperparámetros que son los siguientes:

- Capas ocultas de la red (‘hidden_layer_sizes’): Tupla con el número de neuronas para cada una de las capas ocultas de la red (e.g en el caso de ‘*hidden_layer_sizes*’ = (64, 64, 32, 4) tenemos 4 capas ocultas en las que la primera tiene 64 neuronas, la segunda 64, la tercera 32 y la cuarta 4 neuronas), el valor por defecto de este parámetro es (100,). Nuestros hiperparámetros para el entrenamiento han sido ‘*hidden_layer_sizes*’: $[(64, 32, 4), (256, 128, 32, 32, 16), (32, 16, 16, 8, 4, 2), (64, 16, 16, 4), (128, 64, 64, 16), (64, 64, 16, 8), (32, 32, 16, 8, 8, 4), (32, 32, 16, 4), (100, 50, 16, 8), (50, 25, 8)]$.
- Solucionador(‘solver’): Solucionador para la optimización de los pesos, existen 3 tipos:
  - *lbfgs*: Optimizador de la familia de los métodos cuasi-Newton.
  - *sgd*: Utiliza el descenso del gradiente estocástico.
  - *adam*: Se refiere a un optimizador estocástico basado en gradientes propuesto por Kingma, Diederik y Jimmy Ba.

  Generalmente el *solver* ‘*adam*’ funciona mejor para conjuntos de datos más grandes (miles de muestras), en cambio ‘*lbfgs*’ funciona mejor y suele converger más rápido con conjuntos de datos más pequeños. En este caso el parámetro por defecto es “*adam*”. En nuestro modelo hemos utilizado los hiperparámetros ‘*solver*’: $[“lbfgs”, “adam”]$,
- ‘alpha’: Es el mismo parámetro α mostrado anteriormente (Apartado 2.1.1.1 Proceso de aprendizaje), es decir, la constante de proporcionalidad entre el cambio del peso y el error cometido. El valor por defecto de este parámetro es ‘α’= 0,0001. En nuestro caso los hiperparámetros son ‘*alpha*’: $[0.00001, 0.0001, 0.001, 0.01, 0.00005, 0.0005, 0.005, 0.05]$

A pesar de que no sea modificado en nuestro modelo hace falta recalcar el parámetro de *activation*, ya que este representa, como ya he dicho anteriormente (Apartado 2.1.1 Perceptrón Multicapa), la función de activación de las neuronas. Este parámetro tiene por defecto la opción ‘*activation*’ = “relu”[28].

4.2.2. *Random Forest*

Para el *Random Forest* se utiliza la función "RandomForestClassifier" de "sklearn.ensemble" con un único parámetro fijo '*random_state*'= 23, como hemos hecho anteriormente, gracias a la función "sklearn.model_selection.GridSearchCV" hemos entrenado el *Random Forest* con una serie de hiperparámetros que son los siguientes:

- Número de estimadores ('n_estimators'): Número de árboles de decisión del *Random Forest*. El valor por defecto de este parámetro es '$n_estimators$'= 100. Nuestros hiperparámetros para el entrenamiento han sido '$n_estimators$': $[100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000]$.
- Función de medida de calidad ('criterion'): Es la función utilizada para medir la calidad de una división en función de las clases. Vamos a utilizar dos tipos:
  - Gini: El Indice Gini ha sido explicado en el apartado 2.1.2.2 Indice Gini.
  - Entropy: La entropia ha sido explicada en el apartado 2.1.2.1 Ganancia de Información.

  El valor por defecto de este parámetro es '*criterion*'="*gini*". De forma que nuestros hiperparámetros para el entrenamiento quedan '*criterion*': ["*gini*","*entropy*"].
- Profundidad máxima ('max_depth'): Es la profundidad máxima a la que puede llegar el árbol, es decir, un criterio de parada del algoritmo. El valor por defecto de este parámetro es '*max_depth*'="None", es decir, no existe límite de profundidad de árbol. En nuestro caso los hiperparámetros son '*max_depth*': $[10, 20, 40, 50, \text{"}None\text{"}]$.
- Número máximo de características ("max_features"): Es el número máximo de características que hay que considerar a la hora de buscar la mejor división de los datos. Hemos utilizado 2 tipos:
  - Sqrt: En el que $max_features = sqrt(n_features)$, siendo n_features el número de características totales.
  - Log2: En el que $max_features = log_2(n_features)$.

  El valor por defecto de este parámetro es 'max_features' = "sqrt"[29]. En nuestro caso los hiperparámetros son '$max_features$': ["*sqrt*","*log*2"].

4.2.3. *Ada Boost*

Para *Ada Boost* se utiliza la función "AdaBoostClassifier" de "sklearn.ensemble" con un único parámetro fijo '*random_state*'= 23, como hemos hecho anteriormente, gracias a la función "sklearn.model_selection.GridSearchCV" hemos entrenado *Ada Boost* con una serie de hiperparámetros que son los siguientes:

- Estimador base ('base_estimator'): Es el estimador base a partir del cual se construye el ensemble. El valor por defecto de este parámetro es 'base_estimator' = "None". En nuestro caso los estimadores base para *Ada Boost* van a ser *Decision Trees* (sus parámetos han sido explicados en el apartado 4.2.2 *Random Forest*), por lo que los hiperparámetros quedan como
  'base_estimator': [$DecisionTreeClassifier(criterion =$'*entropy*'$, max_depth = 5)$,
  $DecisionTreeClassifier(criterion =$'*entropy*'$, max_depth = 10)$,
  $DecisionTreeClassifier(criterion =$'*entropy*'$, max_depth = 15)$,

$DecisionTreeClassifier(criterion =\text{'}entropy\text{'}, max_depth = 20),$
$DecisionTreeClassifier(criterion =\text{'}entropy\text{'}, max_depth = 25),$
$DecisionTreeClassifier(criterion =\text{'}gini\text{'}, max_depth = 5),$
$DecisionTreeClassifier(criterion =\text{'}gini\text{'}, max_depth = 10),$
$DecisionTreeClassifier(criterion =\text{'}gini\text{'}, max_depth = 15),$
$DecisionTreeClassifier(criterion =\text{'}gini\text{'}, max_depth = 20),$
$DecisionTreeClassifier(criterion =\text{'}gini\text{'}, max_depth = 25)].$

- Número de estimadores ('n_estimators'): Número máximo de estimadores en los que finaliza el *Boosting*, en el caso en el que el ajuste sea perfecto y los ejemplos estén bien clasificados el algoritmo se detendrá antes de tiempo. El valor por defecto de este parámetro es 'n_estimators' = 50 [30]. Nuestros hiperparámetros para el entrenamiento son 'n_estimators' = $np.arange(1, 50)$, es decir, todos los valores entre el 1 y el 49 ambos incluidos.

4.2.4. Métrica de error para la clasificación

Para la medida de acierto y error de los 3 modelos que acabamos de presentar en la clasificación de imágenes hemos utilizado la métrica F1 *score*, la cual pertenece a la librería "sklearn.metrics".

En cuanto a la implementación, la función *f1_score* tiene los siguientes parámetros:

- y_true: Un *array* con las clases reales para cada una de las instancias del *dataset*
- y_pred: Un *array* con las clases predecidas para cada una de las instancias del *dataset*
- *average*: Es obligatorio para modelos multiclase como el nuestro. Determina que promedio se va a realizar con los datos. En nuestro caso hemos utilizado el promedio *macro* que calcula la métrica para cada clase y realiza después una media de la siguiente forma:

$$MacroF1_score = \frac{1}{N}\sum_{i=0}^{N} F1 - score_i \tag{25}$$

La función quedaría del modo siguiente:

$$F1_score = sklearn.metrics.f1_score(y_true, y_pred, average = \text{'}macro\text{'}) \cdot 100 \tag{26}$$

4.3. *Deep Learning*

4.4. YOLOv5

En cuanto a la implementación de la quinta versión de *You Only Look Once* hemos utilizado el modelo y repositorio creado por Ultralytics [31] que es un grupo de inteligencia artificial que desarrolla modelos y los hacen fácilmente accesibles y entrenables a todo el mundo. Este modelo está implementado con ayuda de la librería Pytorch [32], la cual es ampliamente utilizada en el campo de las redes neuronales convolucionales.

4.4.0.1. Formato del *dataset*

Antes de realizar el entrenamiento es necesario modificar el formato del *dataset* al formato *Roboflow*, de forma que tengamos un archivo de texto por cada una de las imágenes de entrenamiento (con el nombre de la imagen + .txt) en el cual se encuentren las coordenadas del centro de las *Bounding Boxes* y la altura y anchura de las mismas normalizadas (entre 0 y 1) (Figura 37)

Class	**x**	**y**	**width**	**height**
0	0.38192	0.58242	0.49011	0.40291
1	0.23022	0.44221	0.89331	0.33425
1	0.38702	0.53200	0.28081	0.40292
⋮				

Figura 37: Ejemplo de formato de datos para entrenamiento de YOLOv5

4.4.0.2. Entrenamiento de la red

En este caso el entrenamiento de la red con el *dataset* ya reformateado es muy sencillo ya que lo único que hay que hacer es modificar el numero de clases y los nombres de las mismas en un archivo de configuración y ejecutar en una terminal el comando

$$python\ train.py\ \text{--}img\ 640\ 360\ \text{--}rect\ \text{--}batch\ 16\ \text{--}epochs\ 100\ \text{--}data\ \text{`}../data_txt.yml\text{'}\ \text{--}cfg\ \text{`}models/custom_yolov5m.yaml\text{'}\ \text{--}name\ yolov5m_results \tag{27}$$

Que indica que se va a entrenar el modelo con imágenes de tamaño 640 x 360 píxeles, con un *batch size* o tamaño de lote (muestras procesadas antes de la actualización del modelo) de 16, 100 *epochs* o numero de épocas (numero de pases completos a través del conjunto de entrenamiento), con el --data se indica el archivo yaml del dataset, con el --cfg se indica cual de los tamaños de YOLO queremos usar para entrenar (*nano, small, medium, large, Xlarge*) y finalmente el --name indica la carpeta en la que guardan los resultados.

4.4.0.3. *Testing* de la red

En cuanto al testeo de la red se realiza con el comando

$$\begin{aligned} &\textit{python detect.py --weights ../weights/best.pt --img } 416 \textit{ --conf } 0{,}4 \\ &\qquad \textit{--source ../test} \end{aligned} \tag{28}$$

en el cual --weights indica la carpeta en la que se encuentran los pesos de la red ya entrenada, --img que indica de nuevo el tamaño de las imágenes, --conf que indica el umbral de confianza y --source que indica la carpeta en la que se encuentran las imágenes que queremos testear.

5. Resultados

5.1. Maching Learning

Ahora se van a proceder a exponer todos los resultados obtenidos en la parte de *Machine Learning* del proyecto.

5.1.1. Un modelo por cada *Bounding Box*

5.1.1.1. Clasificación

***Bounding Boxes* cuadradas (*Square*)**

En cuanto a los resultados de las *Bounding Boxes* con forma cuadrada se puede observar que el mejor modelo es el que utiliza un *Random Forest* con $max_depth = 10$, $n_estimators = 100$, $criterion =$'*gini*' y $random_state = 23$, con unos *F-scores macro* de 62.85 en *train*, 44.83 en *validation* y 43.83 en *test* (Tabla 1).

	Square		
	Train	Validation	Test
MLPClassifier	59.81	41.75	42.15
Random Forest	62.85	44.83	43.83
AdaBoost	33.1	31.99	31.61

Tabla 1: *F1-score macro* para *Bounding Box* Cuadradas

***Bounding Boxes* rectangulares horizontales (*Rectangle right*)**

Con respecto a los resultados de las *Bounding Boxes* con forma rectangular horizontal se puede observar que el mejor modelo es el que utiliza un *MLP Classifier* con $alpha = 1e{-}05$, $hidden_layer_sizes = (64, 32, 4)$, $max_iter = 1000$, $solver =$ '$lbfgs$' y $random_state = 23$, con unos *F-scores macro* de 88.93 en *train*, 44.72 en *validation* y 45.01 en *test*(Tabla 2).

	Rectangle Right		
	Train	Validation	Test
MLPClassifier	88.93	44.72	45.01
Random Forest	69.72	45.2	43.76
AdaBoost	38.38	37.39	35.8

Tabla 2: *F1-score macro* para *Bounding Box* Rectangulares horizontales

***Bounding Boxes* rectangulares verticales (*Rectangle Up*)**

En el caso de los resultados de las *Bounding Boxes* con forma rectangular vertical se puede observar que el mejor modelo es el que utiliza un *MLP Classifier* con $alpha = 1e{-}05$, $hidden_layer_sizes = (64, 32, 4)$, $max_iter = 1000$, $solver =$ '$lbfgs$' y $random_state = 23$, con unos *F-scores macro* de 85.53 en *train*, 36.75 en *validation* y 36.58 en *test* (Tabla 3).

	Rectangle Up		
	Train	Validation	Test
MLPClassifier	85.53	36.75	36.58
Random Forest	77.28	36.17	35.85
AdaBoost	29.7	28.16	28.64

Tabla 3: *F1-score macro* para *Bounding Box* Rectangulares verticales

Impresiones de la clasificación

En este apartado de clasificación se observa que se obtienen mejores resultados generalmente con los modelos que utilizan un *MLP Classifier* frente a los que utilizan un *Random Forest* (2 frente a 1), se observa también que los resultados de los modelos de *AdaBoost* no funcionan para nada correctamente.

5.1.1.2. Detección

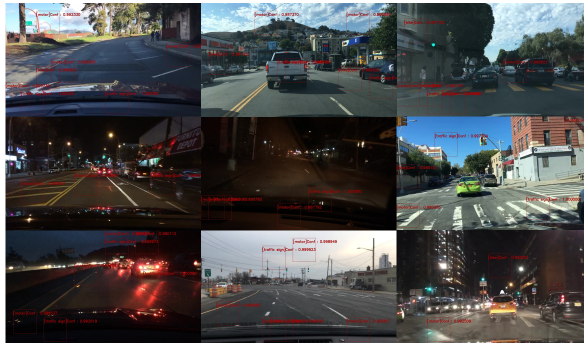


Figura 38: Ejemplo de falsos positivos en las detecciones del modelo con diferentes *Bounding Boxes*

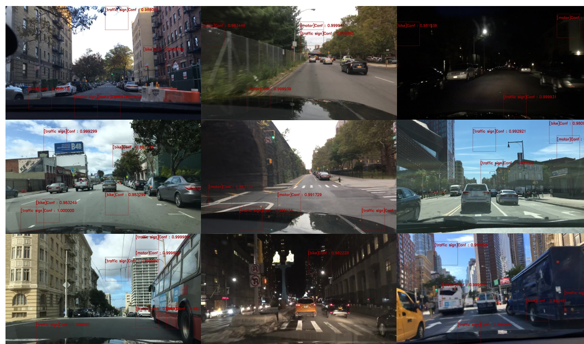


Figura 39: Otro ejemplo de falsos positivos en las detecciones del modelo con diferentes *Bounding Boxes*

Figura 40: Otro ejemplo de falsos positivos en las detecciones del modelo con diferentes *Bounding Boxes*

Tras la exposición de los resultados se puede observar que ya en el apartado de clasificación los modelos no generalizan bien, ya que, pese a que en algunos modelos el *F1-score macro* del apartado de *train* sea relativamente alto, al no generalizar bien el modelo, las pruebas en test solo obtienen en torno a un 30-40 de *F1-score macro*, viendo que además en el apartado de detección el mAP no es para nada alto (mAP = 0) y el modelo de detección de objetos no es para nada funcional, se decidió optar por una segunda metodología en la que se pueda conseguir mayor generalización en el apartado de clasificación y tal vez, un mAP superior y un modelo funcional, se opto entonces por realizar 6 modelos diferentes en los que cada uno clasifique una clase diferente.

5.1.2. Un modelo por cada clase

5.1.2.1. Clasificación

Modelo binario con la clase *Traffic sign*

En cuanto a los resultados del clasificador binario con las clases *'Traffic sign'* y 'otros' se puede observar que el mejor modelo es el que utiliza un *Random Forest* con $max_depth = 10$, $n_estimators = 850$, $criterion =$'*entropy*' y $random_state = 23$, con unos *F-scores macro* de 93.61 en *train*, 85.87 en *validation* y 85.33 en *test* (Tabla 4).

	Traffic sign		
	Train	Validation	Test
MLPClassifier	90.35	85.28	85.02
Random Forest	93.61	85.87	85.33
AdaBoost	86.17	83.54	85.04

Tabla 4: *F1-score macro* para modelo binario de *Traffic sign*

Modelo binario con la clase *Motor*

En lo que respecta a los resultados del clasificador binario con las clases *'Motor'* y 'otros' se puede observar que el mejor modelo utiliza un *MLP Classsifier* con $alpha = 1e-05$, $hidden_layer_sizes = (32, 32, 16, 4)$, $max_iter = 1000$, $solver =$'$lbfgs$'

y *random_state* = 23, con unos *F-scores macro* de 90.39 en *train*, 90.75 en *validation* y 91.02 en *test* (Tabla 5).

	Motor		
	Train	Validation	Test
MLPClassifier	90.39	90.75	91.02
Random Forest	99.64	90.42	90.53
AdaBoost	90.50	90.30	90.50

Tabla 5: *F1-score macro* para modelo binario de *Motor*

Modelo binario con la clase *Bike*

En el caso de los resultados del clasificador binario con las clases *'Bike'* y 'otros' se observa que el mejor modelo es el que utiliza un *Random Forest* con *criterion* ='*entropy*', *max_depth* = 40, *n_estimators* = 750 y *random_state* = 23, con unos *F-scores macro* de 99.26 en *train*, 83.51 en *validation* y 84.44 en *test* (Tabla 6).

	Bike		
	Train	Validation	Test
MLPClassifier	99.13	80.59	82.13
Random Forest	99.26	83.51	84.44
AdaBoost	95.94	79.00	80.08

Tabla 6: *F1-score macro* para modelo binario de *Bike*

Modelo binario con la clase *Car*

En cuanto a los resultados del clasificador binario con las clases *'Car'* y 'otros' se puede observar que el mejor modelo es el que utiliza un *Random Forest* con *criterion* ='*entropy*', *max_depth* = 40, *n_estimators* = 1000 y *random_state* = 23, con unos *F-scores macro* de 99.09 en *train*, 81.33 en *validation* y 81.16 en *test* (Tabla 7).

	Car		
	Train	Validation	Test
MLPClassifier	98.98	81.05	80.88
Random Forest	99.09	81.33	81.16
AdaBoost	93.15	79.45	78.27

Tabla 7: *F1-score macro* para modelo binario de *Car*

Modelo binario con la clase *Person*

En el caso de los resultados del clasificador binario con las clases *'Person'* y 'otros' se puede observar que el mejor modelo utiliza un *Random Forest* con *criterion* ='*entropy*', *max_depth* = 20, *n_estimators* = 1000 y *random_state* = 23, con unos *F-scores macro* de 99.12 en *train*, 84.95 en *validation* y 85.44 en *test* (Tabla 8).

	Person		
	Train	Validation	Test
MLPClassifier	99.02	82.89	82.00
Random Forest	99.12	84.95	85.44
AdaBoost	92.50	82.20	82.13

Tabla 8: *F1-score macro* para modelo binario de *Person*

Modelo binario con la clase *Rider*

En lo que respecta a los resultados del clasificador binario con las clases *'Rider'* y 'otros' se observa que el mejor modelo es el que utiliza un *Random Forest* con *criterion* ='*entropy*', *max_depth* = 40, *n_estimators* = 750 y *random_state* = 23, con unos *F-scores macro* de 99.50 en *train*, 87.02 en *validation* y 87.09 en *test* (Tabla 9).

	Rider		
	Train	Validation	Test
MLPClassifier	99.46	85.16	86.04
Random Forest	99.50	87.02	87.09
AdaBoost	94.75	84.36	83.83

Tabla 9: *F1-score macro* para modelo binario de *Rider*

Impresiones de la clasificación

En este apartado de clasificación se observa que se obtienen mejores resultados generalmente con los modelos que utilizan un *Random Forest* frente a los que utilizan un *MLP Classifier* (5 frente a 1, diferente que en el caso anterior), también de nuevo se observa que los resultados de los modelos de *AdaBoost* están por debajo del resto de modelos en todas las clases.

5.1.2.2. Detección

En cuanto a los resultados de la detección tenemos algunos casos en el que alguna *Bounding Box* se acerca o predice parcialmente la clase del objeto como en los siguientes

Figura 41: Ejemplo de coincidencia parcial de señal de tráfico a la izquierda de la imagen

Figura 42: Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen

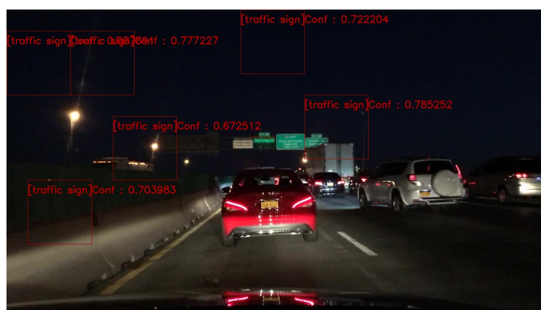


Figura 43: Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen

Figura 44: Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen

Figura 45: Ejemplo de coincidencia parcial de señal de tráfico a la izquierda de la imagen

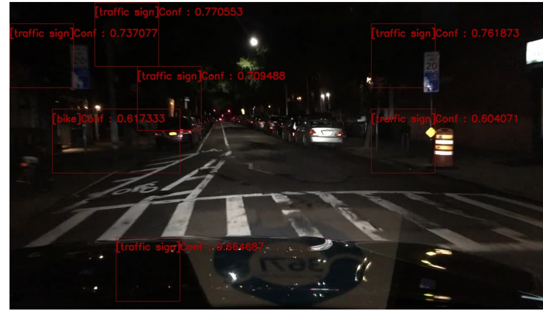


Figura 46: Ejemplo de coincidencia parcial de señal de tráfico a la derecha de la imagen

Figura 47: Ejemplo de coincidencia parcial de señal de tráfico a la izquierda de la imagen

Sin embargo viendo la cantidad de falsos positivos que hay principalmente con la clase señal es probable que estas detecciones sean meras coincidencias.

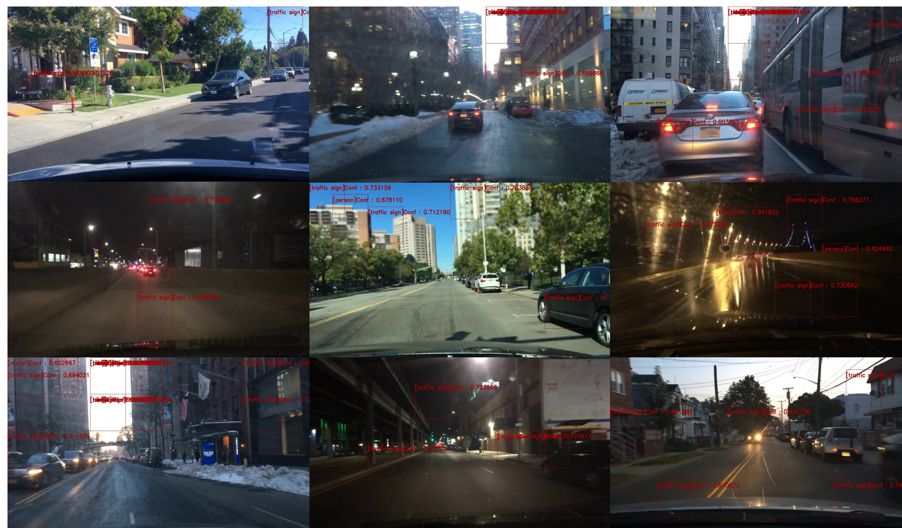


Figura 48: Ejemplo de falsos positivos en las detecciones

En esta segunda metodología se puede apreciar que los resultados en el apartado de clasificación son bastante más favorables, con *F1-score macro* de 90-100 en los apartados de *Train* y un 80-90 en los de *Test*. Por tanto utilizando un modelo por clase la generalización es mayor, sin embargo a pesar de esto en la parte de detección el mAP continua siendo muy bajo (mAP = 0) y, de nuevo, el modelo de detección no es nada funcional a pesar de que la diferencia de ambos modelos en el apartado de clasificación sea grande.

5.2. Deep Learning

Ahora se van a proceder a exponer todos los resultados obtenidos en la parte de *Deep Learning* del proyecto.

5.2.1. YOLO

Con el modelo de YOLO se han obtenido infinitamente mejores resultados que en la parte de *Machine Learning*. Vamos a comenzar con detecciones generales de YOLOv5 y proseguir con algunos errores o casos específicos.

Figura 49: Ejemplo de resultados generales de YOLOv5

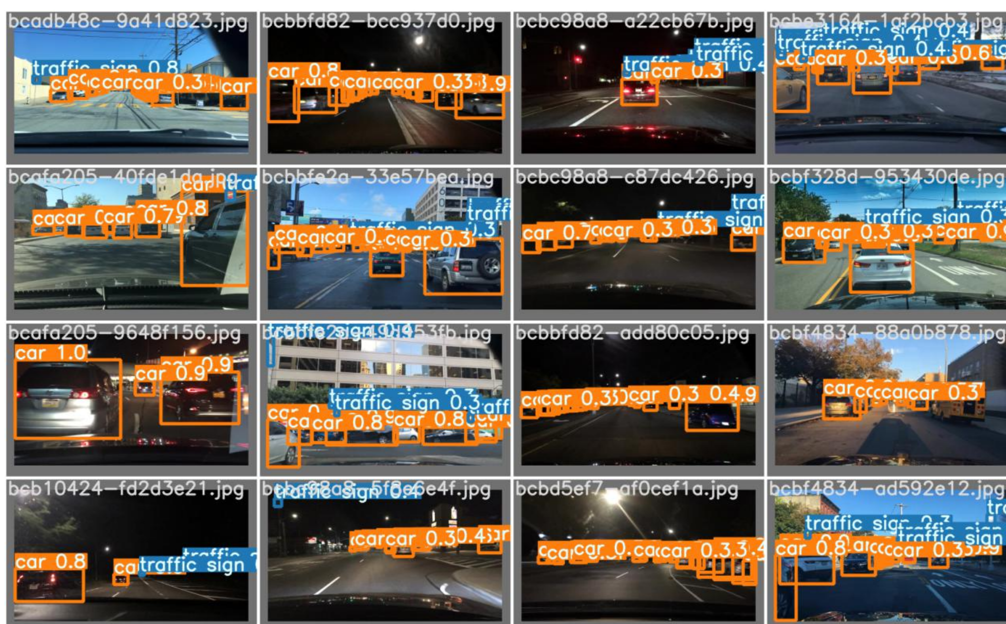


Figura 50: Otro ejemplo de resultados generales de YOLOv5

5.2.1.1. Casos específicos

Casos con diferentes niveles de luz

Figura 51: Ejemplo de resultados de noche de YOLOv5

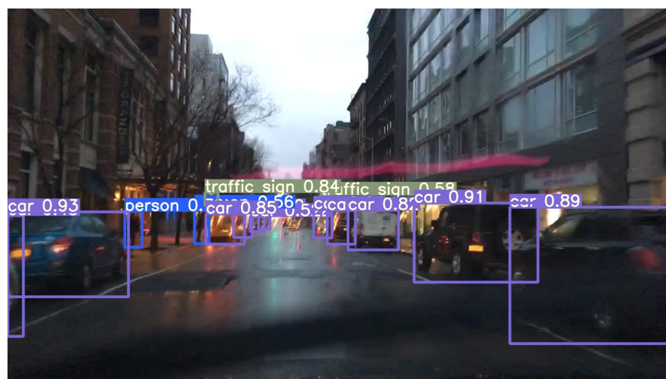


Figura 52: Ejemplo de resultados con cielo nublado de YOLOv5

Figura 53: Ejemplo de resultados con el cielo despejado de YOLOv5

Casos diferentes en detecciones de señales de tráfico

Figura 54: Ejemplo de no detección de señales de tráfico pese a su semejanza (carteles de la derecha) de YOLOv5

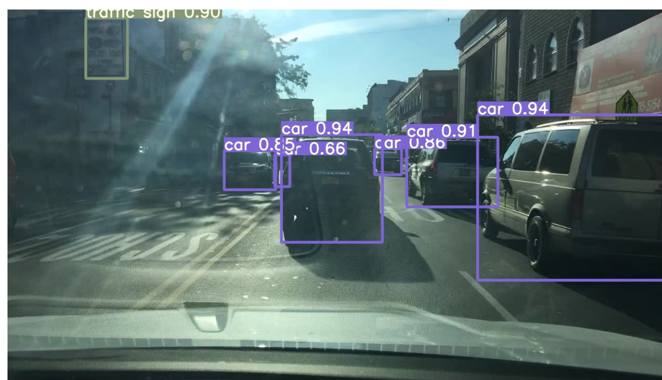


Figura 55: Ejemplo de detección de cartel publicitario como señal de tráfico de YOLOv5

Figura 56: Ejemplo de detección de señal de tráfico pese a la poca visibilidad (señales de la izquierda) de YOLOv5

Casos diferentes de fotografías mal enfocadas

Figura 57: Ejemplo de detección de dos señales (un falso positivo) y dos coches pese a la borrosidad de la imagen con YOLOv5

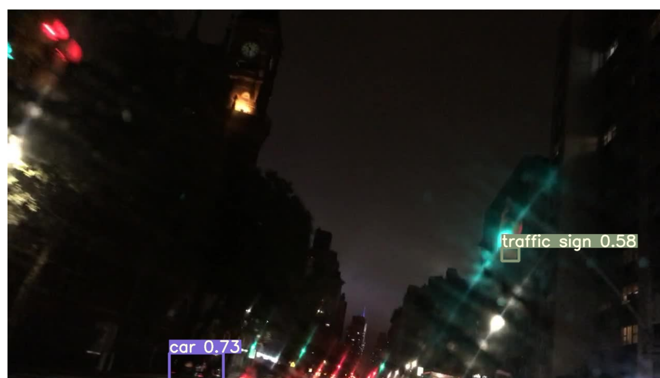


Figura 58: Ejemplo de detección de un coche y una señal de tráfico (falso positivo) pese a la borrosidad y ángulo de la imagen con YOLOv5

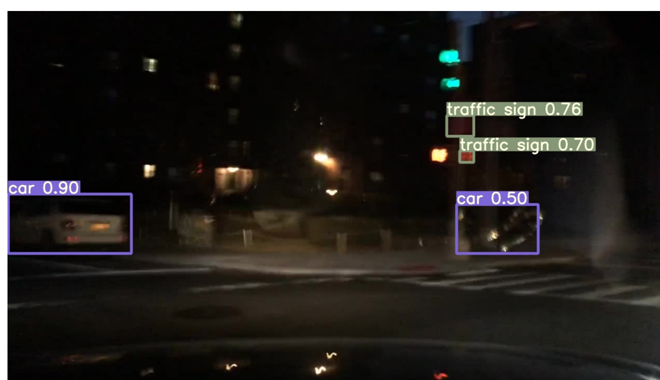


Figura 59: Ejemplo de detección de dos señales y dos coches (uno es un falso positivo) pese a la borrosidad de la imagen con YOLOv5

Casos de detecciones erroneas

Figura 60: Ejemplo de detección erronea de un coche (parte derecha de la imagen) con YOLOv5

Figura 61: Ejemplo de detección erronea de un coche en un cristal mojado con YOLOv5

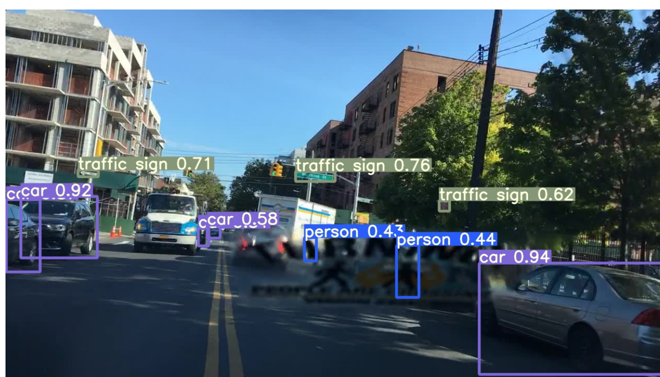


Figura 62: Ejemplo de detección erronea de dos personas en una pegatina de la luna con YOLOv5

Casos de detecciones precisas

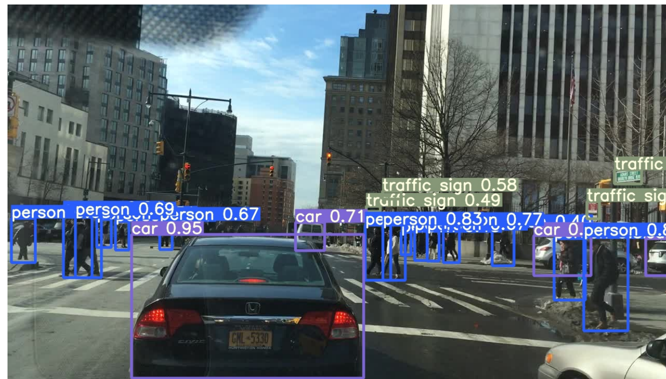


Figura 63: Ejemplo de detección precisa multiclase con YOLOv5

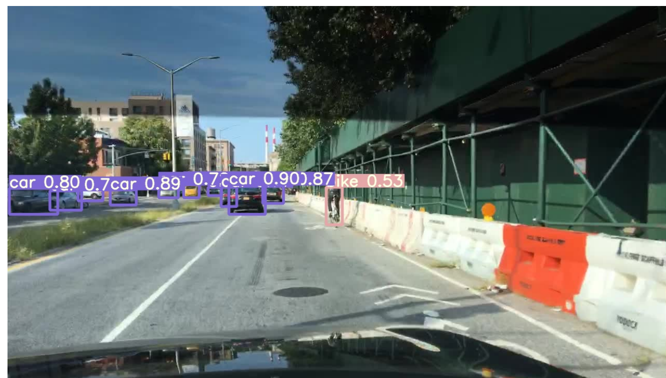


Figura 64: Ejemplo de detección precisa multiclase (con bicicleta) con YOLOv5

Figura 65: Ejemplo de detección precisa multiclase con YOLOv5

El modelo aplicado a un vídeo se puede visualizar en el siguiente link: `https://youtu.be/JYghe7F1N6A`.

En cuanto a resultados teóricos del modelo entrenado de YOLOv5 tenemos la matriz de confusión y la curva *Precision-Recall*. En lo que a la matriz de confusión se refiere (Figura 66) se puede apreciar que existe una gran diferencia entre las clases de, por un lado, '*Traffic sign*', '*Car*', '*Person*' con las clases '*Rider*', '*Bike*' y la clase '*Motor*', ya que se puede apreciar que en el caso de la clase '*Car*' el 53 % de las veces que el modelo predice coche está en lo cierto, mientras que en el 46 % de los casos lo predice como la clase '*Background*', sin embargo en el caso de las bicicletas este acierto disminuye mucho, ya que, solamente en el 13 % de las veces el modelo predice que es una bici cuando está en lo cierto, mientras que el resto de ocasiones lo predice como la clase '*Background*' (82 %), '*Motor*' (3 %) y '*Person*' (3 %), esta efectividad disminuye aun más en el caso de la clase '*Motor*', ya que el 0 % de las ocasiones el modelo predice bien las motos cuando se trata de una moto, mientras que el resto de las ocasiones lo predice como la clase '*Background*' (100 %). Esto puede deberse a la muy diversa cantidad de imágenes que hay de cada clase, ya que en el caso de las tres primeras clases ('*Traffic sign*', '*Car*' y '*Person*') el número de imágenes contenidas en el *dataset* es superior a 100.000 en los tres casos (274.591, 815.708, 104.609 respectivamente), sin embargo en el caso de las clases '*Rider*' y '*Bike*' el número de imágenes pasa por poco los 5.000 (5.166 y 8.217 respectivamente) y ya en el caso de la clase '*Motor*' el numero de imágenes supera por poco los 3.000 (3.454 imágenes) (El modelo de YOLO no ha sido entrenado con todas las imágenes sino con un porcentaje de cada clase, se mantiene aun así la gran diferencia entre instancias por cada clase), teniendo esto en cuenta es probable que el modelo no esté generalizando bien para las clases que tienen tan pocas instancias y sea por esto que esté fallando más, pasaría lo mismo en menor medida con las clases que más acierta como coches o señales de tráfico, ya que se podría aumentar la efectividad entrenando el modelo con más imágenes.

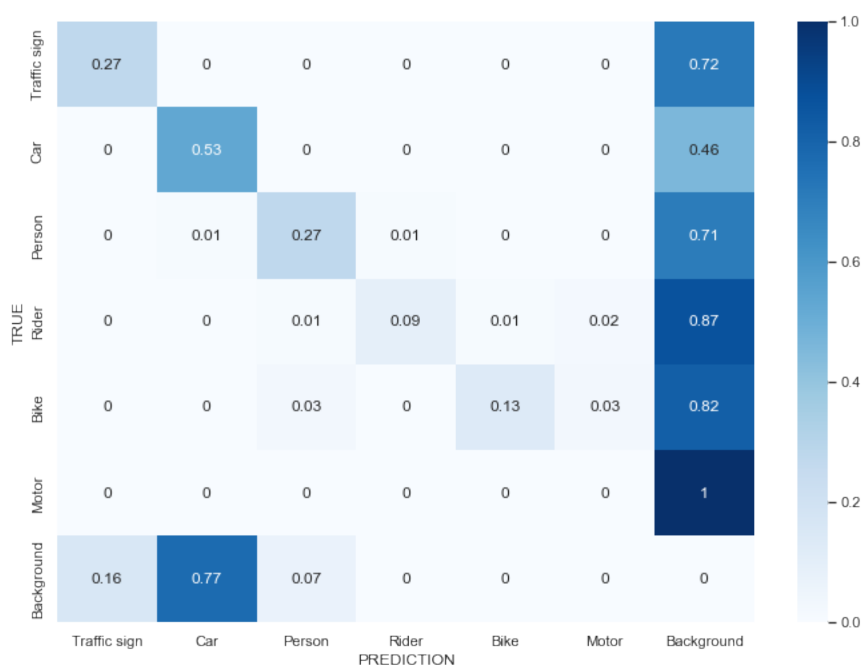


Figura 66: Matriz de confusión del modelo de YOLOv5

Esto lo apoya también la gráfica de *Precision-Recall* (Figura 67), ya que se puede ver (en la leyenda de la gráfica) que el AP (*Average Precision*) para cada clase sigue el orden mencionado anteriormente, en el primer caso el AP más alto lo tiene la clase *'car'* ($AP = 0{,}515$, clase con mayor número de imágenes), en el segundo caso la clase *'traffic sign'* ($AP = 0{,}263$), siguiendole de cerca la clase *'person'* ($AP = 0{,}247$), en cuarto lugar la clase *'rider'* ($AP = 0{,}139$), seguido de cerca por la clase *'bike'* ($AP = 0{,}129$) y finalizando con la clase *'motor'* ($AP = 0{,}070$), esto se ve también reflejado en los valores de *Precission* y *Recall* de la gráfica, ya que la *precision* se mantiene alta para la mayoría de las clases (menos para motor), sin embargo el *Recall* se mantiene en unos niveles medio bajos ya que mide la sensibilidad del modelo y en este caso el número de predicciones totales es considerablemente más grande que cada una de las predicciones positivas por clase. Teniendo todo esto en cuenta YOLOv5 para este *dataset* consigue un mAP de 0.227 en la detección de objetos.

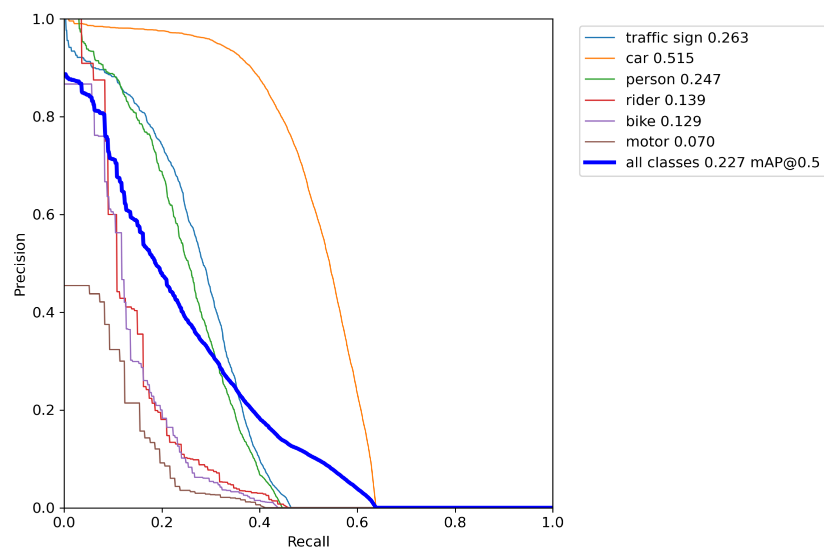


Figura 67: Curva de *Precision-Recall* de YOLOv5

6. Conclusiones

Finalmente, tras haber mostrado todo el trabajo que se ha realizado en la fase de *Machine learning*, tanto la clasificación como detección de objetos teniendo en cuenta distintas formas de *Bounding Boxes* y distintos modelos y tras mostrar además todo el trabajo realizado en la parte de *Deep Learning*, con el modelo de YOLO podemos concluir que el modelo de YOLO (*Deep Learning*) es, no solo más rápido, sino mucho más preciso y efectivo que los modelos utilizados en la parte de *Machine Learning*. Algunos problemas que han surgido son, en primer lugar la cantidad de tiempo que toman los entrenamientos de *Machine learning*, ya que al tener que realizar *Grid Searchs* hemos finalizado el proyecto entrenando un total de 32950 modelos de *Machine Learning*, lo cual ha tomado un tiempo de 931.20 horas, o lo que es lo mismo casi 39 días. Otro problema surgido durante el desarrollo del proyecto es la gran diferencia de documentación y código que hay entre YOLO y SSD, ya que en el caso de YOLO al ser un modelo más conocido hay todo tipo de explicaciones de como funciona y los repositorios que lo albergan tienen mantenimiento y son versiones finalizadas, sin embargo en SSD, al ser menos conocido, la cantidad de información y explicaciones es mucho más reducida al igual que los repositorios, muchos que aun están en proceso y algunos finalizados pero para nada funcionales, debido a esto la comparación con el modelo SSD se ha añadido a trabajos futuros en lugar de realizar la comparación aquí.

7. Lineas futuras

En cuanto a lineas futuras sería interesante aumentar el *dataset* para algunas clases, ya que por ejemplo la clase ‘car’ tiene muchas más instancias que la clase ‘motor’, por tanto seria posible añadir instancias al *dataset* para realizar de nuevo los entrenamientos y así tener la posibilidad de obtener mejores resultados, también sería conveniente aumentar el número de instancias de la clase ‘otros’, ya que en los resultados de la parte de *Machine Learning* se puede apreciar que hay muchos falsos positivos en zonas como el cielo o diferentes edificios, de forma que aumentando las instancias de esta clase en lugares específicos en los que el modelo falle como el cielo, la carretera o los edificios puede ser posible la reducción de estos falsos positivos. También se podrían obtener buenos resultados aplicando la pirámide gaussiana a las *Bounding Boxes* de la parte de *Machine Learning*, de forma que sea más sencillo detectar tanto objetos pequeños como grandes. En cuanto a la parte de *Deep Learning* queda pendiente la utilización o implementación del modelo de SSD que ha sido explicado en este proyecto. Existen otras opciones como la del *transfer learning*, para así utilizar parte de una arquitectura de una red preentrenada y adaptarla a mis clases. Para aumentar la calidad de las detecciones a la hora de utilizar un modelo para la detección de objetos en vídeos (*frame* a *frame*) se podría utilizar un algoritmo de *tracking*.

Referencias

[1] Joseph Redmon et al. «You Only Look Once: Unified, Real-Time Object Detection». En: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2016-December (jun. de 2015), págs. 779-788. ISSN: 10636919. DOI: 10.48550/arxiv.1506.02640. URL: https://arxiv.org/abs/1506.02640v5.

[2] Wei Liu et al. «SSD: Single shot multibox detector». En: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9905 LNCS (2016), págs. 21-37. ISSN: 16113349. DOI: 10.1007/978-3-319-46448-0{_}2/FIGURES/5. URL: https://link.springer.com/chapter/10.1007/978-3-319-46448-0_2.

[3] Samuel Arthur et al. «Some studies in machine learning using the game of checkers». En: *IBM Journal of research and development* 3.3 (1959), págs. 210-229.

[4] Robert E. Schapire. «The strength of weak learnability». En: *Machine Learning 1990 5:2* 5.2 (jun. de 1990), págs. 197-227. ISSN: 1573-0565. DOI: 10.1007/BF00116037. URL: https://link.springer.com/article/10.1007/BF00116037.

[5] F. Rosenblatt. «The perceptron: A probabilistic model for information storage and organization in the brain». En: *Psychological Review* 65.6 (nov. de 1958), págs. 386-408. ISSN: 0033295X. DOI: 10.1037/H0042519. URL: /record/1959-09865-001.

[6] David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams. «Learning representations by back-propagating errors». En: *Nature 1986 323:6088* 323.6088 (1986), págs. 533-536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0.

[7] Yoav Freund, Robert E Schapire et al. «Experiments with a new boosting algorithm». En: *icml*. Vol. 96. 1996, págs. 148-156.

[8] *A Guide To Understanding AdaBoost — Paperspace Blog*. URL: https://blog.paperspace.com/adaboost-optimizer/.

[9] R K McConnell. *Method of and apparatus for pattern recognition*. Ene. de 1986. URL: https://www.osti.gov/biblio/6007283.

[10] Satya Mallick. *Histogram of Oriented Gradients explained using OpenCV*. URL: https://learnopencv.com/histogram-of-oriented-gradients/.

[11] Yann LeCun, Yoshua Bengio y Geoffrey Hinton. «DeepLearning». En: (2015). DOI: 10.1038/nature14539.

[12] Ji Hyun Park et al. «Rgb image prioritization using convolutional neural network on a microprocessor for nanosatellites». En: *Remote Sensing* 12.23 (dic. de 2020), págs. 1-22. ISSN: 20724292. DOI: 10.3390/RS12233941.

[13] Joseph Redmon y Ali Farhadi. *YOLO9000: Better, Faster, Stronger*. Inf. téc. URL: http://pjreddie.com/yolo9000/.

[14] Joseph Redmon y Ali Farhadi. *YOLOv3: An Incremental Improvement*. Inf. téc. URL: https://pjreddie.com/yolo/..

[15] Alexey Bochkovskiy, Chien-Yao Wang y Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. Inf. téc. URL: https://github.com/AlexeyAB/darknet.

[16] *GitHub - ultralytics/yolov5: YOLOv5 in PyTorch ¿ONNX ¿CoreML ¿TFLite*. URL: https://github.com/ultralytics/yolov5.

[17] *GitHub - meituan/YOLOv6: YOLOv6: a single-stage object detection framework dedicated to industrial applications.* URL: https://github.com/meituan/YOLOv6.

[18] *GitHub - WongKinYiu/yolov7: Implementation of paper - YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*. URL: https://github.com/WongKinYiu/yolov7.

[19] Xiang Long et al. *PP-YOLO: An Effective and Efficient Implementation of Object Detector*. Inf. téc. URL: https://github.com/PaddlePaddle/.

[20] Chien-Yao Wang y Hong-Yuan Mark Liao. *Scaled-YOLOv4: Scaling Cross Stage Partial Network*. Inf. téc.

[21] Xin Huang et al. *PP-YOLOv2: A Practical Object Detector*. Inf. téc. URL: https://arxiv.org/abs/2104.10419.

[22] *Your Comprehensive Guide to the YOLO Family of Models*. URL: https://blog.roboflow.com/guide-to-yolo-models/.

[23] Karen Simonyan y Andrew Zisserman. «VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION». En: (2015). URL: http://www.robots.ox.ac.uk/.

[24] *VGG-16 — CNN model - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/vgg-16-cnn-model/.

[25] Jia Deng Jonathan Krause Michael Stark y Li Fei-Fei. «3D Object Representations for Fine-Grained Categorization». En: *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*. 2013.

[26] Alexander Masalov et al. «Specialized cyclist detection dataset: Challenging real-world computer vision dataset for cyclist detection using a monocular RGB camera». En: *IEEE Intelligent Vehicles Symposium, Proceedings* 2019-June (jun. de 2019), págs. 114-118. DOI: 10.1109/IVS.2019.8813814.

[27] Yu & Fisher & Chen & Haofeng & Wang & Xin & Xian & Wenqi & Chen & Yingying & Liu & Fangchen & Madhavan & Vashisht & Darrell & Trevor. «BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning». En: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020. URL: https://bdd-data.berkeley.edu/.

[28] *sklearn.neural_network.MLPClassifier — scikit-learn 1.1.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[29] *sklearn.ensemble.RandomForestClassifier — scikit-learn 1.1.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[30] *sklearn.ensemble.AdaBoostClassifier — scikit-learn 1.1.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html.

[31] *YOLOv5 and Vision AI - Ultralytics*. URL: https://ultralytics.com/.

[32] *PyTorch*. URL: https://pytorch.org/.