Técnicas de estadística computacional para visión por computador

escrito por

JULIÁN BLASCO HERREIZ

Director: Rafael Marcos Luque Baena
Co-Director: Alberto Borobia Vizmanos

Facultad de Ciencias
UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Trabajo presentado para la obtención del título de
Máster Universitario en Matemáticas Avanzadas de la UNED.
Especialidad Estadística e Investigación Operativa

JULIO 2018

ABSTRACT

Abstract en español:

A lo largo de este trabajo Fin de Máster se repasarán las actuales tendencias en análisis de imagen mediante técnicas de Redes Neuronales Convolucionales y sus aplicaciones en vigilancia, conducción autónoma y procesado de imagen.

Finalmente se incluirá un ejemplo práctico llevado a cabo mediante el proyecto público Mask_RCNN que permite adaptar la versión base a cualquier conjunto de imágenes para ser entrenada y verificada. La salida de esta red neuronal convolucional se aplicará, junto con diferentes algoritmos, para el recuento de objetos y se estudiarán aproximaciones de cálculo basadas en técnicas estocásticas.

Abstract in English:

This Master Thesis will review the state of the art of algorithms and Convolucional Neural Networks used for image processing, surveillance and autonomous driving.

Finally, a practical example will be studied using public Mask_RCNN that allows to be adapted easily to any image dataset for training and verification. The output from the neural network will be applied to object detection and count, using for this purpose different algorithms and modifications based on stochastic calculus.

Keywords: Mask_RCNN, image, Machine Learning, MLP, Kalman Filter

Dedicatoria y agradecimientos

The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer.

Alan Turing. Computing Machinery and Intelligence (1950)

Dedicado a todos aquellos que, antes que yo, sentaron las bases para que la nueva revolución digital vea la luz.

Este trabajo no hubiera visto la luz sin el apoyo que me han otorgado así como la paciencia en mis largas horas trabajando en él.

Tabla de Contenidos

Página

Índice de tablas viii

Índice de figuras ix

1 Introducción 1

2 Aprendizaje Automático 3

2.1. Tipos de algoritmos 3

2.1.1. Aprendizaje Supervisado 4

2.1.2. Aprendizaje no Supervisado 4

2.1.3. Aprendizaje profundo 6

2.2. Enfoques 7

2.2.1. Árboles de decisión 7

2.2.2. Reglas de asociación 8

2.2.3. Algoritmos Genéticos 9

2.2.4. Redes Neuronales Artificiales 9

2.2.5. Máquina de Vectores de Soporte 10

2.2.6. Algoritmo de Agrupamiento 11

2.2.7. Redes Bayesianas 11

2.3. Software de uso común para aplicaciones de Aprendizaje Automático 12

2.3.1. Python 12

2.3.2. TensorFlow 12

2.3.3. CUDA nVIDIA 13

2.3.4. Keras 13

2.3.5. Otros 13

3 Redes Neuronales Artificiales 15

3.1. Breve historia de las Redes Neuronales 16

3.2. Fundamentos de las Redes Neuronales 17

3.2.1. La neurona desde un punto de vista biológico 17

3.2.2. La neurona desde un punto de vista artificial 18
3.3. Entrenamiento de una red neuronal 19
3.4. Perpectrón Multicapa 20
3.4.1. Entrenamiento del MLP 20
3.4.2. Aplicaciones del MLP 21
3.4.3. Limitaciones del MLP 22
3.5. Algoritmo *Backpropagation* 23
3.5.1. Función de activación 26
3.6. Pasos futuros para el MLP 27

4 Redes Neuronales Convolucionales 29
4.1. Arquitectura de las CNN 31
4.2. Funcionamiento de una CNN 33
4.2.1. Estructura 33
4.3. Conceptos adicionales en CNN 39
4.3.1. Rellenado y deslizamiento 39
4.3.2. Clasificación, localización, detección y segmentación 41
4.4. Desarrollos de visión computerizada y CNN 41
4.4.1. AlexNet (2012) 41
4.4.2. ZF NET (2013) 43
4.4.3. VGG Net (2014) 44
4.4.4. GoogLeNet (2015) 45
4.4.5. Microsoft ResNet (2015) 46
4.4.6. CNN basadas en Regiones 46
4.4.7. *Generative Adversarial Networks* (2014) 53
4.4.8. Generación de descripción de imágenes (2014) 53
4.4.9. Redes de transformación espacial (2015) 54

5 Desarrollo aplicado de aprendizaje automático 57
5.1. Mask R-CNN para detección de objetos y segmentación 58
5.1.1. Detección paso a paso 59
5.1.2. Entrenamiento de Mask R-CNN 62
5.2. Aplicación de Mask RCNN 64
5.2.1. Revisión del conjunto de datos 64
5.2.2. Entrenamiento de los pesos 71
5.2.3. Revisión del modelo generado 73
5.2.4. Revisión de los pesos 82
5.3. Filtro de Kalman 83
5.3.1. Implementación del Filtro de Kalman en Python 87

5.4. Aplicación 88
5.4.1. Preparación de la etapa de procesado 88
5.4.2. Recuento de objetos detectados 93

6 Conclusiones **123**

A COCO Tools **125**
A.1. Categorías COCO 128
A.2. Métricas usadas en la detección COCO 130
A.2.1. Intersección sobre Unión 130
A.2.2. Precisión promediada 131
A.2.3. AP escalada 131
A.2.4. Sensibilidad promediada 131
A.2.5. AR escalada 132
A.2.6. Cálculo de las métricas 132
A.3. Uso de la biblioteca COCO 133

B Guía instalación de las herramientas empleadas en la memoria **135**
B.1. Python 135
B.1.1. Anaconda 136
B.2. CUDA 136
B.2.1. nVIDIA cuDNN 137
B.3. TensorFlow 138
B.4. Resto de módulos necesarios 138
B.5. Free Video to JPG Converter 139
B.6. OpenShot Video Converter 140

C Convolución **141**
C.1. Propiedades 141
C.2. Ejemplos de aplicación del operador convolución 143

D Código fuente **145**
D.1. Inspect Data 145
D.2. Entrenamiento de los pesos en terminal Anaconda 152
D.3. Procesado sobre la imagen 159
D.4. Preparación de vídeos de CVPR 2018 WAD 161
D.5. Procesado de video 164
D.6. Postrocesado: extracción de información 173
D.7. Postrocesado: discriminación de nuevos objetos mediante posición 175

D.8. Postprocesado: discriminación de nuevos objetos mediante trayectoria y Filtro de Kalman 176
D.9. Postprocesado: eliminación de objetos espurios 178

Bibliografía **179**

Índice de tablas

Tabla **Página**

1.1. Evolución temporal del aprendizaje automatizado 2

3.1. Algoritmos de aprendizaje más conocidos 21
3.2. Evaluación de la precisión en GPU de MLP y diversas CNN 27

5.1. Comparativa de algoritmos de recuento de objetos 108
5.2. Comparativa de algoritmos con Filtro de Kalman y diferentes límites de proximidad de trayectoria 111
5.3. Comparativa final de clases detectadas 115

Índice de figuras

FIGURA	Página
2.1. Matriz de distancias	5
2.2. Dendogramas de agrupamiento jerárquico	5
2.3. Ejemplo de árbol de decisión	8
3.1. Esquema de una neurona biológica	16
3.2. Neurona e impulsos eléctricos en su periferia	18
3.3. Esquema de una neurona artificial	18
3.4. Perceptrón MultiCapa	22
3.5. Perceptrón MultiCapa n:m:1	22
3.6. Perceptrón multicapa genérico	23
4.1. Corteza visual primaria	29
4.2. Cortex visual	30
4.3. Mapa de activación de una Red Neuronal Convolucional	31
4.4. Vista general de una Red Neuronal Convolucional	32
4.5. Desglose de capas de una CNN y aplicación a una imagen	32
4.6. Procesado en una Red Neuronal Convolucional	33
4.7. Primera capa de una Red Neuronal Convolucional	34
4.8. Filtro para detección de curvas	35
4.9. Imagen a la que se aplica el filtro convolucional	35
4.10. Convolución de la curva con el filtro	36
4.11. Convolución de otra zona con el filtro	36
4.12. Fragmentación y superposición de una imagen antes de un filtro convolucional	37
4.13. Red Neuronal Convolucional completa	38
4.14. Pooling de 2x2	39
4.15. Delizamiento de una unidad	40
4.16. Delizamiento de dos unidades	40
4.17. Rellenado de volúmenes	40
4.18. Clasificación, segmentado y detección en una imagen	41
4.19. Arquitectura AlexNet	42

4.20. Arquitectura ZF Net 43
4.21. Arquitectura VGG Net 44
4.22. Flujo de trabajo de una R-CNN 47
4.23. Arquitectura de una R-CNN 47
4.24. Flujo de trabajo de una *Fast* R-CNN 48
4.25. Arquitectura de una Fast R-CNN 49
4.26. Flujo de trabajo de una *Faster* R-CNN 49
4.27. Arquitectura de *Faster* R-CNN 50
4.28. Entrenamiento de una Region Proposal Network 51
4.29. Comparativa entre redes R-CNN 51
4.30. Salida de *Mask* R-CNN 52
4.31. Flujo de trabajo de una GAN 53
4.32. Detección de imágenes con descripción 54
4.33. Módulo de transformación espacial 55

5.1. Ejemplo de resultado Mask R-CNN 58
5.2. Mask R-CNN: filtrado 59
5.3. Mask R-CNN: refinado 60
5.4. Mask R-CNN: máscaras 60
5.5. Mask R-CNN: activación 61
5.6. Mask R-CNN: histogramas 61
5.7. Mask R-CNN: Tensorboard 61
5.8. Mask R-CNN: resultado final 62
5.9. Comparación entre RCNN pre-entrenada y custom 63
5.10. Imagen de COCO y sus máscaras 65
5.11. Imagen de COCO con máscaras y cajas en objetos 66
5.12. Anclaje del objeto en una imagen de COCO 68
5.13. Anclajes positivos, negativos y neutros 69
5.14. ROIs de una imagen COCO 70
5.15. Imagen original y máscaras para la comprobación del modelo 71
5.16. Detecciones con pesos originales y entrenados 73
5.17. Imagen original y detectada 74
5.18. Curva de sensibilidad y precisión para los pesos calculados 75
5.19. Tabla de predicciones para los pesos calculados 76
5.20. Anclajes positivos 79
5.21. Propuestas finales de objetos detectados 80
5.22. Propuestas finales de objetos etiquetados 81
5.23. Propuestas finales de máscaras sobre objetos 82
5.24. Imagen de referencia del vídeo procesado 89

5.25. Imagen con clases recortadas . . . 91
5.26. Imagen con objetos con nivel de aceptación bajo . . . 92
5.27. Imagen final con etiquetas porcentuales . . . 92
5.28. Secuencia de 3 imágenes con un objeto perdido . . . 95
5.29. Recuento de objetos: algoritmo inicial . . . 97
5.30. Error de conteo debido a objetos contados por duplicado . . . 98
5.31. Recuento de objetos: algoritmo mejorado con detección de posición . . . 100
5.32. Error de conteo debido a objetos contados por duplicado solucionado mediante cálculo de la posición . . . 101
5.33. Recuento de objetos: algoritmo mejorado con Filtro de Kalman . . . 105
5.34. Error de conteo debido a objetos contados por duplicado solucionado mediante cálculo de la trayectoria . . . 106
5.35. Resultados finales para los tres algoritmos de recuento . . . 110
5.36. Detalle de objeto espurio . . . 112
5.37. Probabilidad eliminar detección de objeto espurio . . . 113
5.38. Eficacia eliminar detección de objeto espurio . . . 114
5.39. Recuento de objetos: algoritmo final . . . 115
5.40. Vídeo 00, valores finales . . . 116
5.41. Vídeo 01, valores finales . . . 117
5.42. Vídeo 02, valores finales . . . 117
5.43. Vídeo 03, valores finales . . . 118
5.44. Vídeo 04, valores finales . . . 118
5.45. Vídeo 05, valores finales . . . 119
5.46. Vídeo 06, valores finales . . . 119
5.47. Vídeo 07, valores finales . . . 120
5.48. Vídeo 08, valores finales . . . 120
5.49. Vídeo 09, valores finales . . . 121
5.50. Vídeo 10, valores finales . . . 121
5.51. Vídeo 11, valores finales . . . 122

A.1. Ejemplo de imagen en COCO . . . 126
A.2. Imagen “COCO 37777” del conjunto de validación de 2017 . . . 126
A.3. Imagen “COCO 37777” con metadatos y máscaras . . . 127
A.4. Imagen “COCO 37777” con etiquetas . . . 128
A.5. Categorías de COCO . . . 129
A.6. Ejemplo de IoU . . . 130
A.7. Cálculo de IoU . . . 131

B.1. Detalle de modelo de gráfica usada en la memoria para el procesado de imágenes . . . 137

B.2. Contenido .zip cuDNN 137
B.3. Consola principal *Free Video to JPG Converter* 139
B.4. Consola principal *OpenShot Video Converter* 140

C.1. Convolución de $u(t) * e^{-t}$ 143

CAPÍTULO 1

INTRODUCCIÓN

Actualmente vivimos en un mundo en el que la cantidad de información generada crece exponencialmente. Si nos fijamos en los siglos pasados, esta información ya fuera en forma de texto o imágenes, era fácilmente manejable por las personas, sin embargo hoy por hoy dada la magnitud del volumen se hace necesario utilizar elementos electrónicos que, entrenados adecuadamente, puedan procesar la información para destacar los aspectos relevantes y actuar en consideración. Para poder procesar e identificar toda esta información los entrenamientos a los que se someten los elementos electrónicos (ordenadores principalmente) basan toda su algoritmia en desarrollos matemáticos enfocados al aprendizaje automático o "Machine Learning".

Llamamos aprendizaje automático o aprendizaje de máquinas al subcampo de las ciencias de la computación, así como una rama de la inteligencia artificial, que tiene como objetivo el desarrollo de técnicas para permitir a las máquinas aprender. De una manera más exacta, se trata de la generación de programas capaces de lograr una generalización de comportamientos a partir de información dada como ejemplos. Es, por tanto, un proceso de inducción del conocimiento. Este campo de actuación del aprendizaje automático se solapa normalmente con el de la estadística computacional, ambos basados en el análisis de datos. Por contra, el aprendizaje automático también se centra en la parte de la complejidad computacional de los problemas. Muchos de estos problemas son de clase NP-hard[1], por lo que el diseño de soluciones factibles a esos problemas recoge gran parte de la investigación enfocada en aprendizaje automático. Por último, el aprendizaje automático puede ser visto como un intento de lograr la automatización, mediante métodos matemáticos, de algunas partes del método científico.

Esta rama desarrollada y comenzada durante el siglo pasado, que se arrastra hasta nuestros

[1]Al menos tan complejo como un problema NP, pero no necesariamente NP

Década	Resumen
<1950s	Descubrimiento y refinamiento de los métodos estadísticos
1950s	Usando algoritmos simples se investigan de forma pionera los algoritmos de aprendizaje automático
1960s	Se introduced los métodos Bayesianos para inferencia de probabilidad en el aprendizaje automatizado
1970s	El pesimismo sobre la eficiencia del aprendizaje automatizado causa el "Invierno de la I.A.[1]"
1980s	El redescubrimiento del algoritmo de la retroalimentación ayuda al resurgir del desarrollo del aprendizaje automatizado
1990s	El trabajo con respecto al aprendizaje automatizado cambia de plano desde una aproximación basada en el conocimiento hacia una basada en los datos. Se comienzan a desarrollar programas para ordenadores a fin de analizar grandes cantidades de datos y extraer conclusiones de los resultados.
2000s	Los métodos Kernel crecen en popularidad y el aprendizaje automatizados se vuelve más popular
2010s	El aprendizaje profundo ("Deep Learning") se vuelve factible, el cual conduce a que el aprendizaje automatizado pueda ser usado ampliamente en servicios software y aplicaciones

TABLA 1.1. Evolución temporal del aprendizaje automatizado[2]

días, sigue creciendo de forma rápida alimentada principalmente por el exponencial desarrollo de los elementos físicos encargados de potenciarlas como son los ordenadores y sus diferentes elementos (a destacar el uso intensivo de las GPUs para estas labores).

Siendo esta rama de la matemática computacional una de las que más rápido está creciendo, es de esperar que al término de esta memoria ya existan nuevos desarrollos que hagan necesario incluir más puntos a revisar para tener una visión actual del problema asociado al aprendizaje automatizado.

A lo largo de la presente memoria se hará una breve descripción de los enfoques referentes al aprendizaje automático hasta llegar al *core* de este trabajo, las Redes Neuronales Convolucionales, en adelante CNN[2], y sus diferentes variantes o mejoras.

Finalmente se aplicarán los conceptos y técnicas expuestas en un problema concreto, como es la detección y recuento de objetos en imágenes. Este problema práctico irá explorando casos reales tomados mediante cámaras de tráfico o en vehículos para hacer diversas aproximaciones de algoritmos. A su vez, estos algoritmos se compararán entre ellos para solventar por un lado problemas reales de objetos detectados de forma errónea debido a espurios u objetos que se cuentan de forma incorrecta por haber desaparecido de la detección. Para ello se utilizarán diversas aproximaciones como es el cálculo de una órbita de proximidad entre series temporales de objetos o la inferencia de trayectorias mediante Filtro de Kalman.

[2]Por sus siglas en inglés *Convolutional Neural Network*

Capítulo 2

Aprendizaje Automático

El aprendizaje automático es la rama de la Inteligencia Artificial que tiene como principal objetivo el desarrollo de técnicas que permitan aprender a las computadoras[3]. De una manera más exacta, se trata de la generación de programas capaces de generalizar comportamientos a partir de información dada como ejemplos.

Esta inducción se considera completa cuando se han observado todos los casos particulares, de manera que se considera válida esta la generalización a la que ha dado lugar. Sin embargo, en la mayoría de los casos es imposible obtener una inducción completa, por lo que el enunciado queda sometido a un cierto grado de incertidumbre de manera que no se puede considerar como un esquema de inferencia formalmente válido ni se puede encontrar una justificación empírica. Frecuentemente el campo de actuación del aprendizaje automático se solapa con el de *Data Mining*, ya que las dos disciplinas están enfocadas en el análisis de datos, sin embargo el aprendizaje automático se centra más en el estudio de la complejidad computacional de los problemas para así hacerlos factibles desde el punto de vista práctico, no únicamente teórico.

2.1. Tipos de algoritmos

Los diferentes algoritmos de aprendizaje automático se clasifican en función de la salida de los mismos. Dos de los más importantes, y los que repasaremos a continuación, serían el aprendizaje supervisado y el aprendizaje no supervisado. Además de estas dos clasificaciones de algoritmos tendríamos otros, como por ejemplo, los aprendizajes por refuerzo, transducción o multi-tarea que estarían ya fuera del alcance de esta memoria.

2.1.1. Aprendizaje Supervisado

El Aprendizaje Supervisado es una técnica para inferir una función a partir de datos obtenidos de entrenamientos. Los datos utilizados para el entrenamiento consisten en vectores. Por un lado, un componente del par son los datos de entrada y por otro lado, los resultados esperados. La salida de la función puede ser tanto un valor numérico (vectorial o simple) o una etiqueta, que se corresponderían con los problemas de regresión y clasificación respectivamente. El objetivo del aprendizaje es, a partir de la observación de un número dado de ejemplos, crear una función que haga una predicción sobre el valor correspondiente a esos datos de entrenamiento. De esta manera, ha de ser capaz de generalizar a partir de nuevos datos de información no incluidos en los vectores de entrenamiento.

Así pues, los datos que constituyen el entrenamiento están formados por varios pares de patrones de entrada y salida. Al conocerse la salida, el sistema se beneficia de la supervisión por parte de un maestro, de ahí que se le llame Aprendizaje Supervisado. Para un nuevo patrón de entrenamiento, en la etapa $(m+1)_{ésima}$ los pesos se adaptarán de la siguiente manera:

$$w_{ij}^{m+1} = w_{ij}^{m} + \Delta w_{ij}^{m}$$

Dentro de los algoritmos para el aprendizaje supervisado, uno de los que más importancia histórica y de uso tiene es el algoritmo de Propagación hacia Atrás *"Backpropagation"* que será descrito en el capítulo dedicado al Perceptrón MultiCapa *MLP*[1].

2.1.2. Aprendizaje no Supervisado

A diferencia del caso anterior, no existe un conocimiento a priori de las características de los conjuntos de datos. De esta manera, el Aprendizaje no Supervisado[4][5] trata los objetos de entrada como un conjunto desconocido de variables aleatorias construyendo en este caso un modelo de densidad para el conjunto de los datos.

El *Santo Grial* del aprendizaje no supervisado es la creación de un código factorial de los datos, esto es, un código con componentes estadísticamente independientes. El aprendizaje no supervisado produce mejores resultados cuando los datos iniciales son traducidos a un código factorial.

Ejemplos típicos son la Regla de Aprendizaje de Hebb, y la Regla de Aprendizaje Competitivo. Un ejemplo del primero consiste en reforzar el peso que conecta dos nodos que se excitan simultáneamente.

Otra forma de aprendizaje no supervisado es la agrupación (en inglés, clustering), el cual no tiene porqué estar basado en distribuciones de probabilidad. En el capítulo 12 de [6] se puede encontrar la descripción de varios algoritmos de agrupación y ordenación mediante aprendizaje no supervisado tanto con métodos jerárquicos como no jerárquicos. Los métodos de agrupación

[1]Por sus siglas en inglés MultiLayer Perceptron

jerárquicos buscan construir árboles de agrupaciones en función de las distancias entre los elementos. El siguiente ejemplo incluido en [6] ilustra de forma muy sencilla en qué consisten estos métodos.

Sea la siguiente matriz de distancias:

$$\mathbf{D} = \{d_{ik}\} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & & & & \\ 9 & 0 & & & \\ 3 & 7 & 0 & & \\ 6 & 5 & 9 & 0 & \\ 11 & 10 & \textcircled{2} & 8 & 0 \end{bmatrix} \end{array}$$

FIGURA 2.1. Matriz de distancias

Aplicando los algoritmos de enlace simple y completo, podemos dibujar los dendogramas de la agrupación de elementos con respecto a sus distancias mínimas y máximas respectivamente.

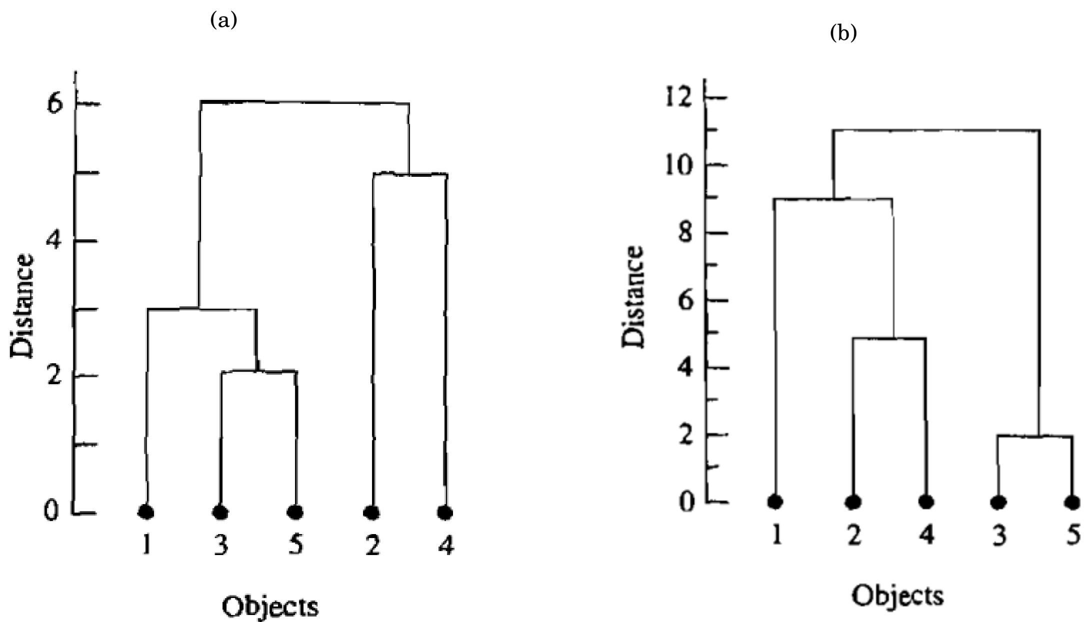


FIGURA 2.2. Dendogramas resultados del agrupamiento de enlace simple y completo, respectivamente

Como se puede observar en la Figura 2.2, los dos métodos producen resultados similares aunque variando la jerarquía de la agrupación, ya que uno funciona por distancias máximas y el

otro por distancias mínimas. Los elementos 1 y 3,5 en ambos algoritmos están a un solo salto uno de otro, pero la distancia que los une es de 3 u 11 unidades dependiendo de un algoritmo u otro.

Por otro lado existen también los algoritmos de agrupamiento no jerárquicos. En este caso no se busca encontrar las relaciones entre los diferentes miembros de un conjunto a ordenar, sino más bien, ser capaces de agruparlos en diferentes *clusters*. Uno de los algoritmos utilizados para este agrupamiento fue propuesto por MacQueen[8] que sugirió el término *K-means* para la descripción del mismo de forma que asigna cada elemento a ordenar dentro del grupo de forma que la distancia al centroide sea mínima.

Este algoritmo de puede resumir de manera muy concisa en los siguientes tres pasos:

1. Se asignan inicialmente los elementos en los K *clusters* y se calculan los centroides para cada uno de los *clusters*.

2. Se repasa la lista de elementos asignando cada uno al *cluster* que tenga el centroide más cercano. Normalmente esta distancia se calcula mediante la distancia euclídea ya sea con observaciones estandarizadas o no estandarizadas. Se recalcula el centroide del *cluster* que recibe el nuevo elemento así como el que lo pierde.

3. Se repite el paso 2. hasta que no hay más reasignaciones posibles.

Una variante consiste en que se asignan los K centroides al inicio y directamente se podría pasar al paso 2. Como se puede observar es un algoritmo muy sencillo y fácilmente programable. En el capítulo 12.4 de [6] se pueden encontrar múltiples ejemplos prácticos de este algoritmo.

En el aprendizaje competitivo, si un patrón nuevo pertenece a una clase reconocida previamente, entonces la inclusión de este nuevo patrón a la clase matizará la representación de la misma. Si el nuevo patrón no pertenece a ninguna de las clases reconocidas anteriormente, entonces la estructura y los pesos de la red neuronal serán ajustados a fin de poder reconocer la nueva clase[7].

2.1.3. Aprendizaje profundo

Se llama Aprendizaje Profundo [2] al conjunto de algoritmos que pretenden modelar abstracciones de alto nivel en datos mediante estructuras compuestas de transformaciones no lineales múltiples.

Cada una de las observaciones puede ser representada de diversas maneras, pero algunas de ellas hacen más sencillo aprender tareas de interés sobre los ejemplos y la investigación de este campo intenta definir cuáles son mejores y cómo crear modelos para reconocerlas.

Enfocando el campo de visión por computador, que es el que a esta memoria más le interesa, se pueden encontrar diferentes estructuras de aprendizaje profundo (redes neuronales profundas y convolucionales). En estas aplicaciones, dichas redes han mostrado resultados de vanguardia.

[2] En inglés Deep Learning

De manera general se trata de un conjunto de algoritmos generados para el aprendizaje automático. Partiendo de este punto común, existen diferentes características, como por ejemplo:

- Usar una estructura de capas en cascada con unidades no lineales para tanto extraer como transformar variables. Cada una de las capas usa la salida de la capa inmediata anterior como entrada. Los algoritmos usados pueden utilizar tanto aprendizaje supervisado como no supervisado. Las aplicaciones típicas incluyen tanto reconocimiento de patrones como modelización de datos.

- Estar basados en el aprendizaje de representaciones de datos o múltiples niveles de características. La representación jerárquica se forma derivando las características de más alto de nivel a partir de las de más bajo.

- Aprender varios niveles de representación en función de diferentes niveles de abstracción. Dichos niveles se ordenan formando una jerarquía de conceptos.

Todas estas formas de definición del aprendizaje profundo tienen en común:

- Existen múltiples capas no lineales para el procesamiento.

- El aprendizaje, tanto supervisado como no supervisado, de representaciones de características en cada capa.

2.2. Enfoques

A continuación presentamos una lista con los enfoques más representativos del Aprendizaje Automático de manera que se podrá construir la base para ir detallando, en los siguientes capítulos, la base matemática y práctica de las RCNN[3].

2.2.1. Árboles de decisión

Llamamos árbol de decisión a un modelo de predicción empleado en sectores que van desde la inteligencia artificial hasta incluso la economía. Para un conjunto de datos se construyen diagramas de conexiones lógicas, como se haría en los sistemas de predicción basados en reglas, siendo usados para representar y clasificar diversas condiciones que se presentan de manera sucesiva en la búsqueda de la solución de un problema.

Una de las áreas en la que los árboles de decisión son aplicados de manera intensiva es la Teoría de Juegos, que a su vez se emplea en campos que van desde la economía hasta las ciencias de la computación pasando y sin olvidarnos de la política, biología, sociología... En la Teoría de Juegos, estos árboles de decisión se emplean en la llamada forma extensiva, modelando los juegos que presentan algún orden o secuenciación. También es posible que estos juegos de forma

[3]Por sus siglas en inglés Region Convolutional Neural Network

extensiva modelen a su vez juegos simultáneos, siendo representados con una línea extensiva entre los nodos de elección de cada jugador, sin saber de antemano qué opción ha escogido su contrincante. En este tipo de juegos se puede utilizar como ejemplo el clásico "piedra, papel o tijera"[9].

Por ejemplo, el árbol de decisión de un juego muy simple en el cual en función de la apuesta de cada uno de los dos jugadores se obtiene su ganancia o pérdida podría ser el esquema propuesto en la Figura 2.3.

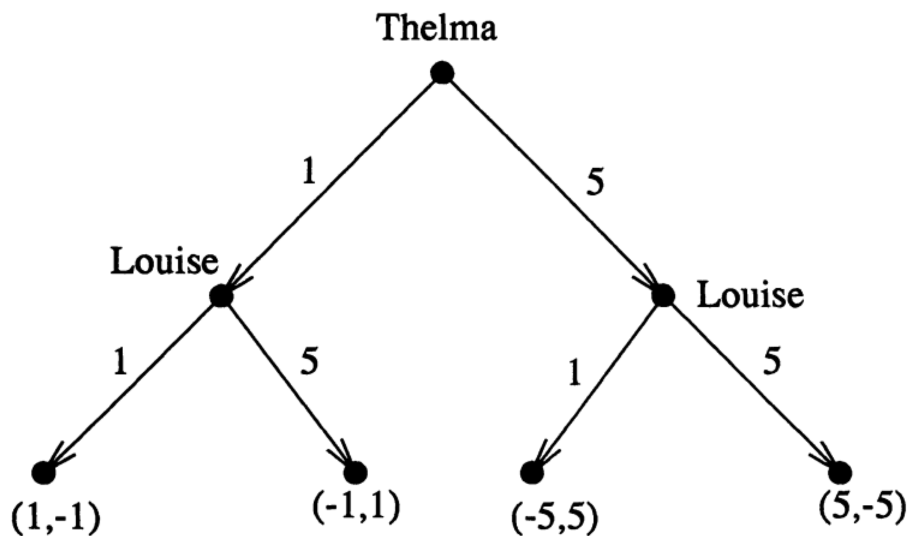


FIGURA 2.3. Árbol de decisión para el juego de *Los Chinos* simplificado

Este juego es una variante de uno bastante popular en mi zona de nacimiento llamado *Los Chinos*. En este caso cada jugador guarda en su mano un número de piezas o monedas que hay que adivinar. El juego simplificado elimina la parte de adivinación y simplemente se gana o pierde en función del valor de la moneda guardada en la mano[10].

2.2.2. Reglas de asociación

Las reglas de asociación se emplean, normalmente en minería de datos y aprendizaje automático, para encontrar pequeños hechos que se manifiestan en común dentro de un conjunto de datos dado. Diferentes métodos de aprendizaje de reglas de asociación han sido estudiados mostrando unos resultados muy interesantes para encontrar las relaciones entre variables cuando se está trabajando con grandes conjuntos de datos.

Piatetsky-Shapiro[11] muestra el análisis y la presentación de reglas “fuertes” encontradas en bases de datos mediante diferentes medidas de interés. Tomando el concepto de regla fuerte, Agrawal et al.[12] mostraron un trabajo en el que exponían las reglas de asociación que detallaban

las relaciones entre los datos recogidos a gran escala en los lineales de unos supermercados. Como por ejemplo la siguiente regla:

$$\{cebollas, vegetales\} \rightarrow \{carne\}$$

Si la anterior asociación se tomase de los datos de ventas de un supermercado, mostraría que un comprador que adquiere cebollas y verdura de forma simultánea, es posible que compre también carne. Dicha información se puede emplear como punto de partida para fijar decisiones sobre ofertas para ciertos productos o ubicaciones en los lineales. Además del ejemplo aplicado al análisis en supermercados, hoy en día, las reglas de asociación también se aplican en otras muchas áreas como el *Web mining*, detección de intrusos o bioinformática.

2.2.3. Algoritmos Genéticos

Durante la década de los 70 en el siglo XX, gracias a John Henry Holland, se desarrolló una de las líneas de trabajo más prometedoras de la inteligencia artificial, los algoritmos genéticos. Tienen este nombre debido a su inspiración en la evolución biológica así como su base genético-molecular.

Estos algoritmos, al igual que el famoso Juego de la vida, hacen evolucionar una población de individuos sometiéndola a diferentes acciones aleatorias similares a las que se manifiestan en la evolución biológica, así como también a una selección de acuerdo con diferentes, en función de los cuáles se decide quiénes son los individuos más adaptados, que sobrevivirían en este caso, y cuáles los menos aptos, que serían descartados.

Los algoritmos genéticos se engloban dentro de los algoritmos evolutivos, incluyendo también las estrategias evolutivas así como la programación evolutiva y genética.

Los algoritmos genéticos han demostrado su eficacia en caso de querer calcular funciones no derivables aunque su uso sería posible con cualquier función con independencia de la complejidad de su derivada.

Deben tenerse en cuenta las siguientes consideraciones:

- El número de iteraciones para asegurar el punto crítico global (máximo o mínimo) va depender de forma directa el número de puntos críticos locales en la función a optimizar.
- Si la función a optimizar contiene diversos puntos con valores próximos al óptimo, sólamente podemos asegurar que se encontrará uno de ellos y no será necesariamente el óptimo.

2.2.4. Redes Neuronales Artificiales

Las redes neuronales artificiales son un modelo computaciones elaborado de manera similar al comportamiento de los axones de las neuronas en los cerebros biológicos, esto es, basado en grandes estructuras de unidades neuronales simples o neuronas artificiales. Cada una de estas unidades está conectada a muchas otras y los enlaces entre ellas pueden modificar el

estado de activación ya sea incrementando o incluso anulándolo. Cada neurona trabaja utilizando únicamente operaciones suma y a la salida podría haber una función que fije un umbral en cada conexión de manera que hay que superarse un límite antes de propagar la información al resto de neuronas. Estos sistemas son capaces de aprender y formarse a sí mismos en lugar de funcionar debido a una programación fija, resultando muy útiles debido a su forma de funcionar en áreas donde el cálculo de soluciones es complicado de encontrar con programación convencional.

Las redes neuronales están formadas normalmente por varias capas o un diseño tridimensional en forma de cubo, siendo la propagación de la señal de adelante hacia atrás. Para el ajuste de los pesos de cálculo se utiliza la programación hacia atrás junto con los resultados esperados para así calcular el incremento de dichos pesos. Las redes modernas son más complejas en las que la información tiende a fluir de una manera más caótica y no únicamente en una dirección lineal.

El principal objetivo de una red neuronal es calcular soluciones a los problemas de una manera similar al cerebro humano aunque trabajando de forma más abstracta. La diversidad de proyectos de trabajo con redes neuronales abarcan configuraciones de unas miles de neuronas a millones de ellas con sus consiguientes conexiones que, en cualquier caso, siguen siendo más simples que el cerebro humano. Parece anecdótico que se asemejen más a la potencia de cálculo de un gusano. Las nuevas investigaciones sobre el cerebro humano estimulan nuevos patrones para la configuración y dimensión de las redes neuronales.

Un punto a destacar de estos sistemas es que el autoaprendizaje no garantiza un éxito seguro. Tras entrenar la red, algunos sistemas son capaces de resolver problemas mientras que otros no funcionan adecuadamente. A fin de capacitarlos se emplean varios ciclos de iteraciones que podrían llegar hasta miles.

Las redes neuronales se han empleado en la resolución de problemas que van desde la visión por computador hasta el reconocimiento de voz, siendo normalmente complicados de resolver usando la clásica programación basada en reglas.

Debido al incremento de potencia de cálculo gracias al uso de las GPU y computación distribuida las redes neuronales han vuelto a desplegarse a gran escala, principalmente en problemas de imagen y reconocimiento visual. Esto se conoce como aprendizaje profundo, a pesar de que el aprendizaje profundo no es siempre sinónimo de redes neuronales profundas.

Las redes neuronales artificiales se estudiarán en detalle en un capítulo dedicado por su importancia en el desarrollo de la parte aplicada de esta memoria.

2.2.5. Máquina de Vectores de Soporte

Las máquinas de soporte vectorial, también llamadas máquinas de vectores de soporte (SVMs[4]) son un conjunto de algoritmos de aprendizaje automático y supervisados desarrollados por el equipo de Vladimir Vapnik en los laboratorios AT&T.

[4]Por sus siglas en inglés Support Vector Machines

Estos métodos están directamente relacionados con problemas de clasificación y regresión. Formalmente hablando, una SVM construye un hiperplano o conjunto de ellos en un espacio con un número muy alto de dimensiones (o incluso infinita) que puede ser usado en los problemas antes descritos de clasificación o regresión. En función de la separación entre las clases se permitirá una mejor o peor clasificación.

2.2.6. Algoritmo de Agrupamiento

Un algoritmo de agrupamiento es un conjunto de reglas de cálculo enfocadas a la agrupación de una muestra de vectores de acuerdo a uno, o varios, criterios. Dichos criterios suelen ser distancia o similitud. La proximidad se define en función de una determinada relación de distancia, como podría ser la euclídea o distancia del taxi. La medida más usada para medir la similitud es la matriz de correlación entre los NxN casos. Sin embargo, también existen otros algoritmos que buscan maximizar la verosimilitud.

Normalmente los vectores de un mismo conjunto comparten propiedades. El grado de conocimiento sobre esos grupos permite una descripción sintética del conjunto de datos en varias dimensiones, por ello se emplean en minería de datos. Esta descripción atómica se consigue sustituyendo la descripción de la totalidad de los elementos por la de un representante canónico del grupo.

Es cierto que en algunos contextos, minería de datos por ejemplo, al algoritmo de agrupamiento se lo considera una técnica con entrenamiento no supervisado (2.1.2) dado que busca encontrar relaciones entre variables pero no las que guardan con un objetivo final.

2.2.7. Redes Bayesianas

Una red bayesiana, también llamada red de Bayes, es un modelo-grafo probabilístico que representa a un conjunto de variables aleatorias así como sus dependencias condicionales mediante un grafo acíclico dirigido o DAG[5]. Por ejemplo, una red bayesiana podría mostrar las relaciones probabilísticas entre enfermedades y síntomas. Por tanto, dado un conjunto de síntomas, la red podría ser usada para calcular la probabilidad de que el paciente sufra diversas enfermedades.

Existen diversos algoritmos que calculan la inferencia y el aprendizaje en redes bayesianas de una manera eficiente. A las redes bayesianas usadas para modelar secuencias de variables (como por ejemplo en señales del habla o secuenciación de proteínas) son llamadas redes bayesianas dinámicas. Por último, se llama diagramas de influcencia a las generalizaciones de estas redes que son capaces de representar y resolver problemas de decisión incluso bajo incertidumbre.

[5]Por sus siglas en inglés Directed Acyclic Graph

2.3. Software de uso común para aplicaciones de Aprendizaje Automático

Por último en este capítulo se revisarán las diferentes opciones existentes para implementar algoritmos de aprendizaje automático. Cabe destacar que esta lista, debido al gran incremento y mejoras de este campo, puede que ya esté obsoleta a la hora de finalizar la memoria.

Al final de la memoria, en el ANEXO B, se incluirá el listado e instrucciones de instalación de toda la suite necesaria para ejecutar los ejemplos y el código práctico del trabajo.

2.3.1. Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible[13].

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada *Python Software Foundation License*, que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.

Para más información o descarga se puede consultar https://www.python.org/

2.3.1.1. Anaconda

Anaconda es un distribución de los lenguajes Python y R libre y abierto, utilizada en ciencia de datos, y aprendizaje automático (machine learning). Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos). Está orientado a simplificar el despliegue y administración de los paquetes de software. Las diferentes versiones de los paquetes se administran mediante el sistema de administración del paquete conda, el cual lo hace bastante sencillo de instalar, ejecutar y actualizar software de ciencia de datos y aprendizaje automático como podría ser Scikit-team, TensorFlow y SciPy. La distribución Anaconda está utilizada por 6 millones de usuarios e incluye más de 250 paquetes válidos para Windows, Linux y MacOS[14].

Para más información o descarga se puede consultar https://anaconda.org/

2.3.2. TensorFlow

TensorFlow es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. Actualmente es utilizado tanto para la

investigación como para la producción de productos de Google frecuentemente remplazando el rol de su predecesor de código cerrado, DistBelief. TensorFlow fue originalmente desarrollado por el equipo de Google Brain para uso interno en Google antes de ser publicado bajo la licencia de código abierto Apache 2.0 el 9 de noviembre de 2015[15].

Para más información o descarga se puede consultar https://www.tensorflow.org/

2.3.3. CUDA nVIDIA

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema[16].

Gracias a millones de GPUs CUDA vendidas hasta la fecha, miles de desarrolladores, científicos e investigadores están encontrando innumerables aplicaciones prácticas para esta tecnología en campos como el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de la dinámica de fluidos, la reconstrucción de imágenes de TC, el análisis sísmico o el trazado de rayos, entre otras.

Los sistemas informáticos están pasando de realizar el procesamiento central en la CPU a realizar coprocesamiento repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las GPUs GeForce, ION Quadro y Tesla GPUs, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones.

Para más información o descarga se puede consultar
http://www.nvidia.es/object/cuda-parallel-computing-es.html

2.3.4. Keras

Keras es una librería de Python que proporciona de manera limpia y sencilla la creación de una gama de modelos de Deep Learning encima de otras librerías TensorFlow, Theano o CNTK. Keras fue desarrollado y es mantenido por François Chollet, un ingeniero de Google, y su código ha sido liberado bajo la licencia permisiva del MIT[17].

La estructura de datos principal en Keras es un modelo (model), que está pensada para facilitar la organización de capas, pues en realidad los modelos en Keras son básicamente una secuencia de capas.

Para más información o descarga se puede consultar https://keras.io/

2.3.5. Otros

Existen luego ya muchos más paquetes *menores* que han de ser instalados y puestos a punto para comenzar a trabajar, sin embargo debido a que son de menos importancia computacional no merecen un apunte diferenciado. Por ejemplo, *Jupyter Notebook* es de mucha utilidad para tener

scripts de Python con mucha legibilidad, sin embargo no constituyen una capa importante para el entrenamiento y posterior procesado de imágenes.

Una breve lista, no conclusiva, sería:

- Jupyter Notebook
- Numpy
- skimage
- scipy
- cv2

En cualquier caso, lo más recomendable es, al ir avanzando con la descripción de paquetes importados en el fichero de Python, instalar cualquier faltante para tener siempre versiones adecuadas y actualizadas.

CAPÍTULO 3

REDES NEURONALES ARTIFICIALES

Las redes neuronales artificiales son un modelo computaciones elaborado de manera similar al comportamiento de los axones de las neuronas en los cerebros biológicos, esto es, basado en grandes estructuras de unidades neuronales simples o neuronas artificiales. Cada una de estas unidades está conectada a muchas otras y los enlaces entre ellas pueden modificar el estado de activación ya sea incrementando o incluso anulándolo. Cada neurona trabaja utilizando únicamente operaciones suma y a la salida podría haber una función que fije un umbral en cada conexión, de manera que se debe superar un valor antes de propagar la información al resto de neuronas. Estos sistemas son capaces de aprender y formarse a sí mismos en lugar de funcionar debido a una programación fija, resultando muy útiles gracias a su forma de funcionar en áreas donde el cálculo de soluciones es complicado de encontrar con programación convencional.

La manera en la que operan es radicalmente diferente a la empleada por las estructuras de cálculo convencionales. Cada uno de los procesadores del cerebro (neuronas) trabajan de manera paralela. A diferencia de procesadores típicos no ejecutan un programa típico sino que transmiten señales de a través de los transmisores o sinapsis llegando a los centros de las neuronas y saliendo a su vez señales eléctricas a través de los canales de transmión o axones.

La importancia de cada una de las sinapsis en todo el proceso se actualiza de manera continua así como diferentes propiedades de las neuronas, de forma que se consigue un sistema de autoprogramación siendo además adaptable y por tanto, capaz de sustituir la programación externa. De esta manera se puede considerar que los programas y datos varían durante todo el tiempo.

Debido a que los cerebros no poseen una arquitectura o mapa de conexiones específico pueden operar de forma concurrente dedicando un gran número de neuronas a cada actividad. Esto da lugar a una gran cantidad de actividades diversas y complejas así como una gran adaptación

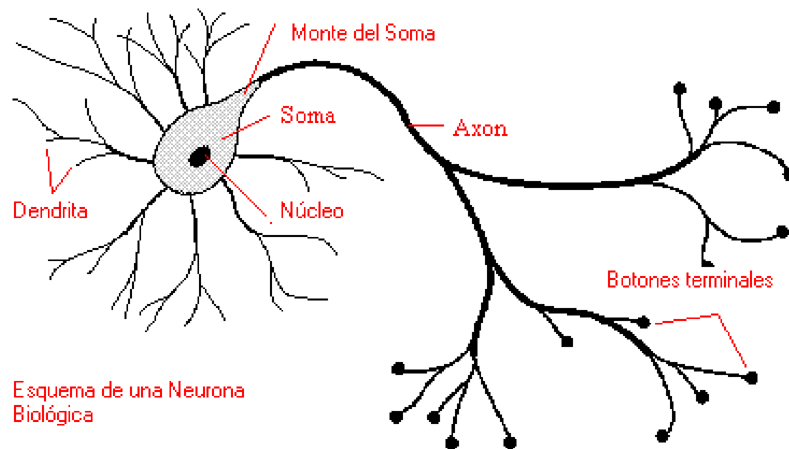


FIGURA 3.1. Esquema de una neurona biológica

al cambio, reconocer objetos deformados, difusos o no totalmente visibles. En resumen, como se indica en [18]: las redes neuronales son esencialmente diferentes de los sistemas de cómputo de la vida cotidiana.

3.1. Breve historia de las Redes Neuronales

La definición formal de neuronal en 1943, gracias a McCulloch y Pitts, se da como una máquina binaria con varias entradas y salidas. Esto fue el punto de partida para los modelos conexionistas. Años más tarde, en 1949, Hebb definió dos conceptos clave basándose en investigaciones psicofisiológicas, marcando así el campo de las redes neuronales[19]:

- El aprendizaje se localiza en las sinapsis o conexiones entre las neuronas.
- La información se representan en el cerebro mediante un conjunto de neuronas activas o inactivas.

Estas hipótesis se pueden condensar en la regla de aprendizaje de Hebb, que incluso se emplea en los modelos actuales. Dicha regla dice que los cambios que ocurren en los pesos de las sinapsis son generados por la interacción entre las neuronas pre y post sinapsis. Nos tenemos que remontar hasta 1956 para encontrar la primera conferencia sobre IA y que además se discutiera sobre la posibilidad de simular aprendizaje en computadoras. A partir de ese punto se han desarrollado diferentes tipos de redes neuronales.

En 1958 apareció el perceptrón de la mano de Frank Rosenblat. Se trataba de un algoritmo de reconocimiento de patrones que trabajaba mediante una red de aprendizaje usando dos capas que ejecutaban simples sumas y restas. Con notación matemática Rosenblat describió circuitería

en dicho perceptrón, como puede ser la XOR u OR-exclusiva[1]. Dicha función lógica no se pudo procesar hasta mucho tiempo después, ya en el año 1975 mediante el algoritmo de propagación hacia atrás desarrollado por Paul Werbos. Dicho algoritmo fue un avance clave ya que permitía resolver de manera eficaz el problema y de una manera más general, la formación de redes neuronales de múltiples capas.

Todo el conjunto de investigación en redes neuronales sufrió un estancamiento tras la publicación del trabajo de aprendizaje automático de Marvin Minsky y Seymour Papert en el año 1969. Dicha investigación descubrió dos puntos fundamentales de las máquinas que procesan las redes neuronales:

- Los perceptrones clásicos son incapaces de procesar la función XOR

- Los ordenadores no tienen la suficiente potencia de cálculo para manejar los tiempos de ejecución necesarios debido a la complejidad de las redes neuronales de gran tamaño

En torno a 1985 surgió el nombre de *conexionismo*, que dio lugar a la popularidad del procesamiento distribuido en paralelo. David E. Rumelhart y James McClelland publicaron en 1986 un volumen que incluye una detallada exposición del uso de conexionismo para simular procesos neuronales en ordenadores.

3.2. Fundamentos de las Redes Neuronales

3.2.1. La neurona desde un punto de vista biológico

Con una vista a alto nivel y una simplificación adecuada a su estructura, podríamos decir que el cerebro es un conjunto de varios millones de células particulares llamadas neuronas, que a su vez de interconectan mediante las sinapsis. Éstas últimas son las zonas donde dos neuronas se conectan y a su vez la parte de la célula especializada en esa conexión es la dendrita y ramificaciones del axón.

Cada una de las neuronas genera impulsos eléctricos transmitidos a lo largo del axón y al final de éste, donde sus ramificaciones se conectan finalmente con otras neuronas y sus dendritas.

Finalmente es la sinapsis la unidad que regula la transmisión del impulso eléctrico mediante los neurotransmisores, que no son otra cosa que elementos bioquímicos.

[1] A ó B pero no ambos

FIGURA 3.2. Neurona e impulsos eléctricos en su periferia

3.2.2. La neurona desde un punto de vista artificial

Las neuronas artificiales son modelos matemáticos enfocados en tratar de simular el comportamiento de las neuronas descritas en 3.2.1. Cada una de las neuronas se representa como una unidad de procesado que forma parte a su vez de una sistema mayor, la red neuronal.

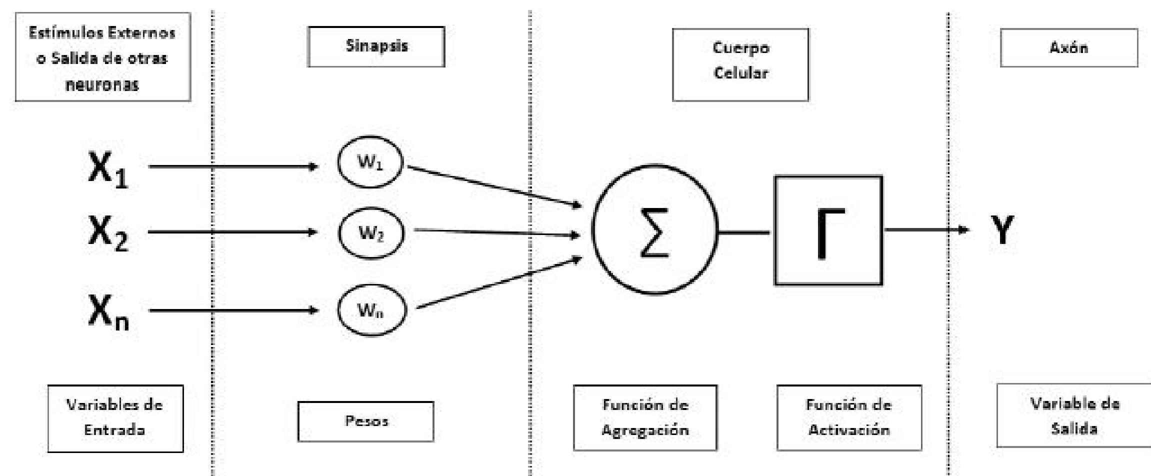


FIGURA 3.3. Esquema de una neurona artificial

Como se puede observar en la Figura 3.3, dicha unidad de procesado consta de una serie

de estímulos de entrada X_i, que equivalen a las dendritas de donde se recibe la estimulación, ponderadas por unos pesos W_i , que representan cómo se evalúan los impulsos de entrada y a su vez se combinan con la función de la red que nos dará el nivel de salida de la neurona.

De esta manera, llamando y a la salida de la red neuronal tendremos:

$$y = f(\sum_{1}^{n}(w_i * x_i))$$

Donde *f()* es la función de activación de cada neurona. Esta función de activación se suele escoger con unas propiedades determinadas de manera que se impida que la neurona desborde bien superiormente bien inferiormente.

Las funciones de activación más utilizadas son:

- **Función lineal**, que equivale a aplicar la función $y = f(u) = x$.
- **Función escalón**, usada cuando las salidas son binarias de manera que si la suma es menor que el umbral la salida será '0', siendo '1' en caso contrario.
- **Función sigmoidal**

$$f(u_i) = \frac{1}{1 + exp\{-\frac{u_i}{\sigma^2}\}}$$

- **Función gaussiana**

$$f(u_i) = c * exp\{-\frac{u_i^2}{\sigma^2}\}$$

Cada uno de estos pesos W_i se irán actualizando durante la fase de aprendizaje de la red neuronal de manera que, según el criterio de convergencia establecido, se determine que la red neuronal está debidamente entrenada para el uso que se le pretende dar.

3.3. Entrenamiento de una red neuronal

Dentro del modelado de una red neuronal existen dos fases claramente diferenciadas:

- **Fase de entrenamiento:** se emplea un conjunto de vectores de entrenamiento para calcular los pesos y sus actualizaciones que definen la red neuronal. Se calculan iterativamente a fin de minimizar el error obtenido entre la salida calculada por la red neuronal y la salida objetivo.
- **Fase de prueba:** el modelo obtenido en la fase anterior es posible que presente un ajuste demasiado exacto a las particularidades de los patrones de entrenamiento en detrimento de la habilidad para generalizar a casos nuevos (sobreajuste). A fin de evitar este problema, se emplea un segundo conjunto de datos diferentes a los usados en el entrenamiento, el grupo de validación, a fin de controlar el aprendizaje.

Una red neuronal queda determinada por el número de neuronas y su distribución en capas así como las diferentes matrices de pesos. A su vez, las capas pueden clasificarse en tres tipos:

- **Capa de entrada:** formada por las neuronas que introducen los vectores de entrada a la red. Esta capa no realizar ningún tipo de procesado de datos.
- **Capas ocultas:** formadas por las neuronas cuyas entradas vienen de capas anteriores, ya sean de entrada o también ocultas, y a su vez sus salidas alimentan neuronas de capas posteriores, ya sean ocultas o de salida.
- **Capa de salida:** formada por las neuronas cuyas salidas son directamente las salidas de la red neuronal.

Entre dos capas adyacentes de neuronas existe una red de pesos W_i que se pueden catalogar de la siguiente forma:

- Hacia delante, que conectan las neuronas de cada capa con la siguiente.
- Hacia atrás, que conectan las neuronas de cada capa con la anterior.
- Laterales, que conectan neuronas en una misma capa.

De forma general los pesos óptimos se obtienen optimizando (minimizando) alguna función de error. Por ejemplo, un criterio muy utilizado en el entrenamiento supervisado (2.1.1), es minimizar el error cuadrático medio entre el valor de salida y el valor objetivo.

La tabla 3.1 resume los algoritmos de aprendizaje más conocidos para el entrenamiento de una red neuronal.

3.4. Perpectrón Multicapa

Se denomina Perceptrón Multicapa o MLP[2] a una red neuronal artificial formada por múltiples capas con un número de neuronas en cada capa en función de su aplicación final. Esto le permite resolver problemas no linealmente separables, superando en este caso las prestaciones del perceptrón simple. El perceptrón multicapa puede ser total o localmente conectado. Si es totalmente conectado, cada una de las salidas de la capa "*i*" es entrada para todas las neuronas de la capa siguiente "*i+1*", mientras que por otro lado, si es localmente conectado cada neurona de la capa "*i*" es entrada de una región de neuronas de la capa "*i+1*".

3.4.1. Entrenamiento del MLP

Uno de los algoritmos empleado en el entramiento de los MLP es el algoritmo *backpropagation* o propagación hacia atrás (conocido asímismo por el nombre de regla delta generalizada o retropropagación).

[2]Del inglés MultiLayer Perceptron

Algoritmos de aprendizaje más conocidos				
Paradigma	Regla de aprendizaje	Arquitectura	Algoritmo de aprendizaje	Tareas
Supervisado	Corrección del error	Perceptrón o perceptrón multicapa	Algoritmos de aprendizaje perceptrón, retropropagación del error, ADALINE, MADALINE	Clasificación de patrones, aproximación de funciones, predicción, control, ...
		Elman y Jordan recurrentes	Retropropagación del error	Síntesis de series temporales
	Boltzmann	Recurrente	Algoritmo de aprendizaje Boltzmann	Clasificación de patrones
	Competitivo	Competitivo	LVQ	Categorización intra-clase, compresión de datos
		Red ART	ARTMap	Clasificación de patrones, categorización intra-clase
No supervisado	Corrección del error	Red de Hopfield	Aprendizaje de memoria asociativa	Memoria asociativa
		Multicapa sin realimentación	Proyección de Sannon	Análisis de datos
	Competitiva	Competitiva	VQ	Categorización, compresión de datos
		SOM	Kohonen SOM	Categorización, análisis de datos
		Redes ART	ART1, ART2	Categorización
Por refuerzo	Hebbian	Multicapa sin realimentación	Análisis lineal de discriminante	Análisis de datos, clasificación de patrones
		Sin realimentación o competitiva	Análisis de componentes principales	Análisis de datos, compresión de datos

TABLA 3.1. Algoritmos de aprendizaje más conocidos

3.4.2. Aplicaciones del MLP

El MLP se utiliza en aplicaciones de resolución de asociación de patrones, segmentación de imágenes, predictores de vídeo, compresión de datos, entre otros.

Un ejemplo de sería en un predictor de vídeo donde a partir de un contexto de N píxeles se pretende calcular el siguiente en la secuencia. En este caso el MLP tendría N neuronas de entrada, M en la capa oculta y una única neurona en la salida. Mediante el procesado de las imágenes, incluso en tiempo real, se extraería un resultado que sería una imagen con un menor grado de entropía (por ser más predecible) por lo que al aplicar una algoritmo de compresión sin

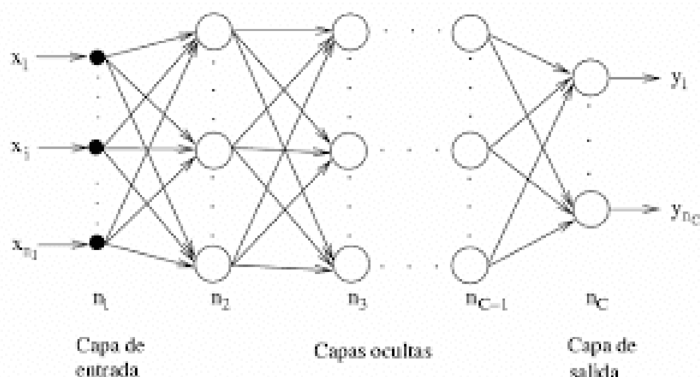


FIGURA 3.4. Perceptrón MultiCapa

pérdidas se obtendrían mejores resultados que al aplicarlo sobre la imagen original. El receptor de esas imágenes sólo necesitaría conocer los pesos utilizados en el MLP y la imagen resultante para poder recalcular la imagen inicial[20][21].

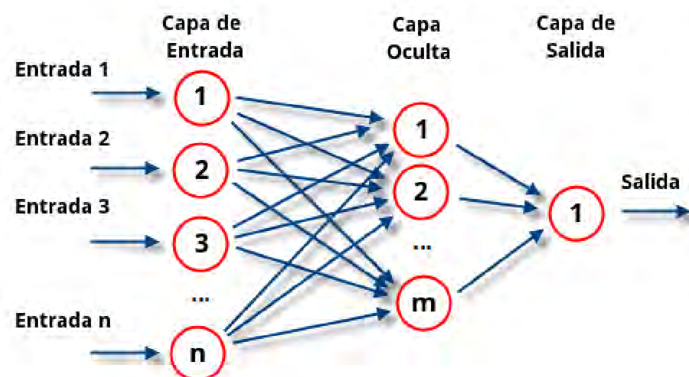


FIGURA 3.5. Perceptrón MultiCapa n:m:1

3.4.3. Limitaciones del MLP

Las limitaciones más destacables del algoritmo de un MLP se deben a que, debido a la forma en la que son entrenados, no pueden garantizar que el punto en el que se detiene (resuelve) sea un mínimo global ya que dicho algoritmo podría detenerse en un mínimo local. Una manera de corregir este problema es hacer que el algoritmo comience cada vez en un punto aleatorio y elegir el modelo que tenga un error RMS[3] menor. Además, el MLP no extrapola bien, si la red se entrena mal o de manera insuficiente las salidas pueden llegar a ser poco precisas.

[3]Del inglés Root Mean Square, error cuadrático medio

3.5. Algoritmo *Backpropagation*

El algoritmo *backpropagation* es un método utilizado en algoritmos de aprendizaje supervisado en el entrenamiento de redes neuronales artificiales y basado en el cálculo del gradiente.

El algoritmo tiene dos partes claramente diferenciadas. Por un lado propaga la señal desde la etapa de entrada hasta la capa de salida pasando por todas las capas intermedias. La salida obtenida se comparar con la salida deseada y se calcula el valor del error cometido en la aproximación para cada una de las salidas. Estos errores calculados se van propagando desde la salida hasta las entradas. He ahí el nombre de *backpropagation*, dado que se va propagando el error hacia atrás. Otra particularidad es que las neuronas de las capas ocultas sólo obtienen una parte de la contribución del error, en función de su aportación a la salida original. Si por ejemplo una neurona tiene por salida un valor próximo a 0, la fracción del error que contribuirá a su actualización en los pesos será mínima. A medida que el entrenamiento continúa en la red, cada una de las neuronas de las capas ocultas modifica su relación con el resto de manera que se organizan para reconocer diferentes características del espacio de entrada. Tras finalizar el entrenamiento, cuando la entrada sea muy ruidosa o incompleta, las neuronas ocultas proporcionarán una salida activa en función de si el nuevo vector de entrada contiene patrones que sean similares a las características que hayan aprendido a reconocer.

La descripción formal del algoritmo *backpropagation* para un MLP sería como sigue[22]:

Sea un MLP con conexión completa entre capas adyacentes con, además, funciones de activación continuas.

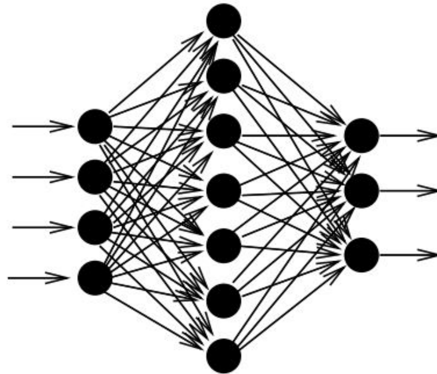


FIGURA 3.6. Perceptrón multicapa genérico

Dicho MLP realiza una función multidimensional $y = \varphi(x)$ entre la entrada $x \in \mathscr{R}^{d_i}$ y la salida $y \in \mathscr{R}^{d_o}$. Cada una de sus neuronas realiza la transformación de la entrada a través de su función de activación $f()$ (como se describió en 3.2.2) al argumento de entrada a a la neurona.

La entrada de cada neurona se obtiene como la suma ponderada de las señales que estimulan la neurona junto con las conexiones, esto es,

$$a = \sum_k w_k z_k$$

donde w_k es el peso asociado a la conexión k-ésima y z_k es la componente de la entrada k-ésima.

Pasamos ahora a describir el algoritmo *backpropagation* con más detalle.

Sea el objetivo minimizar la función

$$C = \frac{1}{2}\sum_{n=1}^{d_0}(y_n - \hat{y}_n)^2$$

donde $\hat{y}_n$ es la salida n-ésima del MLP.

Cada peso w_{ij}[4], se actualiza con la siguiente regla

$$\Delta w_{ij} = -\eta \frac{\partial C}{\partial w_{ij}}$$

La función de activación para la neurona i-ésima es $f_i(a_i)$, donde $f_i : \mathscr{R} \longrightarrow \mathscr{R}$ y

$$a_i = \sum_j w_{ij} f_j(a_j)$$

es la entrada a la neurona i-ésima.

Se tienen de esta manera dos casos diferenciados del algoritmo:

Caso 1: i está en la capa de salida

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}} &= \frac{1}{2}\sum_{n=1}^{d_0}\frac{\partial (y_n - \hat{y}_n)^2}{\partial w_{ij}} \\ &= \frac{1}{2}\frac{\partial (y_i - \hat{y}_i)^2}{\partial w_{ij}} \\ &= -(y_i - \hat{y}_i)\frac{\partial \hat{y}_i}{\partial w_{ij}} \end{aligned} \tag{3.1}$$

$$\begin{aligned} \frac{\partial \hat{y}_i}{\partial w_{ij}} &= \frac{\partial f_i(a_i)}{\partial w_{ij}} \\ &= \frac{\partial f_i(a_i)}{\partial a_i}\frac{\partial a_i}{\partial w_{ij}} \\ &= f_i'(a_i)\frac{\partial \sum_l w_{il}\hat{y}_l}{\partial w_{ij}} \\ &= f_i'(a_i)\hat{y}_j \end{aligned} \tag{3.2}$$

donde la suma sobre l se extiende a todas las neuronas en la capa oculta.

[4] w_{ij} es el peso que conecta la neurona j-ésima de una capa con la neurona i-ésima en la capa siguiente

De las ecuaciones 3.1 y 3.2 se deduce

$$\frac{\partial C}{\partial w_{ij}} = -(y_i - \hat{y}_i) f_i'(a_i) \hat{y}_j \tag{3.3}$$

Definiendo la siguiente expresión

$$\delta_i = (y_i - \hat{y}_i) f_i'(a_i) \tag{3.4}$$

Sustituyendo 3.3 en 3.4 se puede escribir finalmente:

$$\Delta w_{ij} = \eta \delta_i \hat{y}_j \tag{3.5}$$

Caso 2: la neurona j está en la capa oculta

Sea w_{jk} el peso entre la neurona k-ésima en la capa anterior y la neurona j-ésima en la capa oculta más exterior:

$$\Delta w_{jk} = -\eta \frac{\partial C}{\partial w_{jk}} \tag{3.6}$$

Nuevamente

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}} &= \frac{1}{2} \sum_{n=1}^{d_0} \frac{\partial (y_n - \hat{y}_n)^2}{\partial w_{jk}} \\ &= -\sum_{n=1}^{d_0} (y_n - \hat{y}_n) \frac{\partial \hat{y}_n}{\partial w_{jk}} \end{aligned} \tag{3.7}$$

donde

$$\begin{aligned} \frac{\partial \hat{y}_n}{\partial w_{jk}} &= \frac{\partial f_n(a_n)}{\partial w_{jk}} \\ &= \frac{\partial f_n(a_n)}{\partial a_n} \frac{\partial a_n}{\partial w_{jk}} \\ &= f_n'(a_n) \frac{\partial a_n}{\partial w_{jk}} \end{aligned} \tag{3.8}$$

y además

$$\begin{aligned} \frac{\partial a_n}{\partial w_{jk}} &= \frac{\partial \sum_l w_{nl} \hat{y}_l}{\partial w_{jk}} \\ &= \sum_l w_{nl} \frac{\partial \hat{y}_l}{\partial w_{jk}} \\ &= w_{nj} \frac{\partial \hat{y}_j}{\partial w_{jk}} \end{aligned} \tag{3.9}$$

donde, la suma sobre l se extiende a todas las neuronas en la capa oculta más externa.

Por otro lado:

$$\begin{aligned}\frac{\partial \hat{y}_j}{\partial w_{jk}} &= \frac{\partial f_j(a_j)}{\partial w_{jk}} \\ &= \frac{\partial f_j(a_j)}{\partial a_j}\frac{\partial a_j}{\partial w_{jk}} \\ &= f'_j(a_j)\frac{\partial \sum_m w_{jm}x_m}{\partial w_{jk}} \\ &= f'_j(a_j)x_k \end{aligned} \tag{3.10}$$

Finalmente, sustituyendo 3.7, 3.8, 3.9 y 3.10 en la ecuación 3.6 se obtiene:

$$\begin{aligned}\Delta w_{jk} &= \eta\sum_n[(y_n-\hat{y}_n)f'_n(a_n)w_{nj}]f'_j(a_j)x_k \\ &= \eta\sum_n[w_{nj}(y_n-\hat{y}_n)f'_n(a_n)]f'_j(a_j)x_k \\ &= \eta(\sum_n w_{nj}\delta_n)f'_j(a_j)x_k \end{aligned} \tag{3.11}$$

donde δ_n viene definida por 3.4. Se define en este caso

$$\delta_j = (\sum_n w_{nj}\delta_n)f'_j(a_j) \tag{3.12}$$

por lo que 3.11 se convierte en

$$\Delta w_{jk} = \eta\delta_j x_k \tag{3.13}$$

la cual se conoce como la *Regla Delta* del algoritmo. Esta regla se puede emplear para todas las capas de la red neuronal simplemente extendiéndola hacia la entrada.

La condición de parada para el entrenamiento puede ser cualquiera de estas tres:

- Cuando la tasa de cambio de los pesos es menor que un límite
- Cuando las nuevas prestaciones de la red sobre nuevos datos no es del todo satisfactoria
- Cuando el cambio de prestaciones es menor que un límite

Este algoritmo resulta un tanto complicado de analizar mientras que su aplicación es muy sencilla y separable en pasos. Existen multitud de ejemplos de aplicación, habiendo escogido por mi parte el de esta web por su sencillez y simplicidad. Mediante este ejemplo se puede seguir el algoritmo para una red reducida pero representativa de otras de mayor dimensión: http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html.

3.5.1. Función de activación

Dentro de las funciones de activación típicas dentro del MLP y el algoritmo *backpropagation* es también importante tener determinadas sus derivadas ya que, como se puede observar en 3.12 y 3.4, la derivada se usa en el cálculo de la δ_j para la corrección de los pesos.

En 3.3 se presentó una breve lista de funciones de activación típicas para un MLP. A modo de ejemplo se añaden ahora otras también usadas junto con sus derivadas. Los casos presentados se han elegido por la propiedad que se puede escribir la derivada como una combinación lineaL de la original, esto es, $y' = a * y + b$, propiedad muy útil cuando se está programando un método en un ordenador, ya que se tiene guardado en memoria el valor de la función que puede ser usado para la derivada.

Por ejemplo, en una función sigmoide tenemos que:

$$\begin{aligned} y_j &= \frac{1}{1+\exp(-ax_j)} \\ y'_j &= a y_j[1-y_j] \end{aligned} \tag{3.14}$$

Y para la tangente hiperbólica

$$\begin{aligned} y_j &= a\tanh(bx_j)) \\ y'_j &= \frac{b}{a}(a-y_j)(a+y_j) \end{aligned} \tag{3.15}$$

3.6. Pasos futuros para el MLP

Aunque los MLP han sido ampliamente usados y estudiados, se comprobó que existían ciertas limitaciones sobretodo en la aplicación sobre matrices bidimensionales y en algunas situaciones de reconocimiento de texto OCR[5][23].

	Learning rate (%)	Classification rate (%)
MLP	70.72	43.4
caffe-cnn (Lenet)	88	88.39
caffe-cnn (Lenet-5)	86.23	85.53
caffe-cnn (SPnet)	**89.90**	**90.56**

TABLA 3.2. Evaluación de la precisión en GPU de MLP y diversas CNN para reconocimiento de caracteres

En estas situaciones de tareas de visión artificial o clasificación y segmentación de imágenes el MLP estudiado ha evolucionado hacia las Redes Neuronales Convolucionales o CNN[6] que serán el grueso de estudio del siguiente capítulo de la memoria.

[5]Por sus siglas en inglés Optical Character Recognition
[6]Por sus siglas en inglés Convolutional Neural Network

CAPÍTULO 4

Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales o CNN son una variedad de red neuronal artificial en la cual las neuronas se asemejan a las neuronas situadas en la corteza visual primaria dentro de un cerebro biológico[24].

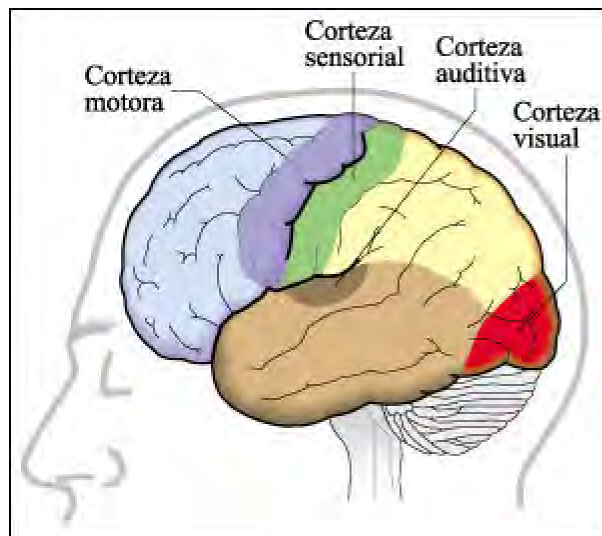


FIGURA 4.1. Localización de la corteza visual primaria

Podemos afirmar que las CNN surgieron a raíz del neocognitrón, una ANN jerárquica y multicapa diseñada por Kunihiko Fukushima en 1980[25]. El neocognitron, a su vez, fue inspirado en el anterior modelo de Hubel & Wiesel en 1959, que encontrando dos variedades de células en la corteza visual primaria: células simples y complejas; las usaron en tareas de reconocimiento de patrones mediante un modelo en cascada.

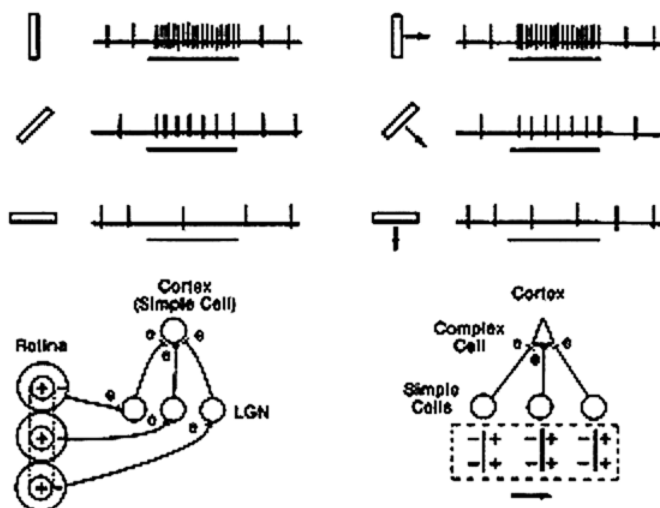


FIGURA 4.2. Cortex visual

De tal manera, una descripción de cada uno de los tipos de células encontrados en el cortex primario del cerebro, sería:

- Las **células simples** contienen regiones excitadoras y otras inhibidoras, formando en ambos casos patrones básicos orientados en una determinada dirección, posición y tamaño. Cuando un estímulo visual llega a la célula simple con la misma orientación y posición, alineándose con los patrones creados por la región, la célula se activa y envía una señal.
- Las **células complejas** trabajan de una forma bastante similar. Tienen de igual manera una orientación determinada sobre la cual son sensibles. Pero por otro lado, no tienen sensibilidad hacia la posición. Por tal razón un estímulo visual debe llegar de forma obligatoria con la orientación correcta para proceder a la activación de la célula.

No sólo es relevante el tipo de células, también lo es la estructura que forman entre ellas. Según se profundiza en las regiones de la corteza visual, la complejidad de los estímulos se van incrementando de la misma manera. De forma simultánea, las activaciones de las células se van haciendo cada vez menos sensibles a la posición y tamaño de los estímulos. Este proceso es debido a que las células están activando y propagando sus propios estímulos a otros elementos también conexionadas con la jerarquía gracias a la mixtura entre células simples y células complejas.

En el año 2005 se estableció el valor de GP-GPU[1] para el aprendizaje automatizado, siendo un punto a partir del cual se describieron métodos más eficientes para el entrenamiento de CNN usando GPUs. En 2011 se refinaron e implementaron en una GPU con resultados muy prometedores. Finalmente, en 2012, se mejoraron notablemente las prestaciones para bases de datos múltiples como por ejemplo MNIST, NORB, HWDB1.0, CIFAR10 e Imagenet.

[1]Por sus siglas en inglés General-Purpose computing on GPU

4.1. Arquitectura de las CNN

Mientras que los MLP se emplean con resultados satisfactorios para reconocimiento de imágenes, debido a la completa conectividad entre nodos no son apropiados para escalar a imágenes de alta resolución. Por ejemplo para una imagen de 32x32x3 se necesitarían 3072 pesos mientras que para una imagen de 200x200x3 harían falta 120.000. Además, no tienen en cuenta la estructura espacial de los datos, tratando los píxeles de entrada que están lejos de la misma manera que los que están más próximos. Por tanto, la conectividad total de neuronas es un desperdicio para reconocimiento de imágenes dominado por patrones de entrada espaciados.

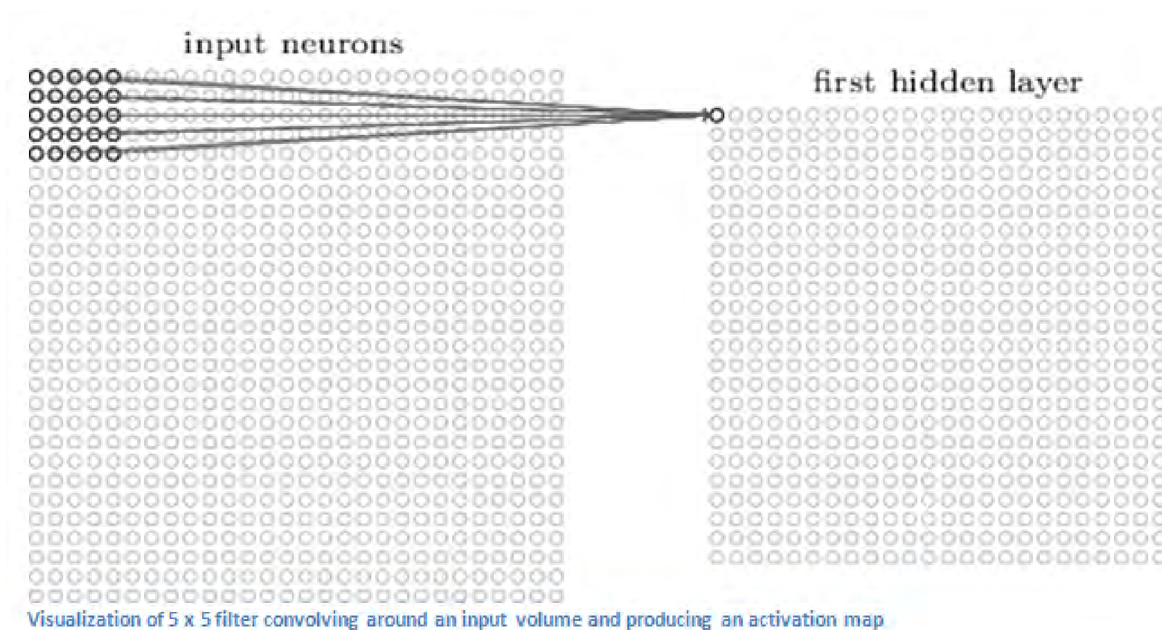


FIGURA 4.3. Mapa de activación de una Red Neuronal Convolucional

Las CNN se distinguen de las redes neuronales clásicas en cómo aprovechan el hecho de que la entrada consiste en imágenes y restringen la arquitectura de una manera óptima. En particular las capas de una CNN se distribuyen en tres dimensiones: ancho, alto y profundidad. Por ejemplo, las imágenes de entrada de CIFAR-10 se encuentran en un volumen de activaciones con un tamaño de 32x32x3. Las neuronas en una capa se conectan únicamente a una pequeña región de la capa anterior en vez de estar totalmente conectadas a todas las anteriores. Es más, la capa de salida de CIFAR-10 tendría únicamente tamaño 1x1x10 debido a que finalmente la arquitectura reduciría la imagen completa a un sólo vector de clases ponderadas ordenadas según la dimensión *profundidad*.

A nivel general de arquitectura se pueden resumir las CNN como redes con múltiples capas de filtros convolucionales con una o más dimensiones. Después de cada capa se suele añadir una

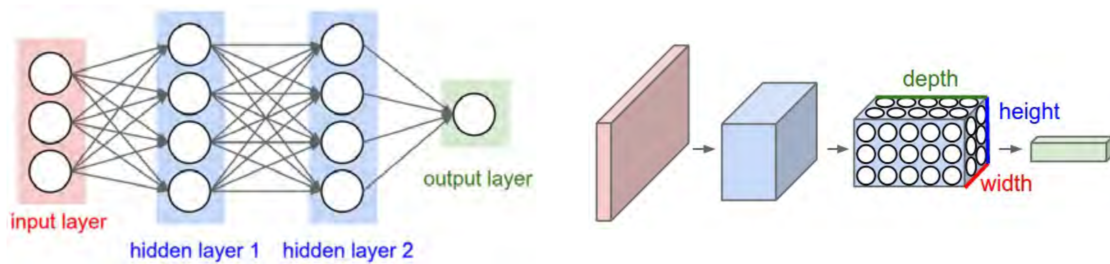
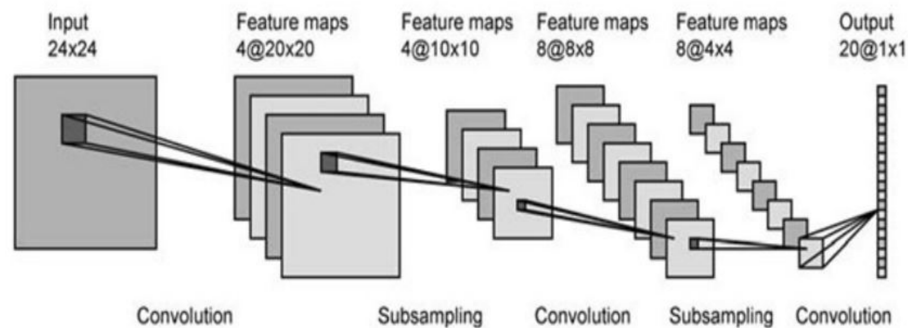


FIGURA 4.4. Red Neuronal Convolucional general de 3 capas con una vista en detalle de la transformación 2D en volúmenes 3D de activaciones de salida

función no lineal. Al ser básicamente redes de clasificación, en la entrada contienen las neuronas encargadas de la fase de extracción de características, siendo éstas neuronas convolucionales y reducción de muestreo. En las capas finales están localizadas las neuronas de perceptrón sencillas que realizan la clasificación final sobre todo el conjunto de características extraídas anteriormente. Esta última fase de extracción de características es similar al proceso de estimulación en las células de la corteza visual de un cerebro biológico. Asimismo, esta fase incluye tanto neuronas convolucionales como de reducción de muestro de forma alterna. Como se vio anteriormente en un cerebro biológico, según progresan los datos a lo largo de esta fase, se reduce su dimensionalidad. De esta manera, las neuronas en capas lejanas son más robustas a ruido en los datos de entrada pero a su vez son activadas por características cada vez más complejas[46].



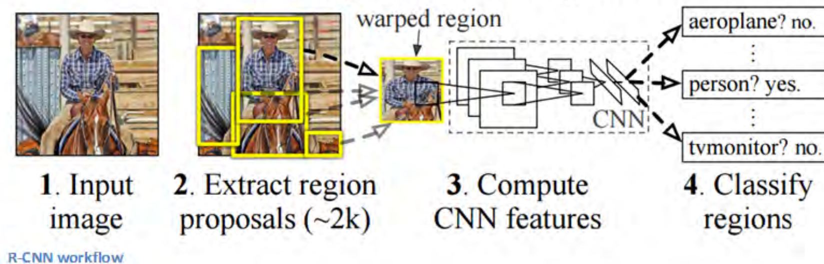


FIGURA 4.5. Desglose de capas de una CNN y aplicación a una imagen

En la Figura 4.5 se puede observar por un lado la distribución de capas y las conexiones entre ellas para una CNN aplicada a una imagen 24x24 y cómo el resultado es un vector donde cada unidad está conectada a un conjunto reducido de las capas anteriores. Además se muestra la aplicación de una CNN a la clasificación de las regiones de una imagen.

4.2. Funcionamiento de una CNN

Si en el capítulo 4.1 se ha mostrado una introducción de qué es una CNN y cómo funciona a grandes rasgos, en este capítulo se repasará de una manera más detallada, no cuáles son las capas de una CNN, sino cómo se opera con ellas y cómo son los resultados cuando se aplican a imágenes.

Estando las CNN directamente relacionadas con el aprendizaje profundo[26] y siendo un tema tan reciente es más común encontrar literatura online que en formato clásico. Así pues, la mayor parte de las referencias estarán relacionadas con webs con un tiempo de renovación muy breve por los rápidos avances actuales de estos temas.

4.2.1. Estructura

Básicamente una CNN toma cada imagen, la pasa a través de una serie de capas convolucionales, no lineales, *pooling* y totalmente conectadas para extraer una salida. La salida puede ser una única clase o bien una distribución de probabilidad entre las clases que mejor describen la imagen.

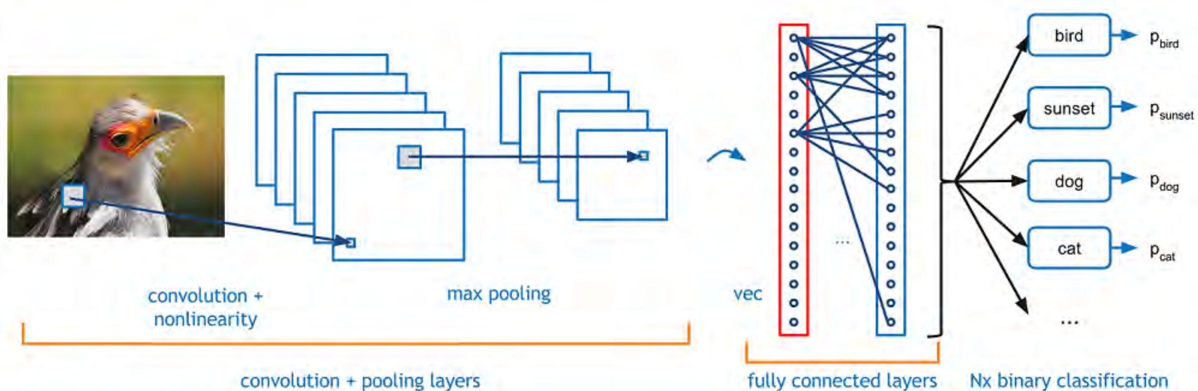


FIGURA 4.6. Procesado en una CNN

La parte complicada es entender el funcionamiento de cada una de las clases. A continuación repasaremos las más importantes[27].

4.2.1.1. Primera capa: parte matemática

La primera de una CNN siempre es una Capa Convolucional. Durante la fase de extracción de características, diferentes procesadores en matriz reemplazan a las neuronas sencillas de un MLP. Estos procesadores realizan una operación sobre los datos obtenidos de la imagen 2D tomada como entrada. La salida de cada neurona convolucional se calcula como:

$$Y_j = g\left(b_j + \sum_i K_{ij} \otimes Y_i\right)$$

Donde la salida Y_j de una neurona j es una matriz que se calcula por medio de la combinación lineal de las salidas Y_i de las neuronas en la capa anterior, cada una de ellas multiplicadas por el núcleo de convolución K_{ij} correspondiente a esa conexión. Esta cantidad se suma a una cantidad *bias* b_j y luego se pasa por una función de activación $g(*)$ no-lineal.

El operador de convolución filtra la imagen de entrada con un núcleo previamente entrenado. Esto transforma los datos haciendo más dominantes ciertas características (determinadas por la forma del núcleo) al tener un valor numérico mayor. Estos núcleos tienen habilidades de procesamiento de imágenes muy concretas, como por ejemplo la detección de bordes que se puede realizar con elementos que resaltan el gradiente en una dirección en particular. Los núcleos que son entrenados por una CNN normalmente son más complejos a fin poder extraer diferentes características más abstractas y menos comunes.

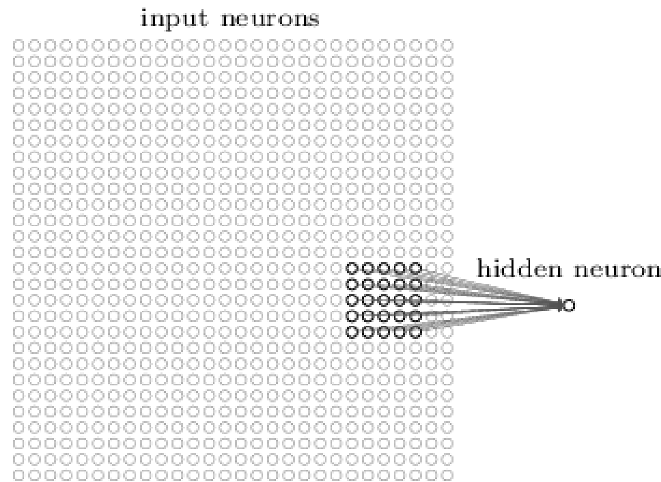


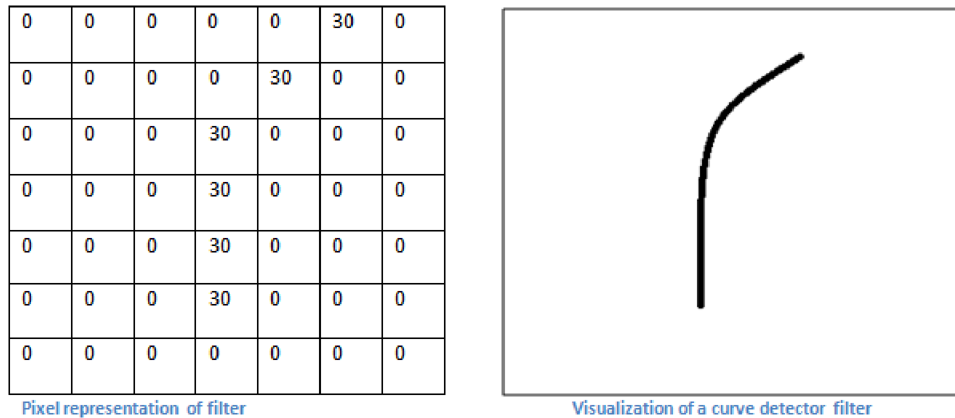
FIGURA 4.7. Primera capa de una CNN

4.2.1.2. Primera capa: vista de alto nivel

Este capítulo mostrará esta primera capa convolucional desde una perspectiva de alto nivel. Cada uno de estos filtros se pueden entender como unos identificadores de características, entendiéndose esta expresión como rectas, colores, curvas, etc. Es decir, esta primera capa tendrá

como resultado una salida que indica cuáles de estas características se han reproducido en la imagen sin importar la posición donde se encuentran o factores de escala.

Supongamos que el filtro se va a comportar como un detector de curvas. En este caso tendrá una estructura de píxeles donde el mayor valor numérico será el área ocupada por la forma de una curva.



0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

FIGURA 4.8. Filtro para detección de curvas

Aplicando el filtro anterior a una imagen muy sencilla de ejemplo en la que queremos clasificar sus trazos tendríamos un valor numérico fruto de la convolución[2] del filtro con la imagen.

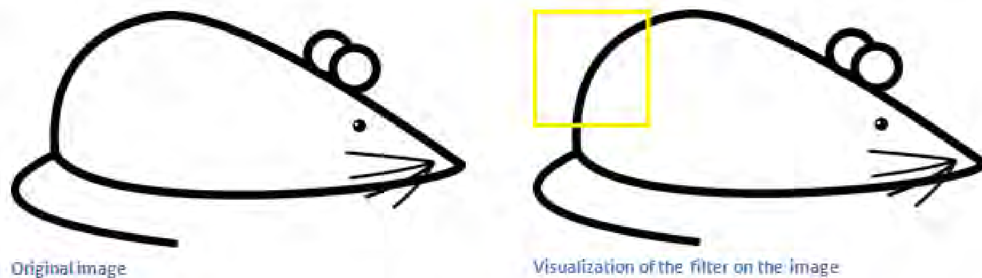


FIGURA 4.9. Imagen a la que se aplica el filtro convolucional

Convolucionando el filtro con la imagen tendríamos un valor numérico que determinaría mediante un límite inferior si en efecto hemos detectado una curva o no. Las figuras siguientes muestras este producto convolución de una manera sencilla, simplemente superponiendo el fragmento de imagen con un filtro, ambos del mismo tamaño, para que sea más sencillo comprobar cómo sería el producto.

[2]Se define el operador convolución como $(f * g)(t) = \int_{-\infty}^{+\infty} f(\eta)g(t-\eta)d\eta$

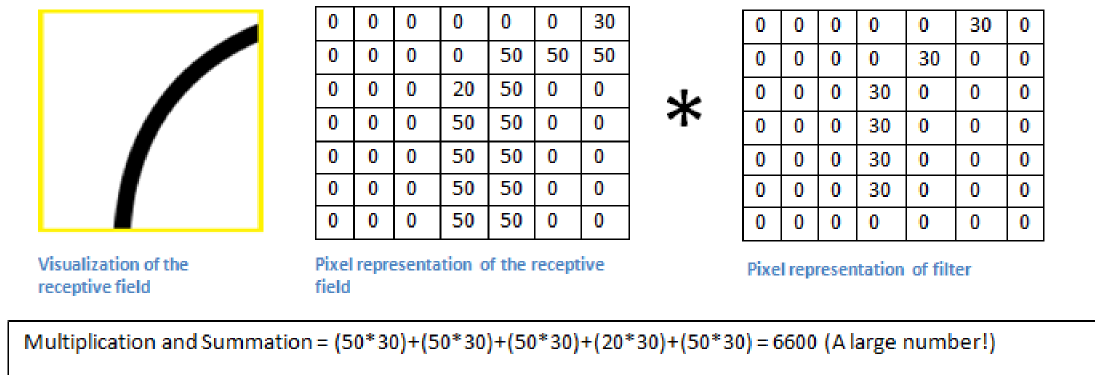


FIGURA 4.10. Convolución de la curva con el filtro

En este caso la convolución entrega un valor final alto, por lo que matemáticamente hablando, hemos detectado correctamente una curva.

Por otro lado, aplicando el mismo filtro a otra zona de la imagen, el resultado calculado sería muy inferior por lo que no se habría aplicado sobre una curva semejante a la del filtro.

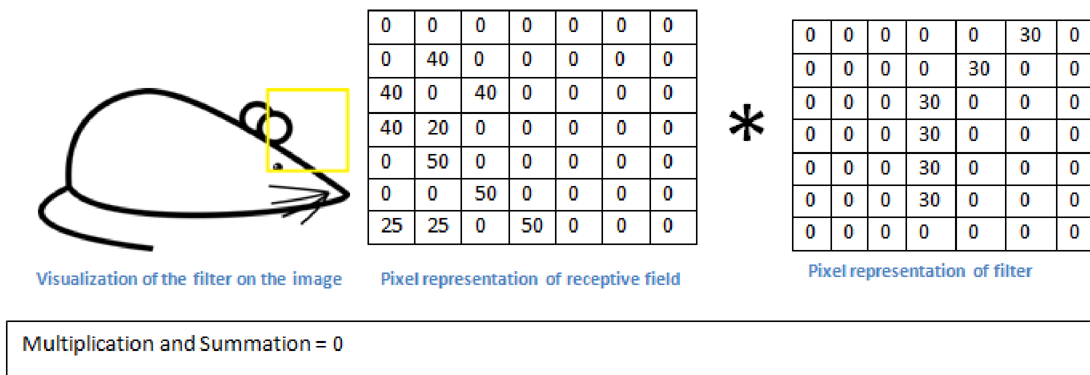


FIGURA 4.11. Convolución de otra zona con el filtro

Fijando un umbral determinado, podríamos comprobar que en el caso de detectar correctamente la curva la salida podría ser binaria 0/1 con un éxito en este caso, mientras que en el segundo producto no se ha encontrado una curva en la imagen a procesa.

Se aprecia que a grandes rasgos, la convolución de la imagen con el filtro se calcula como

$$C[m,n] = \sum_u \sum_v A[m+n, n+v]B[u,v]$$

Es decir, cada elemento de la matriz resultado C se calcula como la suma de los productos de cada elemento individual de A por B.

En resumen, esta capa convolucional realiza un filtrado de la imagen inicial para encontrar diferentes características y tener una salida simplificada con un único valor para cada una de ellas. Cada una de estas características se incluye en un filtro previamente entrenado que servirá para obtener una salida que indica si se han detectado las características o no. Si por ejemplo quisiéramos detectar el color dominante de una imagen, el filtro podría ser simplemente el valor RGB del color y si la salida es superior a un umbral dado, afirmar que la imagen contiene ese color de una manera predominante.

Se puede consultar una animación muy intuitiva sobre cómo funciona el producto convolucional adaptado a imágenes en [28]. La demostración del funcionamiento de una capa convolucional muestra claramente, para dos filtros arbitrarios propuestos, cómo cada fragmento de la imagen se desliza sobre la matriz que contiene el filtro para calcular en el volumen de salida un valor resultado que contiene la información sobre si se ha encontrado, o no, alguna de las características contenidas en el filtro dentro de la imagen. De esta manera, con independencia de la posición de la información en la imagen, el filtro se aplica por toda ella. Si hubiera algún factor de escala también estaría resuelto al trocear la imagen inicial en varios fragmentos más pequeños. El operador convolución realiza un deslizamiento de las imágenes a las que se aplica, por lo que aunque estuvieran desplazadas, tendríamos una coincidencia igualmente. Además, a la hora de fragmentar la imagen se tiene en cuenta posible solapamiento entre cada fragmento, por lo que se aplicará el filtro convolucional de manera eficiente.

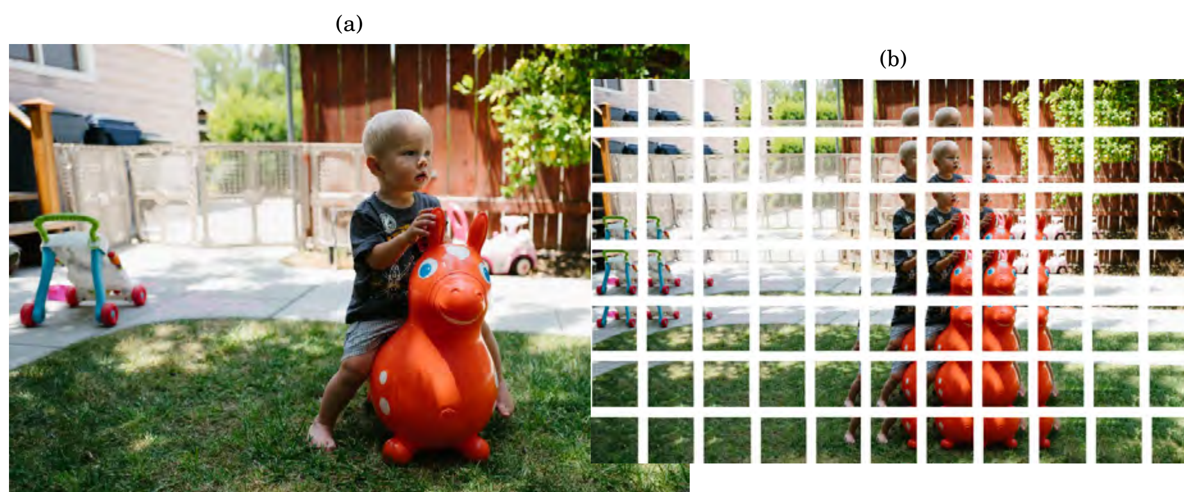


FIGURA 4.12. La imagen de la izquierda es la imagen original 4.12(a), la imagen de la derecha muestra cómo se ha fragmentado y superpuesto los diferentes fragmentos 4.12(b)

Las Figuras 4.12(a) y 4.12(b) han sido extraídas de [29] donde además se puede ampliar información sobre redes neuronales y *machine learning*.

4.2.1.3. Capa totalmente conectada

Ahora que se pueden detectar todas estas características de alto nivel en las capas de entrada y ocultas, es la capa de salida la que necesita estar totalmente conectada. Esta capa básicamente toma un volumen de entrada y tiene por salida un vector de dimensión N donde este valor es el número de clases frente a las cuáles vamos a clasificar los elementos de la imagen. Por ejemplo, si queremos clasificar dígitos, N=10 dado que habrá 10 dígitos. Puede también darse el caso en el cual el vector de salida asigne una probabilidad a diferentes elementos en vez de dar una salida '0' ó '1'. Por ejemplo, en el caso de reconocimiento de dígitos podría obtener una salida [0 .1 .1 .75 0 0 0 0 0 .05], dando en este caso una probabilidad del 10% a que el dígito sea un '1' ó '2', 75% a un '3', etc.

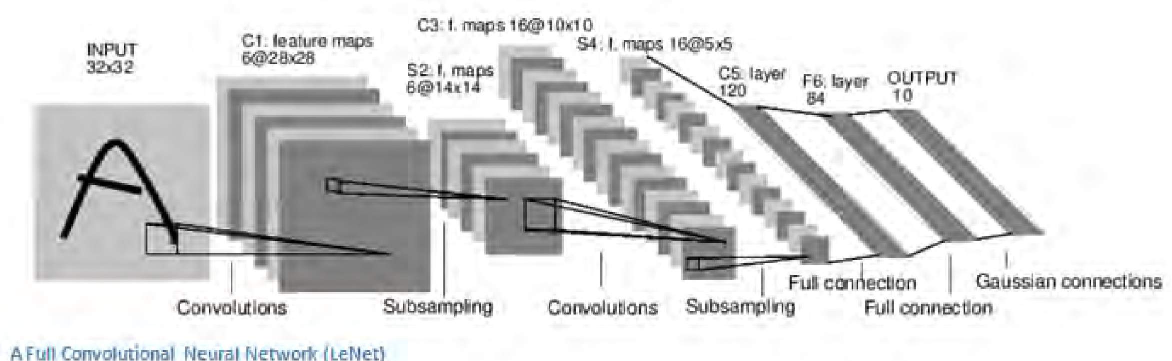


FIGURA 4.13. Red Neuronal Convolucional completa

4.2.1.4. Capa no lineal o de activación

Después de cada capa convolucional se incluye una no lineal o de activación en línea con el funcionamiento visto en 3.2.2. En el pasado se usaban los modelos sigmoidales o *tanh* aunque ahora se emplean principalmente capas ReLU[3] por su alta velocidad de entrenamiento sin impactar de forma notable en la precisión. Estas capas ReLU aplican la función $f(x) = max(0, x)$ a todos los valores del volumen de entrada. Además aumentan las propiedades no lineales del modelo y de la red sin impactar las capas convolucionales[30].

4.2.1.5. Capa de *pooling*

Tras algunas capas ReLU en algunos casos se elige añadir una capa de *pooling* o *downsampling*. Aunque existen varias opciones de implementación la función que elige el máximo de un

[3]Por sus siglas en inglés Rectified Linear Units

grupo es la más popular. Básicamente usa un filtro normalmente 2x2 y un deslizamiento del mismo tamaño y lo aplica al volumen.

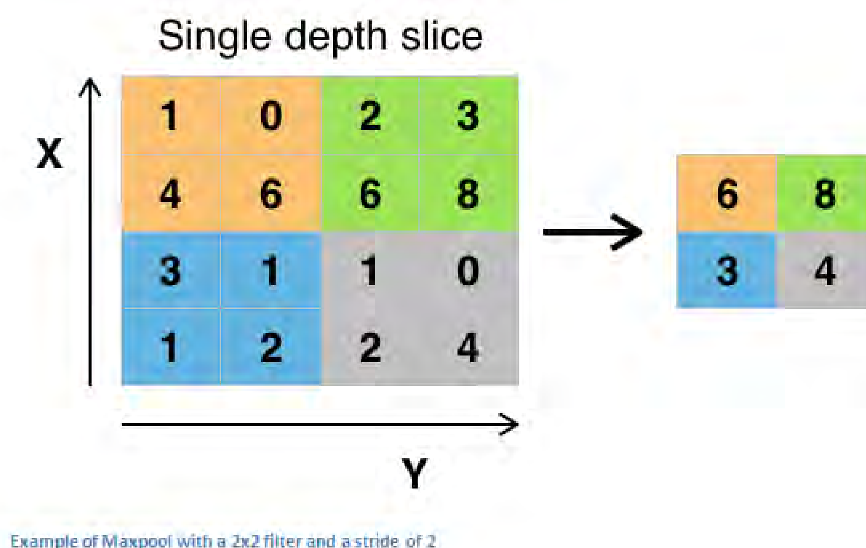


FIGURA 4.14. Pooling de 2x2

4.2.1.6. Capa de *dropout*

Para impedir, o al menos limitar, el sobreajuste de las redes neuronales, esto es, demasiado entrenadas a los patrones de entrenamiento y que pierdan capacidad de generalización futura, se usan estas capas que *pierden* determinadas activaciones ajustándolas a valor 0 [31].

4.2.1.7. Capa de red

Por capa de red se refiere a una capa convolucional donde se ha usado un filtro 1x1 [32].

4.3. Conceptos adicionales en CNN

Este apartado hará un muy breve repaso a diferentes técnicas de mejora de las prestaciones de las redes y capas convolucionales. Se puede ampliar la información con más detalle y ejemplos en [27].

4.3.1. Rellenado y deslizamiento

El deslizamiento[4] controla cómo el filtro convoluciona con el volumen de entrada. Por ejemplo, para un volumen de 7x7 se podría hacer un deslizamiento para convertirlo en un volumen 5x5.

[4] En inglés stride

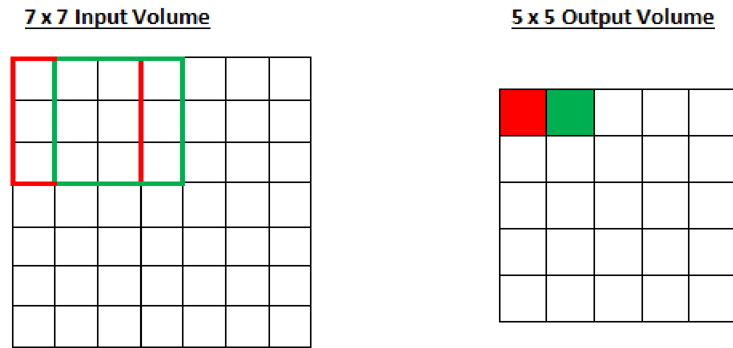


FIGURA 4.15. Deslizamiento de una unidad

Si se hace un deslizamiento de 2 unidades, el volumen resultado tendrá otro tamaño y otra combinación de relleno.

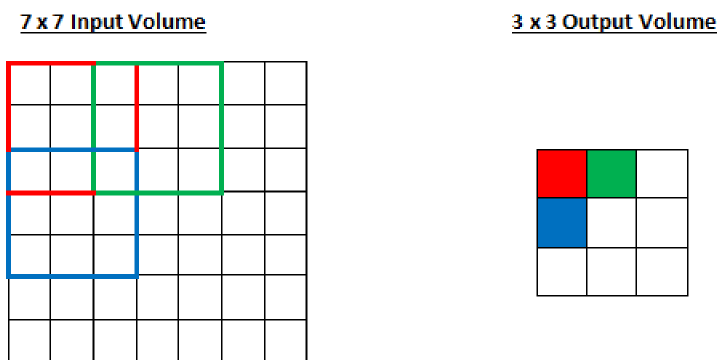


FIGURA 4.16. Deslizamiento de dos unidades

Por otro lado el rellenado de datos se emplea cuando se pretende aumentar el tamaño del volumen para que las ventanas deslizantes de convolución se necesitan variar.

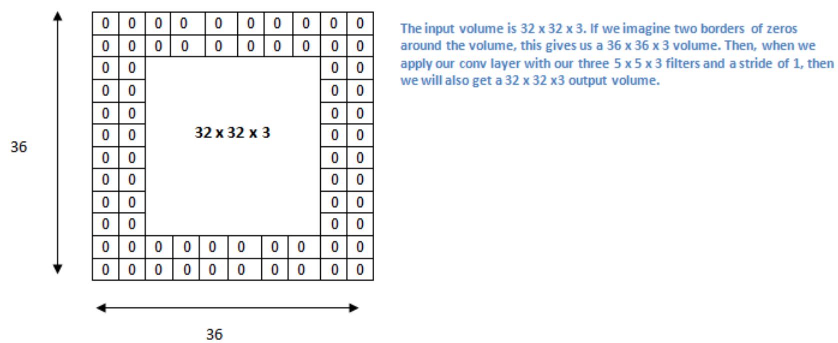


FIGURA 4.17. Rellenado de volúmenes

4.3.2. Clasificación, localización, detección y segmentación

Dentro de, no ya la clasificación de imágenes, sino de la localización de objetos el objetivo no es producir un etiquetado dentro de una clase sino generar una caja que describa dónde está el objeto en la imagen. Además dentro de una misma imagen puede haber diferentes tipos de objetos, por lo que deberán existir diferentes cajas etiquetadas identificando cada objeto. Por último es útil crear un contorno para cada objeto identificado de manera que se determine cuánta área ocupa en la imagen.

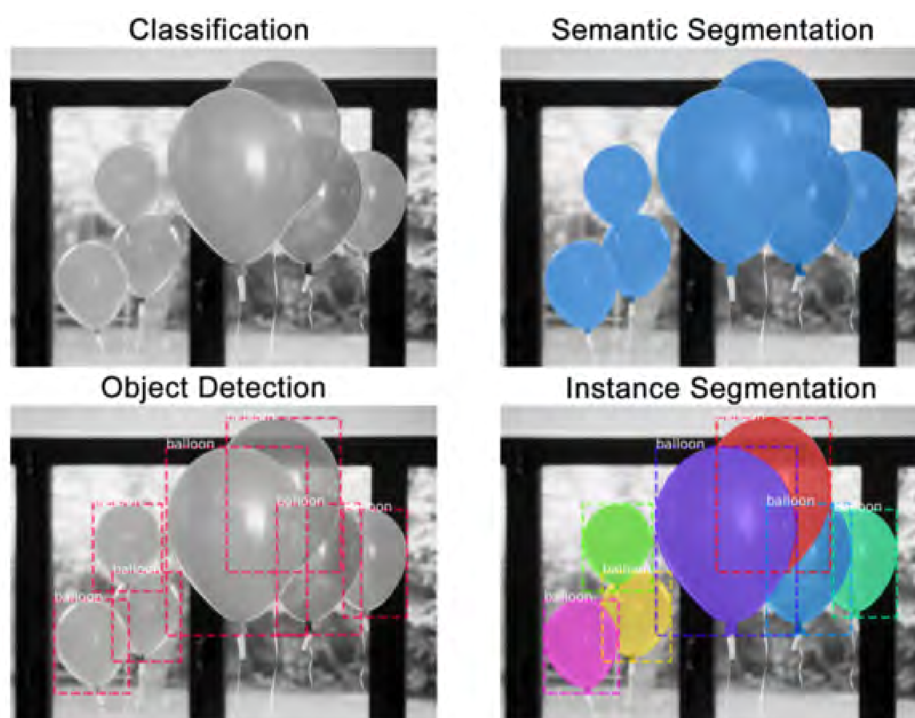


FIGURA 4.18. Clasificación, segmentado y detección en una imagen

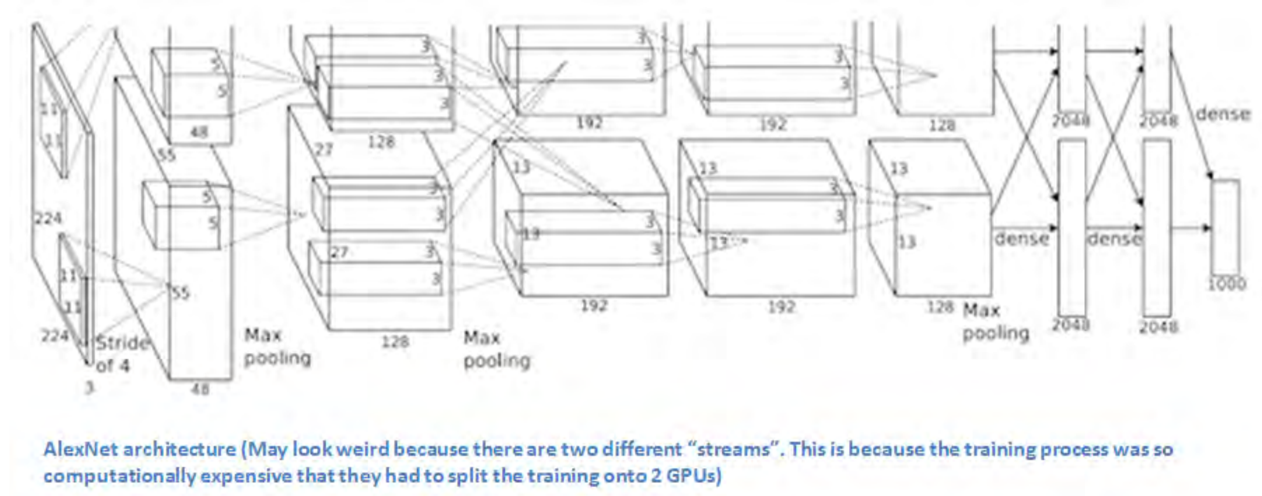
4.4. Desarrollos de visión computerizada y CNN

En esta sección de resumirán diferentes desarrollos en el campo de la visión artificial y las CNN que, de alguna manera, hayan sido relevantes o nuevos. Se incluirán diferentes publicaciones aparecidas en los últimos años y un resumen de su importancia[33].

4.4.1. AlexNet (2012)

La primera red que comenzó el camino fue la llamada AlexNet[34]. La publicación [34] ha sido citada en más de 6000 ocasiones y es considerada como una de las más influyentes en el campo de las CNN. En la publicación, el grupo repasó la arquitectura de la red llamada AlexNet. Se usó

una arquitectura muy báscia comparada con las redes normales. Incluía 5 capas convolucionales, *max-pooling*, *dropout* y 3 totalmente conectadas. La red se diseñó para clasificar hasta 1000 posibles categorías.



AlexNet architecture (May look weird because there are two different "streams". This is because the training process was so computationally expensive that they had to split the training onto 2 GPUs)

FIGURA 4.19. Arquitectura AlexNet

Sus principales características son:

- La red se entrenó con datos ImageNet, conteniendo sobre 15 millones de imágenes para un total de 22000 categorías.
- Se usó ReLU(4.2.1.4) para las funciones no lineales.
- Se usaron técnicas de aumento de datos como traslaciones, reflexiones y extracción de patrones.
- Se usaron capas *dropout* para minimizar el problema de sobreajuste de los datos de entrenamiento.
- El modelo se entrenó usando el método del gradiente descendente estocástico.
- El entrenamiento llevó de 5 a 6 días en dos GTX 580.

La red AlexNet es importante por ser de las pioneras en la comunidad de visión computerizada. Fue el primer modelo desarrollado correctamente en el conjunto de datos ImageNet. Usando técnicas empleadas en la actualidad se demuestran los beneficios de las CNN con un récord a nivel de prestaciones en el concurso.

4.4.2. ZF NET (2013)

Mientras que en 2012 AlexNet ganó la competición, hubo un importante aumento de modelos enviados al ILSVRC[5] en 2013. El ganador del concurso De ese año fue una red diseñada por Matthew Zeiler y Rob Fergus. Esta red, llamada ZF Net[35], consiguió un porcentaje de error del 11.2 %. Dicha red fue más que un ajuste de la estructura previa AlexNet, incluyendo buenas ideas clave sobre mejora de prestaciones.

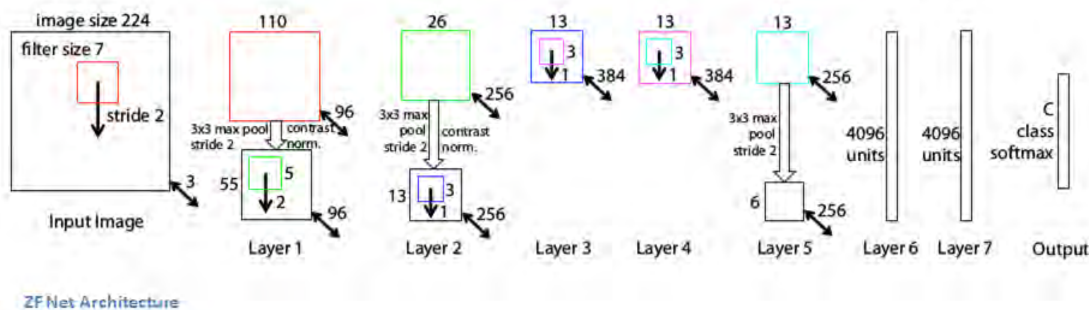


FIGURA 4.20. Arquitectura ZF Net

Sus principales características son:

- Arquitectura muy similar a AlexNet excepto por unas pocas modificaciones.
- ZF Net se entrenó sobre 1.3 millones de imágenes mientras que AlexNet sobre 15 millones.
- En vez de usar filtros de tamaño 11x11 se usaron filtros de tamaño 7x7 y un valor reducido de desplazamiento.
- Según aumenta la red se comprueba cómo el número de filtros también aumenta.
- Se usaron ReLUs para las funciones de activación, como función de error pérdidas de entropía cruzadas y entrenada mediante el método del gradiente descendente estocástico.
- Entrenada sobre una GTX 580 durante 12 días.
- Se desarrolló una técnica de visualización llamada Red Deconvolucional de forma que se ayudó a examinar diferentes características de activación y su relación con el espacio de entrada.

La red ZF Net es importante no sólo por haber ganado el concurso de 2013, también por que aportó una vista del trabajo de las CNN así como más formas para mejorar las prestaciones.

[5]ImageNet Large Scale Visual Recognition Competition

4.4.3. VGG Net (2014)

Este simple y profundo modelo creado en 2014 consiguió una tasa de error del 7.3 %. Karen Simonyan y Adrew Zisserman crearon una red de 19 capas que usó filtros de 3x3 con un deslizamiento de 1 así como capas *maxpool* de 2x2 con desplazamiento de 2[36].

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB imag)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

The 6 different architecures of VGG Net. Configuration D produced the best results

FIGURA 4.21. Arquitectura VGG Net

Sus principales características son:

- Usa filtros de tamaño 3x3 frente a los filtros 11x11 y 7x7 de las primeras capas de AlexNet

y ZF Net, respectivamente.

- 3 capas convolucionales.
- Mientras que el tamaño espacial de los volúmenes de entrada decrece, la profundidad de los volúmenes aumentan debido al mayor número de filtros según se profundiza en la red.
- El número de filtros se dobla tras cada capa *maxpool*. Esto refuerza la idea de reducir la dimensión espacial pero aumentar la profundidad.
- Trabaja bien en tareas de clasificación y localización.
- El modelo fue creado con el paquete Caffe.
- Se usó escalado *jittering* y una técnica de aumento de datos durante el entrenamiento.
- Se usaron capas ReLU después de cada capa convolucional y se entrenó con el método del gradiente descendente.
- Entrenada sobre 4 Nvidia Titan Black durante 2-3 semanas.

Este modelo es importante debido a que es uno de las publicaciones más influyentes dado que refuerza la noción de que las CNN deben tener una red profunda de capas para la representación jerárquica de los datos visuales de trabajo.

4.4.4. GoogLeNet (2015)

Desarrollada por Google, fue la ganadora del ILSVRC en 2014 con una tasa de error del 6.7% con una estructura de 22 capas[37].

Sus principales características son:

- Tiene 100 capas en total, usando 9 módulos paralelos.
- No usa capas totalmente conectadas. Usa un *pool* de promediado, pasando de un volumen de 7x7x1024 a uno de 1x1x1024. Esto ahorra muchos parámetros.
- Usa unas 12 veces menos parámetros que AlexNet.
- Durante el test se crearon múltiples detalles de la misma imagen, alimentadas dentro de la red y las probabilidades softmax fueron promediadas para obtener la solución final.
- Usa conceptos de las R-CNN[6] para su modelo de detección.
- Se entrenó en unas pocas GPUs durante una semana.

GoogLeNet es uno de los primeros modelos que pusieron de manifiesto la idea de que las CNN no siempre necesitan ser apiladas secuencialmente.

[6] Region-Convolutional Neural Network

4.4.5. Microsoft ResNet (2015)

ResNet es una red con 152 capas que fijó nuevos récords en clasificación, detección y localización a través de una arquitectura increíble. Además los récords en cuanto al número de capas, ResNet ganó ILSVRC en 2015 con una tasa de error del 3.6 % [38].

Sus principales características son:

- Ultraprofunda.
- 152 capas.
- Tras las dos primeras capas, el tamaño espacial se comprime desde un volumen de 224x224 a uno de 56x56.
- Los autores indican que un pequeño incremento de capas resultan en mayor entrenamiento y error de test.
- El grupo intentó una red de 1202 pero obtuvo precisión más baja, debido seguramente al sobreajuste.
- El entrenamiento se hizo sobre unas 8 GPUs durante 2-3 semanas.

Con un 3.6 % de tasa de error (un humano puede estar entre un 5-10 %) se trata de una red impresionante.

4.4.6. CNN basadas en Regiones

El desarrollo de las CNN basadas en regiones o R-CNN ha sido uno de los trabajos más determinantes en lo que a nuevos tipos de redes se refiere. La primera de las publicaciones de este tipo de redes acumula ya más de 1600 citas diferentes.

4.4.6.1. R-CNN (2013)

El propósito de las R-CNN es resolver el problema de la detección de objetos. De una determinada imagen ha de ser capaz de dibujar cajas alrededor de todos los objetos [39]. El proceso se puede dividir en dos pasos:

- Propuesta de la región de interés o ROI[7]
- Clasificación de la región

[7] Por sus siglas en inglés Region Of Interest

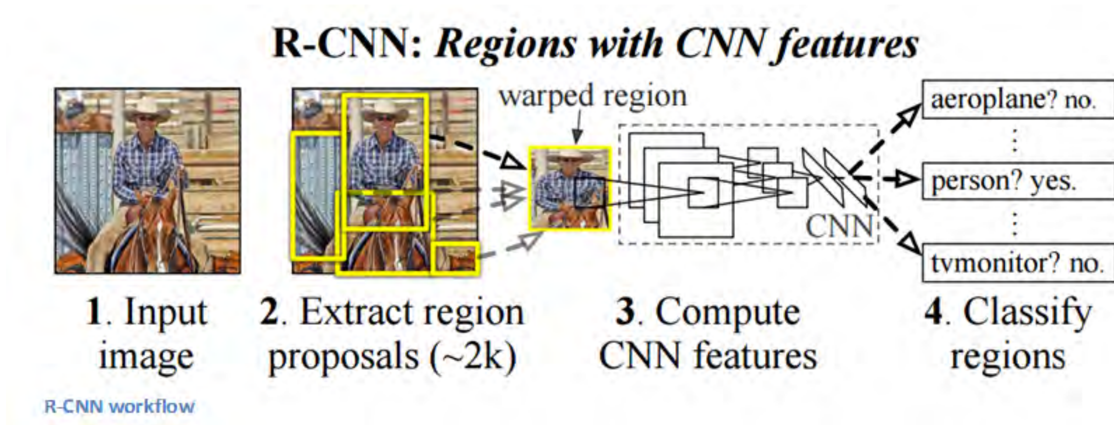


FIGURA 4.22. Flujo de trabajo de una R-CNN

Usando el método mediante propuestas de regiones se llegan a crear hasta 2000 regiones de interés que, a su vez, se deforman para servir de entrada a una red CNN de manera individual. Tras ella, se emplean capas totalmente conectadas a fin de clasificar el objeto y definir de manera precisa la caja que lo rodearía [40].

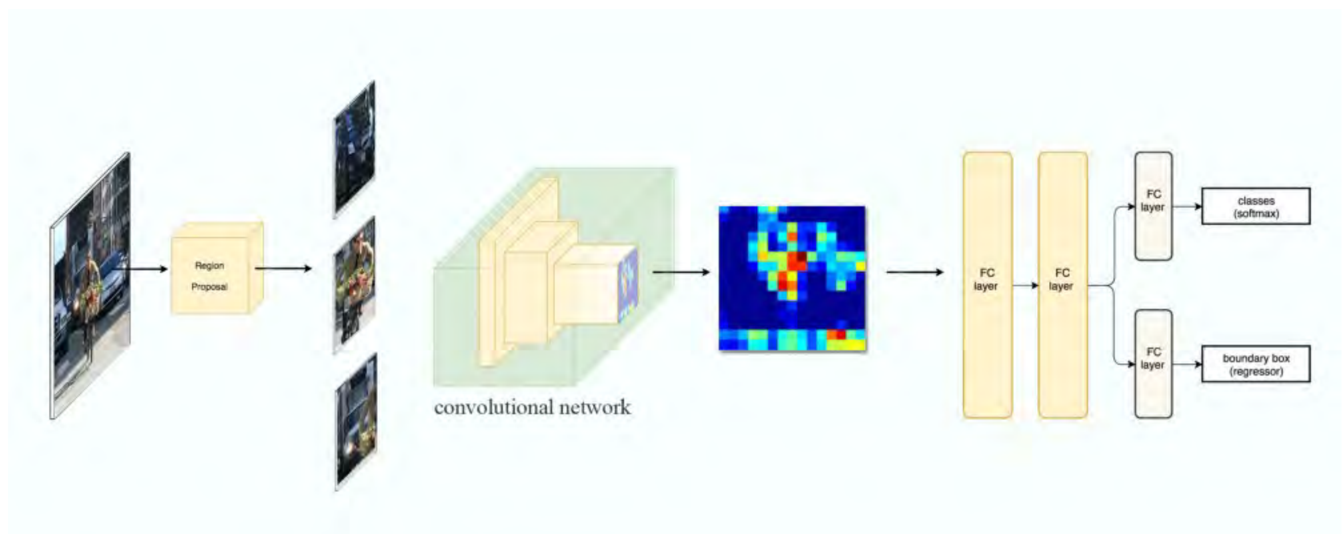


FIGURA 4.23. Arquitectura de una R-CNN

Generalmente los métodos de cálculo para encontrar las propuestas de regiones poseen una alta complejidad computacional. A fin de acelerar el proceso generalmente es una buena idea utilizar una ROI seguida de un regresor lineal mediante capas totalmente conectadas a fin de refinar las cajas de selección.

4.4.6.2. *Fast* R-CNN

Se hicieron varias mejoras a la red original debido a tres problemas principales:

- El entrenamiento usaba demasiadas fases.
- Era computacionalmente hablando muy costosa con muchas regiones solapadas entre ellas.
- Era extremadamente lenta durante el entrenamiento y la inferencia.

Básicamente, una red R-CNN es poco eficiente en lo que respecta al tratamiento de las propuestas ya que cada una de ellas se procesa mediante una CNN de manera independiente. Si tenemos 1000 ROIs, habría que realizar la extracción de características 1000 veces.

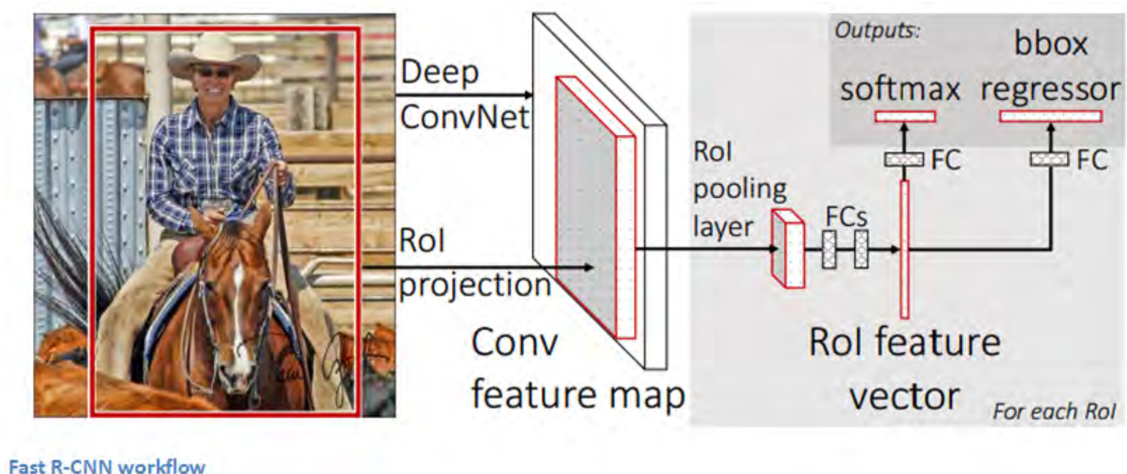


FIGURA 4.24. Flujo de trabajo de una *Fast* R-CNN

Fast R-CNN fue capaz de solventar el problema de la velocidad simplemente compartiendo cálculos de las capas convolucionales al emplear diferentes propuestas e intercambiando el orden de las regiones. En vez de utilizar una CNN para analizar cada fragmento de la imagen, se utilizaba una de manera general para analizar la imagen completa al inicio. Tras ella se hacía una búsqueda selectiva para seleccionar regiones que, combinadas con el mapa de características, permitía generar fragmentos para detección de objetos.

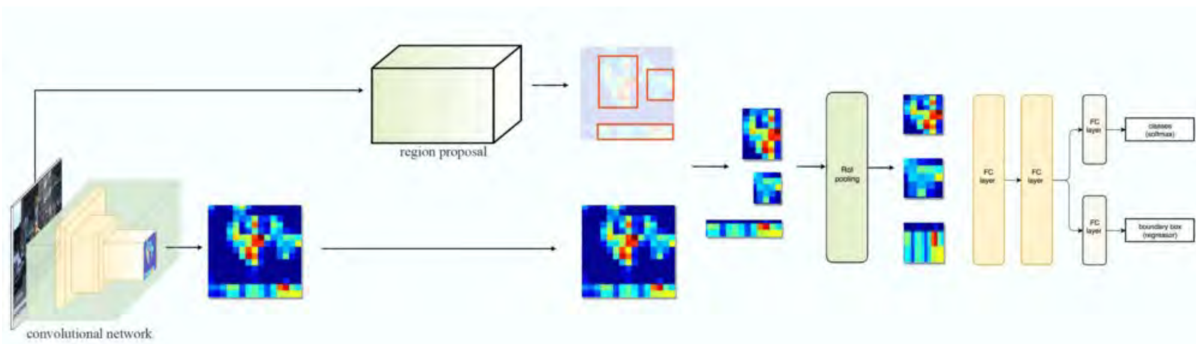


FIGURA 4.25. Arquitectura de una Fast R-CNN

Con todo esto, el grado de mejora era de unas 10 veces más rápida durante el entrenamiento y 150 durante la inferencia.

4.4.6.3. *Faster* R-CNN

Faster R-CNN se empleó para combatir la complejidad del entrenamiento mostrado tanto en R-CNN como en *Fast* R-CNN. Los autores añadieron una RPN[8] después de la última capa convolucional sustituyendo de esta manera el método de propuestas de regiones por una red profunda, de forma que las ROIs se infieren desde el mapa de características. Esta nueva RPN es más eficiente tardando únicamente unos 10ms por imagen para generar ROIs.

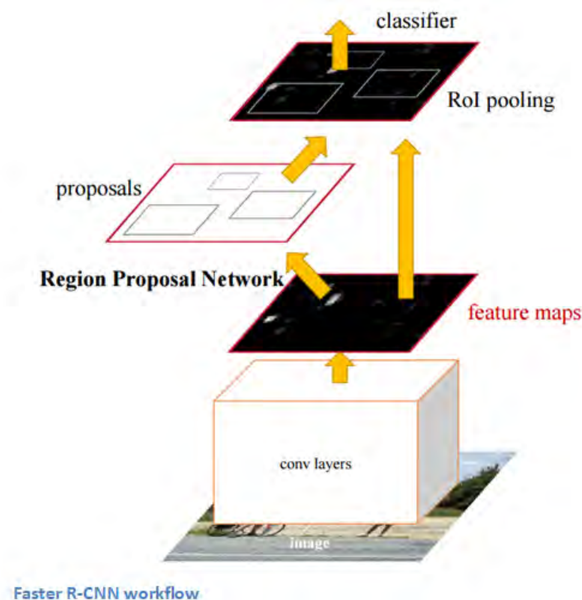


FIGURA 4.26. Flujo de trabajo de una *Faster* R-CNN

[8]Por sus siglas en inglés Region Proposal Network

Estas RPN utilizan regiones candidatas para detectar objetos en ellas que son las llamadas *anchors* o zonas de anclaje[41]. El coste en tiempo de generar las regiones candidatas es mucho menos en RPN que en una búsqueda selectiva dado que las RPN comparten la mayoría de los cálculos con la red de detección de objetos. En resumidas cuentas, las RPN asignan una valoración a cada caja de detección y marca las que, de manera más probable, contienen objetos.

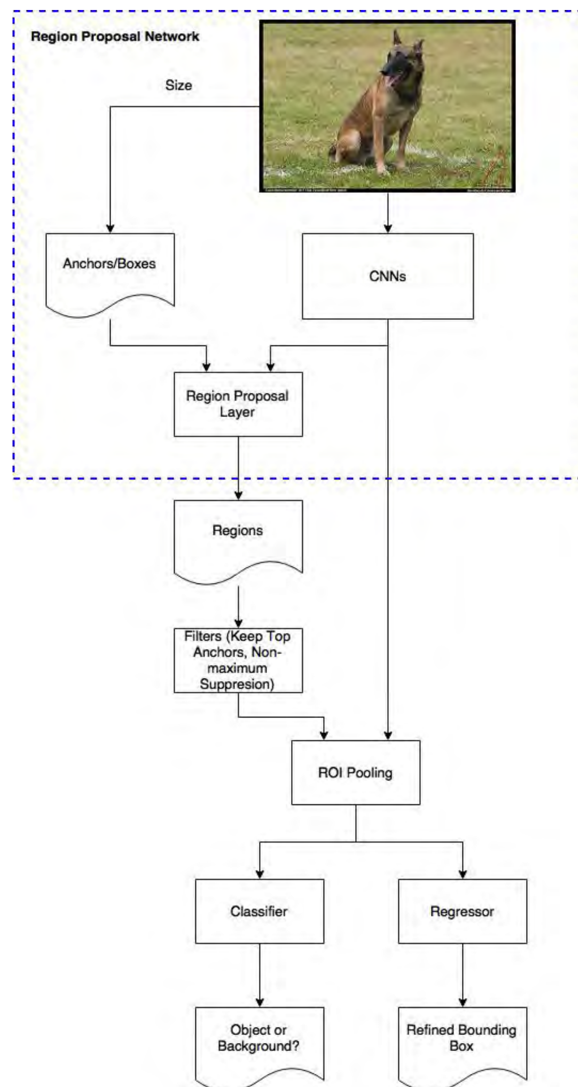


FIGURA 4.27. Arquitectura de *Faster* R-CNN

Las zonas de anclaje juegan un papel importante en *Faster R-CNN*. Básicamente son cajas situadas en las imágenes que determinan la probabilidad de que exista un objeto/región de interés o no. Esta selección se realiza mediante la RPN antes mencionada. La salida de una RPN es un conjunto de cajas y propuestas que se examinarán por un clasificador para comprobar la existencia de objetos. Es decir, una RPN predice la posibilidad de que una sección se encuentre al

fondo o al frente de la imagen y refina el tamaño de la caja.

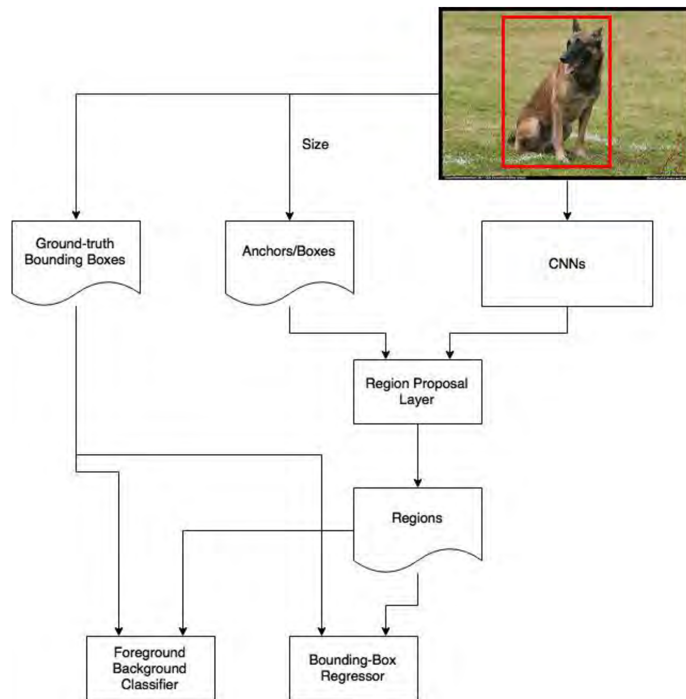


FIGURA 4.28. Entrenamiento de una RPN

Durante el entrenamiento mediante un conjunto de datos como por ejemplo COCO descrito en el Anexo A se pueden encontrar las etiquetas para cada una de las cajas calculadas en la imagen de manera que se diferencien entre el fondo de la imagen y el frente.

Esta red se he convertido actualmente en el estándar para detección de objetos.

La Figura 4.29 muestra una comparativa entre las tres redes antes enunciadas: R-CNN, Fast R-CNN y Faster R-CNN.

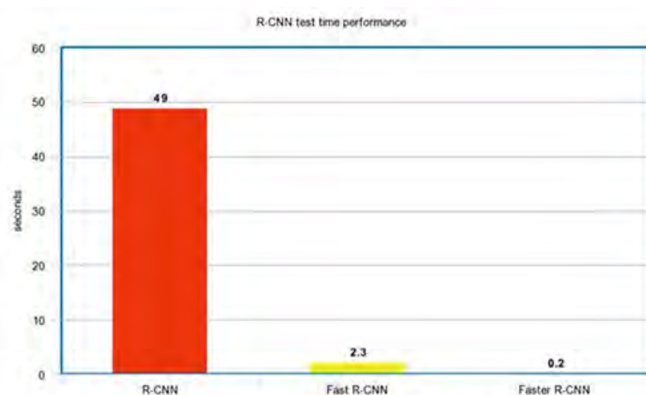


FIGURA 4.29. Comparativa entre redes R-CNN

4.4.6.4. *Mask* R-CNN

Esta red, definida como conceptualmente simple, flexible y con un marco general para segmentación de imágenes, se presenta como una evolución de *Faster* R-CNN añadiendo sólo un pequeño incremento de cálculo[42]. Mientras que *Faster* R-CNN tiene dos salidas para cada propuesta de objeto: una etiqueta de clase y una caja envolvente; esta red añade una tercera salida que es la máscara del objeto.

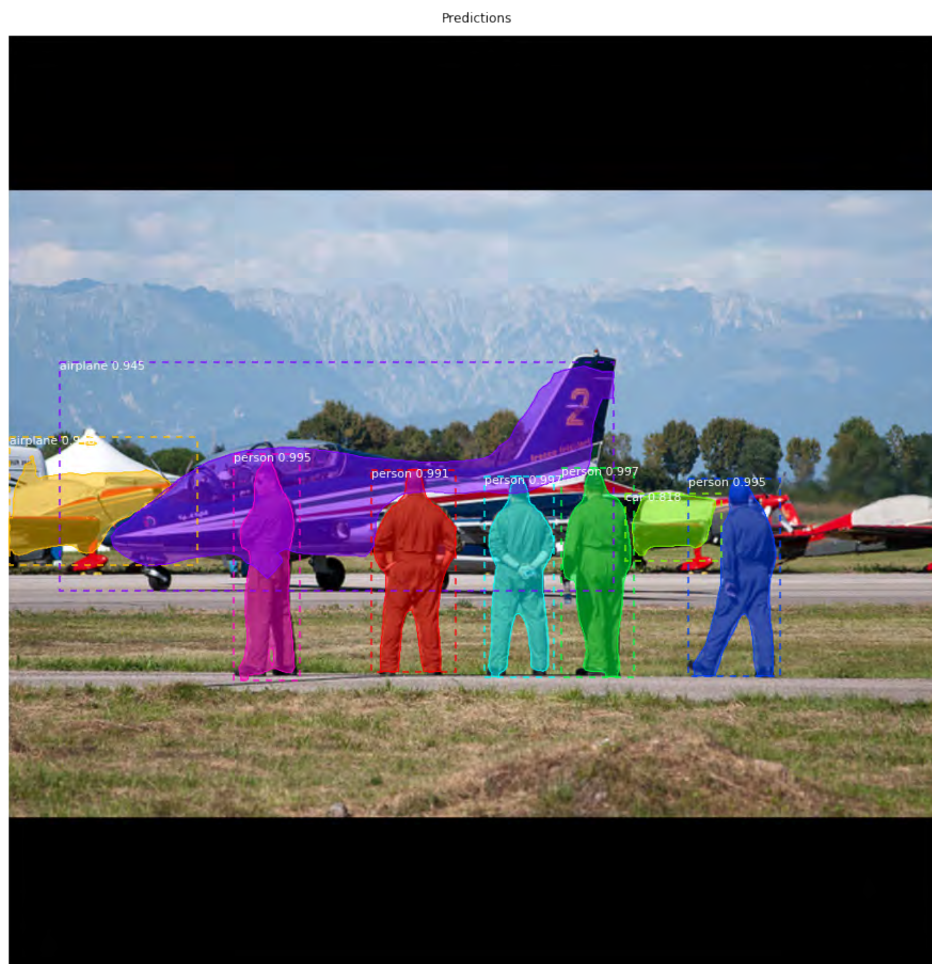


FIGURA 4.30. Salida de *Mask* R-CNN

Siendo la base del desarrollo aplicado de esta memoria, este tipo de redes se estudiarán de forma más detalla en los siguientes capítulos.

4.4.7. *Generative Adversarial Networks* (2014)

Supongamos que tenemos una CNN que funciona bien con datos de ImageNet. Tomando una imagen de muestra y aplicando una perturbación de manera que se maximice el error de predicción tenemos una imagen que aparentemente es igual, pero el objeto cambia de categoría en la predicción. Desde un punto de vista muy simplificado, los ejemplos son básicamente imágenes que engañan a ConvNets [43].

FIGURA 4.31. Flujo de trabajo de una GAN

Este tipo de redes son importantes debido a que el discriminador está ahora alerta frente a las representaciones internas de los datos dado que ha sido entrenado para entender las diferencias entre imágenes reales y las creadas artificialmente. Por tanto puede usarse como un extractor de características que puedes usar en una CNN.

4.4.8. Generación de descripción de imágenes (2014)

Si combinamos una CNN con una RNN[9] obtenemos para una imagen una descripción de los elementos que aparecen en ella [44]. Básicamente el modelo toma una imagen e incluye una descripción de cada uno de los objetos encontrados.

Este tipo de redes son muy importantes debido a que abre la puerta para nuevas ideas de combinación en detección de imágenes y uso común del lenguaje para describir las situaciones, así

[9]Por sus siglas en inglés Recursive Neural Network

como para hacer los modelos más inteligentes cuando se enfrentan a tareas que interrelacionan diferentes campos.

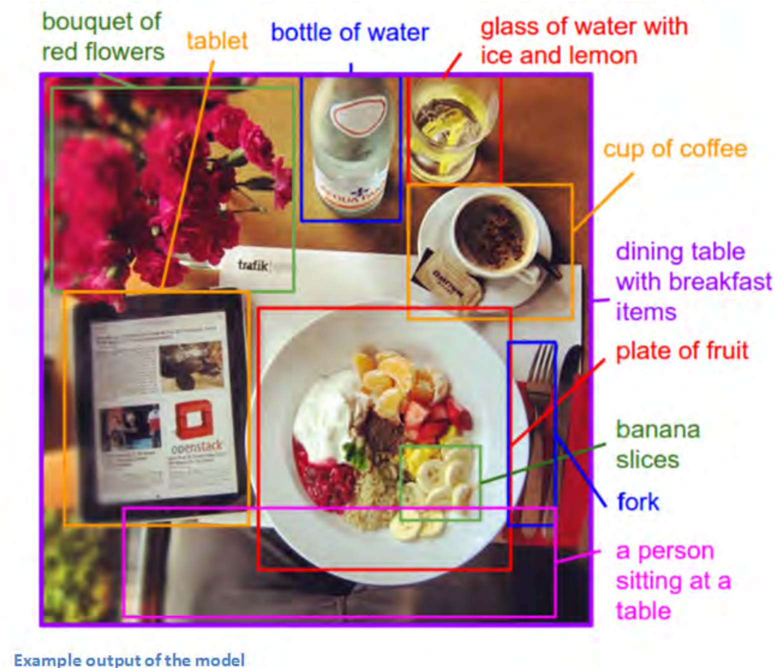


FIGURA 4.32. Detección de imágenes con descripción

4.4.9. Redes de transformación espacial (2015)

En estas redes la principal novedad es la introducción del bloque del cuál toman el nombre, el módulo de transformación espacial [45]. La idea básica es que este módulo modifica y transforma la imagen de entrada para que el resto de capas siguientes hagan de manera más sencilla la clasificación. En vez de variar la arquitectura de la CNN en sí misma a los autores les preocupaba la idea de realizar cambios sobre la imagen antes de ser transmitida a las capas convolucionales.

En las CNN tradicionales, si se quiere hacer el modelo invariante frente a imágenes con diferentes escalas y rotaciones, se necesitan muchos ejemplos de entrenamiento para un aprendizaje correcto.

El modelo consta básicamente de:

- Una red de localización toma el volumen de entrada y tiene por salida los parámetros de la transformación espacial que debe ser aplicada. Estos parámetros, o theta, pueden tener 6 dimensiones para una transformación afín[10].

[10] En geometría una transformación afín entre dos espacios afines consiste en una transformación lineal seguida de una traslación, esto es, $x \to Ax + b$

- La generación de la rejilla de muestreo es el resultado de trasladar la rejilla regular con la transformación afín creada anteriormente.
- Un bloque de muestreo cuya función es realizar la traslación del mapa de entrada.

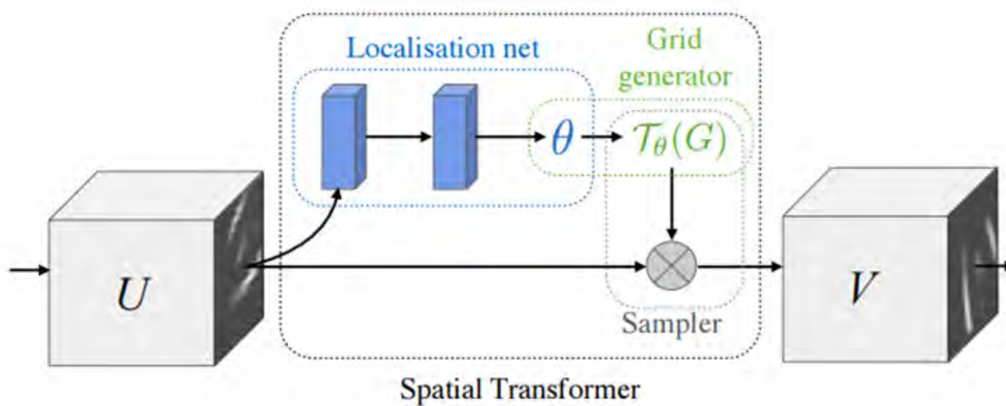


FIGURA 4.33. Módulo de transformación espacial

Este módulo se puede insertar dentro de una CNN en cualquier lugar ayudando a la red a aprender cómo transformar el mapa de características de una manera que minimice la función de costes durante el entrenamiento.

La mayor importancia de esta mejora de las CNN viene del hecho que demuestra que no es necesario hacer cambios drásticos para obtener buenos resultados, no es necesario crear un nuevo ResNet para obtener mejoras. Estos bloques desarrollan la simple idea de hacer una transformación afín a la imagen de entrada de forma que ayude al modelo a ganar invarianza frente a translaciones, rotaciones y cambios de escala.

Capítulo 5

Desarrollo aplicado de aprendizaje automático

En este bloque se van a repasar los conceptos de aplicación de una red neuronal a un problema práctico. Durante los capítulos anteriores se ha ido introduciendo al lector, de una forma leve debido a la ingente cantidad de información al respecto de las redes neuronales, en los conceptos más importantes así como arquitecturas y aplicaciones actuales. Con esta base construida ya es posible abordar un problema práctico de detección, clasificación y etiquetado de objetos en imágenes mediante aprendizaje automático.

En este caso, el modelo de red neuronal empleado será *Mask* R-CNN ya comentado en 4.4.6.4. Se ha elegido este modelo como base debido a que la información disponible en la red es muy abundante en ayuda y tutorización para utilizar y ampliar este modelo. Además, dado que este modelo se ha incluido en Detectron [47], es un buen punto de partida para esta memoria y posibles ampliaciones en el futuro.

La estructura de este capítulo se centrará en la descripción primero del problema a resolver, que en este caso será la detección de diferentes objetos en imágenes o vídeos para luego extraer información complementaria como aviso de señal, detección de personas en la vía o aviso sobre eventos peligrosos. En función de si el modelo elegido detecta todos los objetos buscados se podrá usar *"as is"* o habrá que hacer un entrenamiento adicional para añadir lo que de base no es detectado. Este modelo o red será también analizado en función de sus prestaciones en detección para así dar una medida estadística de su precisión. Por último, las imágenes procesadas y con objetos etiquetados serán analizadas mediante un post procesado para así extraer meta-información que permita a un hipotético usuario poder extraer acciones sobre lo que la cámara está visualizando. Por ejemplo, si una cámara frontal de un vehículo de conducción autónoma detecta un peatón frente a él, haría frenar al vehículo o cambiar su trayectoria para no atropellar al peatón.

5.1. Mask R-CNN para detección de objetos y segmentación

Existen diferentes implementaciones de esta red neuronal brevemente descrita en 4.4.6.4. En esta memoria se han tratado dos de ellas como punto de partida y clonadas desde su respectivo repositorio Git. Dichas implementaciones son las proporcionadas por los enlaces de Karol Majek y matterport. Ambas implementaciones están realizadas mediante Python 3, Keras y TensorFlow. El modelo genera cajas y máscaras de segmentación para cada instancia del objeto en la imagen. Está basada en una FPN[1] y una base ResNet 101.

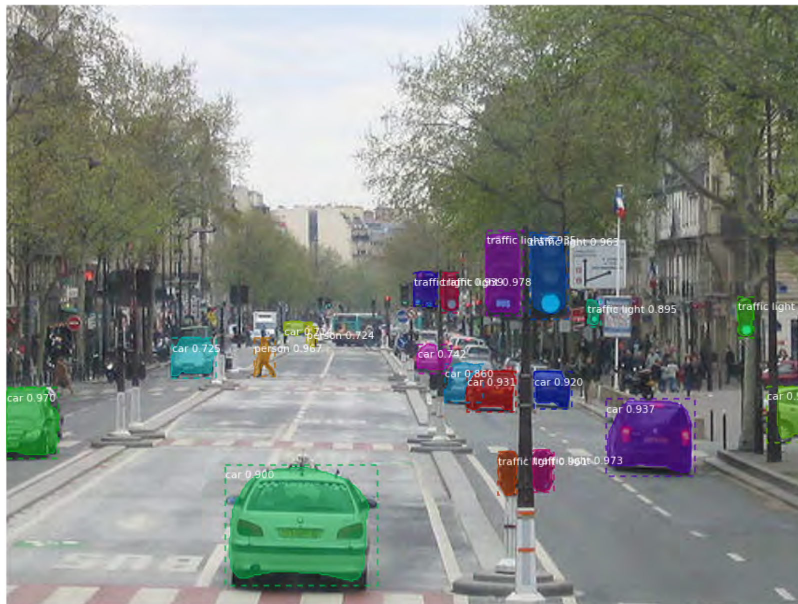


FIGURA 5.1. Ejemplo de resultado Mask R-CNN

El modelo disponible en el respositorio github incluye diversos archivos y funciones útiles para el trabajo con esta red:

- Código fuente de la red construida bajo FPN y ResNet101.
- Código de entrenamiento para MS COCO.
- Pesos pre-entrenados para MS COCO.
- Archivos jupyter para visualizar la detección en cada etapa.
- Clase *ParallelModel* para entrenamiento multi-GPU.
- Evaluación con métricas MS COCO.

[1]Por sus siglas en inglés Feature Pyramid Network

- Ejemplo de entrenamiento con conjunto de imágenes custom.

Todo el código generado está incluido en el anexo D. Los siguientes apartados de la memoria simplemente pretenden recoger en la misma un resumen de la información que se incluye en el repositorio de Karol Majek. Para una vista más detallada del trabajo realizado se puede consultar el material propio en 5.2.

5.1.1. Detección paso a paso

Dentro del repositorio github existen tres diarios jupyter que permiten la visualización paso a paso del modelo para inspeccionar la salida al final de cada punto. Por ejemplo, extraídas directamente de la portada del repositorio, estarían estas fases definidas:

1. Anclajes ordenados y filtrado. Permite la visualización de cada paso en la primera fase y muestra tanto los refinamientos como un prefiltrado de los objetos.

FIGURA 5.2. Mask R-CNN: filtrado

2. Refinado del marco. Incluye las cajas finales sobre el objeto detectado así como el refinado aplicado a los mismos.

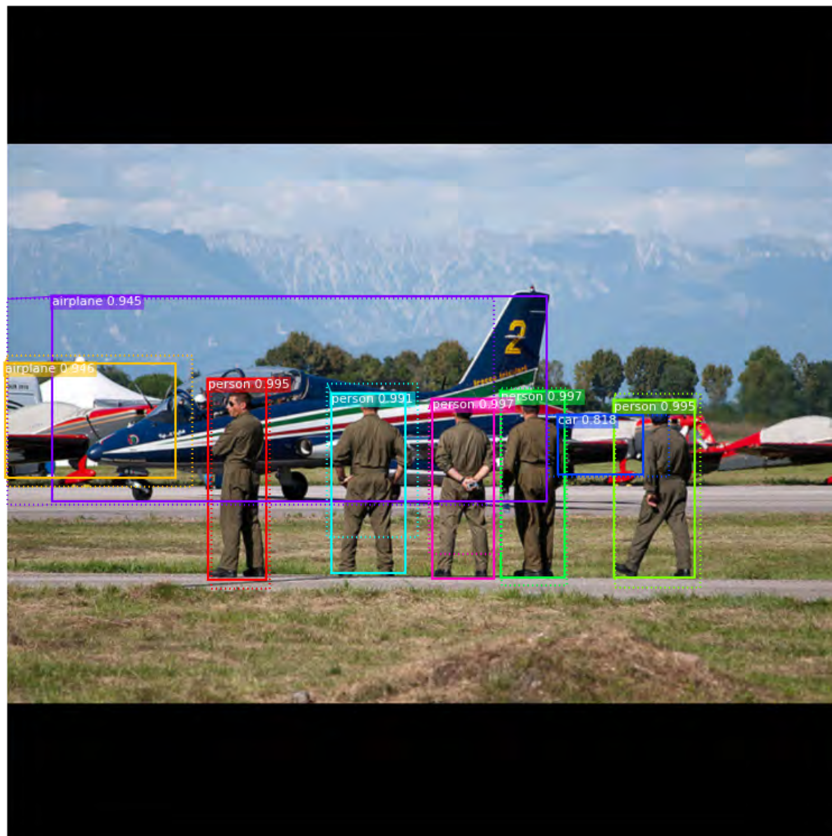


FIGURA 5.3. Mask R-CNN: refinado

3. Generación de máscaras. Genera las máscaras y las escalas, posicionando la imagen en su posición correcta.

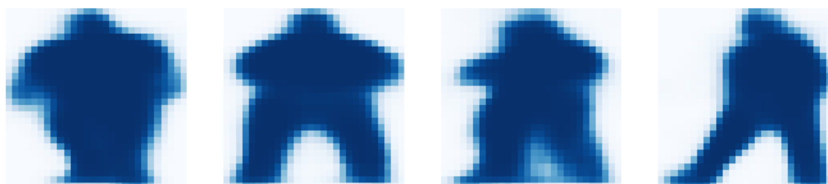


FIGURA 5.4. Mask R-CNN: máscaras

4. Activación de capas. Normalmente ayuda a inspeccionar las activaciones en diferentes capas a fin de reconocer patrones problemáticos (ceros o ruido blanco).

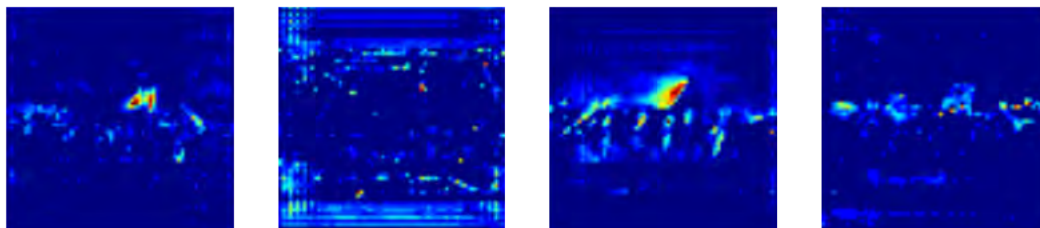


FIGURA 5.5. Mask R-CNN: activación

5. Histogramas de los pesos. Otra herramienta de depurado es la inspección de los pesos.

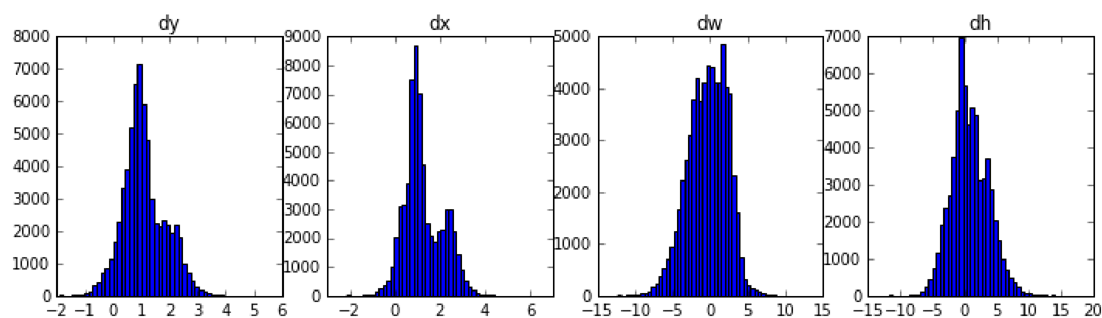


FIGURA 5.6. Mask R-CNN: histogramas

6. Registro a TensorBoard. Otra potente herramienta de depurado y visualización. El modelo está configurado para registrar las pérdidas y guardar los pesos al final de cada epoch.

FIGURA 5.7. Mask R-CNN: Tensorboard

7. Composición de las diferentes piezas de cara al resultado final.

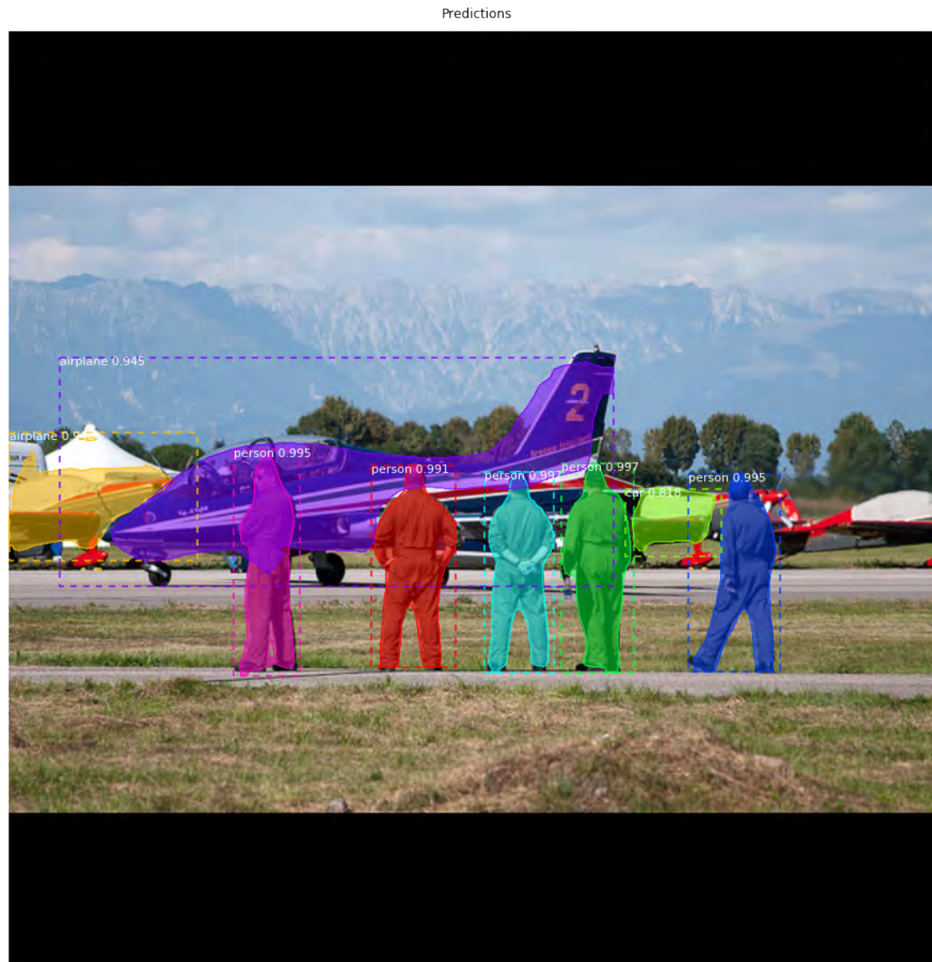


FIGURA 5.8. Mask R-CNN: resultado final

5.1.2. Entrenamiento de Mask R-CNN

El repositorio usado incluye un conjunto de pesos pre-entrenados para hacer más sencillo el comienzo. Estos pesos se pueden incorporar como punto de partida a fin de entrenar cualquier variación sobre conjuntos de datos propios en la red.

La parte práctica de aplicación de la red se comenzó usando los pesos pre-entrenados pero añadiendo nuevos vectores de entrenamiento mediante los conjuntos de imágenes COCO a fin de poder añadir más clases de objetos y comprobar el ajuste de la red ya entrenada sobre ellos. El conjunto de pesos inicial se había hecho sobre imágenes COCO de años anteriores, mientras que el refinamiento de los mismos ya se ha hecho sobre el conjunto de 2017.

En las Figuras 5.9(a) y 5.9(b) se puede comprobar cómo un nuevo entrenamiento partiendo de 0 produce resultados semejantes para las clases preeexistentes en COCO. Si quisiéramos

añadir una clase nueva, se podría usar el mismo patrón de entrenamiento para añadirla a la lista completa.

FIGURA 5.9. La imagen de la izquierda es el resultado con los pesos preentrenados 5.9(a), la imagen de la derecha es un nuevo entrenamiento 5.9(b)

Se usarán estos métodos de entrenamiento cuando se quieran añadir clases al reconocimiento de COCO o nuevas imágenes con los nuevos objetos a detectar. En particular esta memoria se ha realizado con un nuevo entrenamiento sobre el conjunto de imágenes de COCO2017 frente al conjunto de partida proporcionado en el repositorio, más antiguo.

5.2. Aplicación de Mask RCNN

Una vez revisado el repositorio e interiorizadas todas las partes que lo componen a fin de poder trabajar de forma cómoda con el proyecto, se puede comenzar a realizar entrenamientos de la red a fin de adaptarla a los conjuntos de imágenes deseados. Esta sección de la memoria explora varios apartados tales como:

- Revisión del conjunto de datos.
- Entrenamiento de los pesos.
- Revisión del modelo generado.
- Revisión de los pesos.

Todos ellos son modificaciones del repositorio inicial adaptados para los conjuntos objetivo y con ligeros cambios para que sean compatibles tanto con jupyter notebook como con un terminal básico de Anaconda/Python. Todos los archivos de código se han añadido en el anexo D para su consulta y posible referencia.

5.2.1. Revisión del conjunto de datos

La revisión o inspección del conjunto de datos se ha realizado con una modificación del script de Jupyter llamado *Inspect Training Data* que se puede encontrar en el repositorio *Mask RCNN*. Dicha modificación simplemente adapta el script al conjunto de datos empleado y descargado en el ordenador. Como se comentó antes, todo el conjunto de datos de entrenamiento, validación y comprobación es COCO2017.

Se ha adjuntado una copia de este script en D.1. A continuación presentamos los resultados más importantes de los pasos contenidos en este script.

Tras las importaciones iniciales se carga la configuración deseada, en este caso la correspondiente al conjunto de datos COCO.

```
1  # MS COCO Dataset
2  import coco
3  config = coco.CocoConfig()
4  COCO_DIR = "../../dataset"  # TODO: enter value here
5
6  # Load dataset
7  if config.NAME == 'shapes':
8  dataset = shapes.ShapesDataset()
9  dataset.load_shapes(500, config.IMAGE_SHAPE[0], config.IMAGE_SHAPE[1])
10 elif config.NAME == "coco":
11 dataset = coco.CocoDataset()
12 dataset.load_coco(COCO_DIR, "train")
13
```

```
14 # Must call before using the dataset
15 dataset.prepare()
16
17 print("Image Count: {}".format(len(dataset.image_ids)))
18 print("Class Count: {}".format(dataset.num_classes))
19 for i, info in enumerate(dataset.class_info):
20 print("{:3}. {:50}".format(i, info['name']))
```

Para un número aleatorio de muestras del conjunto se pueden comprobar tanto las imágenes como las máscaras también contenidas en COCO.

```
1 # Load and display random samples
2 image_ids = np.random.choice(dataset.image_ids, 4)
3 for image_id in image_ids:
4 image = dataset.load_image(image_id)
5 mask, class_ids = dataset.load_mask(image_id)
6 visualize.display_top_masks(image, mask, class_ids, dataset.class_names)
```

FIGURA 5.10. Imagen de COCO y sus máscaras, para cada tipo de objeto incluido en la imagen se han añadido las máscaras de cada tipo

A la hora de dibujar las cajas que envuelven cada objeto, en vez de trazarlas a partir de la información proporcionada por el conjunto de datos, se calcula a partir de las máscaras superpuestas a cada objeto. En este ejemplo de imagen es la número 252525 de COCO mostrada en la Figura 5.11.

```
1 # Load random image and mask.
2 image_id = random.choice(dataset.image_ids)
3 image = dataset.load_image(image_id)
4 mask, class_ids = dataset.load_mask(image_id)
5 # Compute Bounding box
6 bbox = utils.extract_bboxes(mask)
7
8 # Display image and additional stats
9 print("image_id ", image_id, dataset.image_reference(image_id))
10 log("image", image)
11 log("mask", mask)
12 log("class_ids", class_ids)
```

```
log("bbox", bbox)
# Display image and instances
visualize.display_instances(image, bbox, mask, class_ids, dataset.class_names)
```

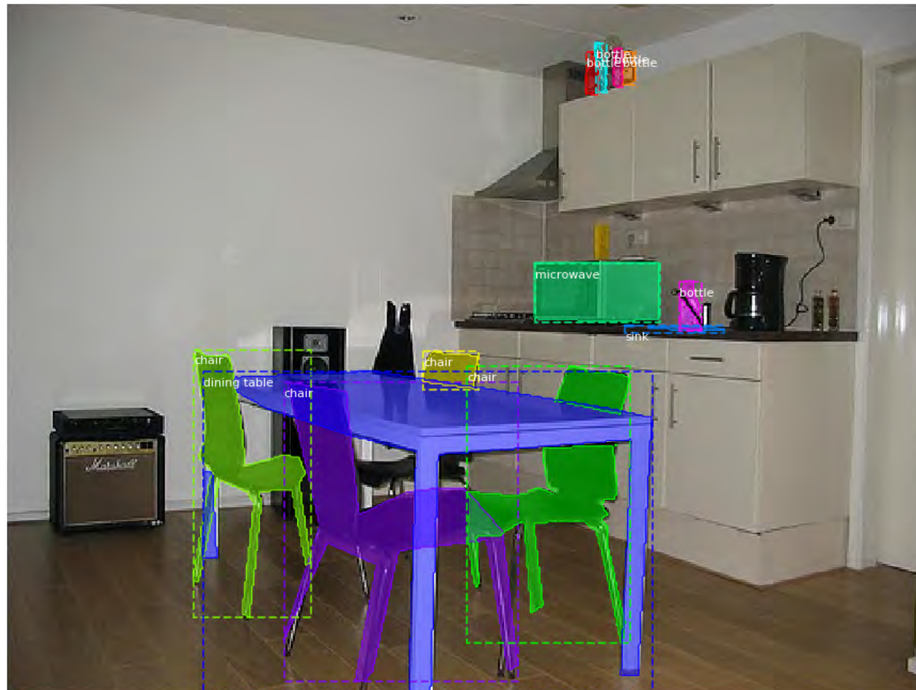


FIGURA 5.11. Imagen de COCO con máscaras y cajas en objetos, se han dibujado las cajas en función de la envolvente del objeto

Adicionalmente las imágenes se reescalan y se calculan máscaras más pequeñas adaptadas a cada objeto. Además, si la imagen no es cuadrada, se añade un marco tanto en los lados como arriba y debajo de la imagen.

Un punto importante del procesado de imágenes es el anclaje de las cajas. Este anclaje básicamente consta en la agrupación de cajas que probablemente contengan algún objeto como se puede observar en la Figura 5.12. El orden de estos anclajes es importante, se debe usar el mismo orden tanto durante el entrenamiento como en la predicción y además debe coincidir durante la ejecución de la convolución.

```
# Generate Anchors
anchors = utils.generate_pyramid_anchors(config.RPN_ANCHOR_SCALES,
config.RPN_ANCHOR_RATIOS,
config.BACKBONE_SHAPES,
config.BACKBONE_STRIDES,
config.RPN_ANCHOR_STRIDE)
```

```
7
8 # Print summary of anchors
9 num_levels = len(config.BACKBONE_SHAPES)
10 anchors_per_cell = len(config.RPN_ANCHOR_RATIOS)
11 print("Count: ", anchors.shape[0])
12 print("Scales: ", config.RPN_ANCHOR_SCALES)
13 print("ratios: ", config.RPN_ANCHOR_RATIOS)
14 print("Anchors per Cell: ", anchors_per_cell)
15 print("Levels: ", num_levels)
16 anchors_per_level = []
17 for l in range(num_levels):
18 num_cells = config.BACKBONE_SHAPES[l][0] * config.BACKBONE_SHAPES[l][1]
19 anchors_per_level.append(anchors_per_cell * num_cells // config.RPN_ANCHOR_STRIDE**2)
20 print("Anchors in Level {}: {}".format(l, anchors_per_level[l]))
```

```
Count:  65472
Scales:  (32, 64, 128, 256, 512)
ratios:  [0.5, 1, 2]
Anchors per Cell:  3
Levels:  5
Anchors in Level 0: 49152
Anchors in Level 1: 12288
Anchors in Level 2: 3072
Anchors in Level 3: 768
Anchors in Level 4: 192
```

```
1 ## Visualize anchors of one cell at the center of the feature map of a specific level
2
3 # Load and draw random image
4 image_id = np.random.choice(dataset.image_ids, 1)[0]
5 image, image_meta, _, _ = modellib.load_image_gt(dataset, config, image_id)
6 fig, ax = plt.subplots(1, figsize=(10, 10))
7 ax.imshow(image)
8 levels = len(config.BACKBONE_SHAPES)
9
10 for level in range(levels):
11 colors = visualize.random_colors(levels)
12 # Compute the index of the anchors at the center of the image
13 level_start = sum(anchors_per_level[:level]) # sum of anchors of previous levels
14 level_anchors = anchors[level_start:level_start+anchors_per_level[level]]
15 print("Level {}. Anchors: {:6}  Feature map Shape: {}".format(level, level_anchors.shape
    [0],
16 config.BACKBONE_SHAPES[level]))
17 center_cell = config.BACKBONE_SHAPES[level] // 2
18 center_cell_index = (center_cell[0] * config.BACKBONE_SHAPES[level][1] + center_cell[1])
```

```
19 level_center = center_cell_index * anchors_per_cell
20 center_anchor = anchors_per_cell * (
21 (center_cell[0] * config.BACKBONE_SHAPES[level][1] / config.RPN_ANCHOR_STRIDE**2) \
22 + center_cell[1] / config.RPN_ANCHOR_STRIDE)
23 level_center = int(center_anchor)
24
25 # Draw anchors. Brightness show the order in the array, dark to bright.
26 for i, rect in enumerate(level_anchors[level_center:level_center+anchors_per_cell]):
27 y1, x1, y2, x2 = rect
28 p = patches.Rectangle((x1, y1), x2-x1, y2-y1, linewidth=2, facecolor='none',
29 edgecolor=(i+1)*np.array(colors[level]) / anchors_per_cell)
30 ax.add_patch(p)
```

```
Level 0. Anchors:  49152  Feature map Shape: [256 256]
Level 1. Anchors:  12288  Feature map Shape: [128 128]
Level 2. Anchors:   3072  Feature map Shape: [64 64]
Level 3. Anchors:    768  Feature map Shape: [32 32]
Level 4. Anchors:    192  Feature map Shape: [16 16]
```

La siguiente imagen muestra los anclajes así como el reescalado y adición de marcos en negro a izquierda y derecha.

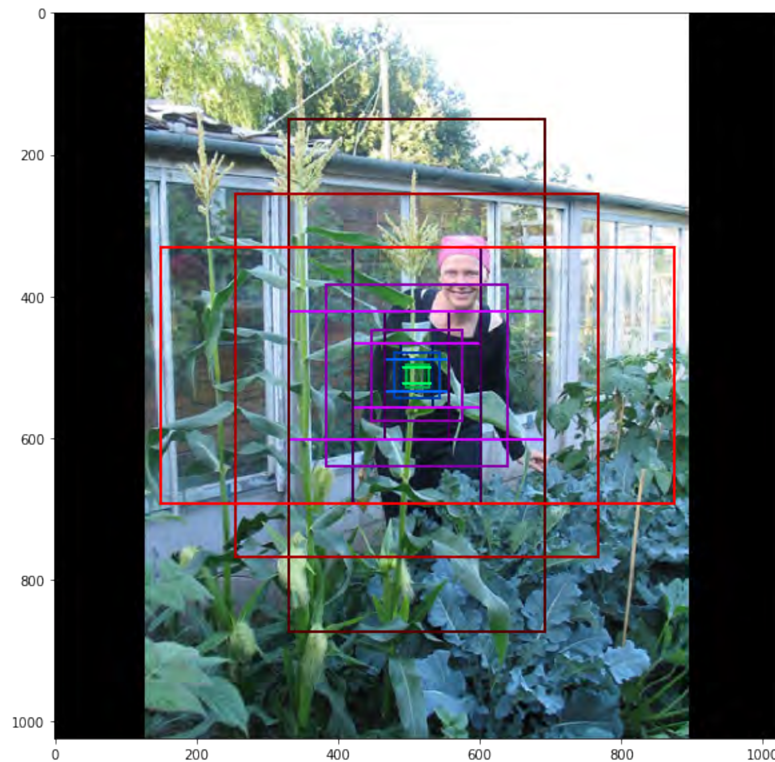


FIGURA 5.12. Anclaje del objeto en una imagen de COCO

La parte final del script genera tanto los anclajes como las cajas que envuelven cada objeto y discierne entre aquellos elementos que intervienen en el entrenamiento como los que no, siendo capaz de generar diferentes imágenes con anclajes positivos, negativos y neutros (no intervienen en el entrenamiento). Como este fragmento de código es muy extenso, se deja para referencia en D.1.

(a) (b) (c)

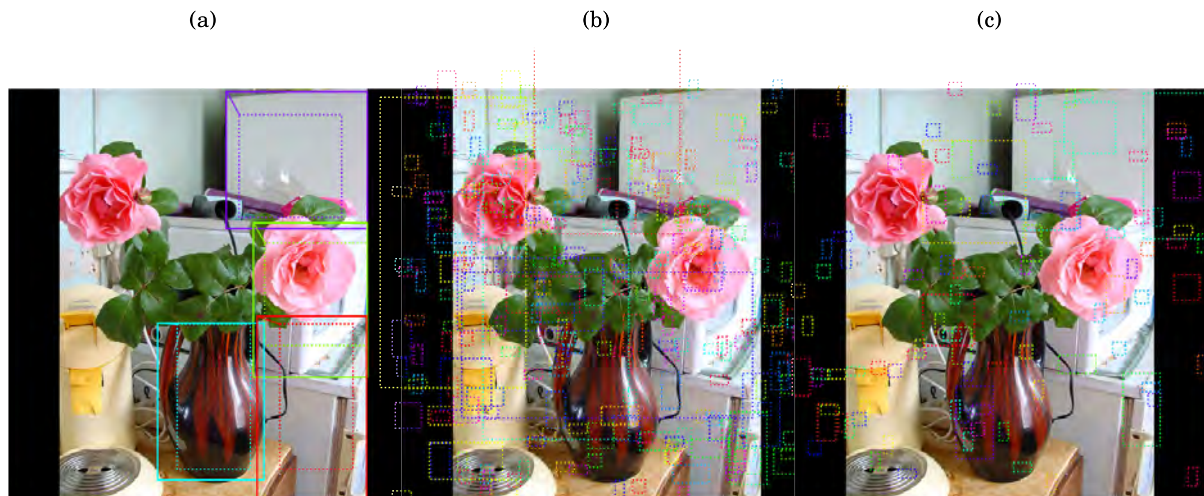


FIGURA 5.13. Comparación en una imagen con anclajes positivos 5.13(a), negativos 5.13(b) y neutros 5.13(c)

Finalmente se calcula el número de Regiones de Interés o ROIs[2] tanto positivas como negativas.

```
if random_rois:
# Class aware bboxes
bbox_specific = mrcnn_bbox[b, np.arange(mrcnn_bbox.shape[1]), mrcnn_class_ids[b], :]

# Refined ROIs
refined_rois = utils.apply_box_deltas(rois[b].astype(np.float32), bbox_specific[:,:4] *
    config.BBOX_STD_DEV)

# Class aware masks
mask_specific = mrcnn_mask[b, np.arange(mrcnn_mask.shape[1]), :, :, mrcnn_class_ids[b]]

visualize.draw_rois(sample_image, rois[b], refined_rois, mask_specific, mrcnn_class_ids[b
    ], dataset.class_names)

# Any repeated ROIs?
```

[2]Por sus siglas en inglés Region of Interest

```
14 rows = np.ascontiguousarray(rois[b]).view(np.dtype((np.void, rois.dtype.itemsize * rois.
       shape[-1])))
15 _, idx = np.unique(rows, return_index=True)
16 print("Unique ROIs: {} out of {}".format(len(idx), rois.shape[1]))
```

```
Positive ROIs:  42
Negative ROIs:  86
Positive Ratio: 0.33
Unique ROIs: 128 out of 128
```

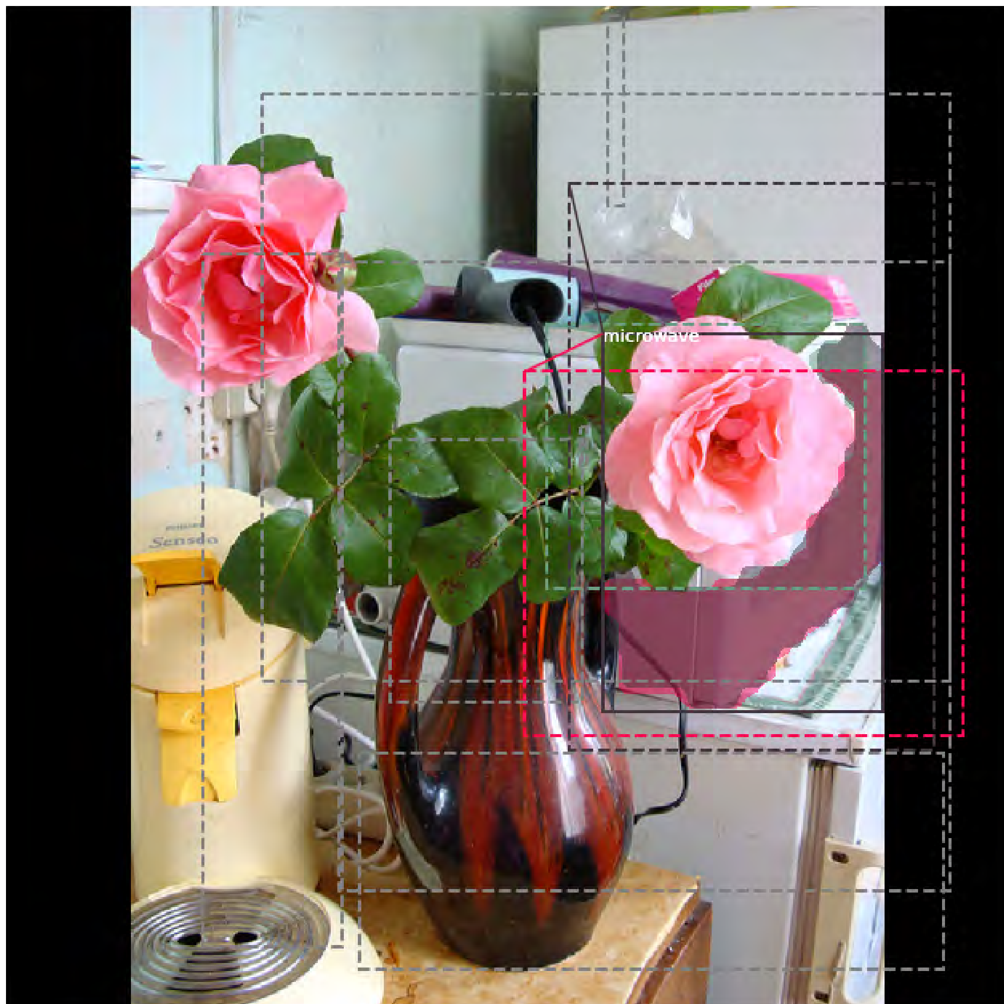


FIGURA 5.14. ROIs de una imagen COCO

5.2.2. Entrenamiento de los pesos

Aunque existe un modelo de script en formato compatible con Jupyter, para el entrenamiento detallado de los pesos del modelo se ha utilizado un script nativo Python, más ligero de utilizar y sin depender del navegador. De esta manera se ha ganado eficiencia a costa de perder visibilidad. Para los detalles de esta sección se ha procesado el script Jupyter incluido en el repositorio con ligeras modificaciones pero para el entrenamiento intensivo se usó el incluido en D.2.

Por otro lado, los scripts de entrenamiento son una modificación del que se entregaba con el repositorio Mask RCNN modificado de acuerdo a los archivos *coco.py*.

Dentro de este script para Jupyter se han incluido también la comparación entre los pesos original del repositorios y los que se puedan obtener como fruto del entrenamiento a fin de comparar cualitativamente la adaptación de uno u otro al conjunto COCO2017.

Una vez realizado todo el entrenamiento de los pesos se puede realizar la comprobación del modelo para una imagen aleatoria del conjunto de validación.

FIGURA 5.15. Imagen original 5.15(a) y la resultante con máscaras 5.15(b)

De una manera breve se definen los siguientes parámetros que miden la calidad de la salida de la red neuronal:

- Intersección sobre Unión, en adelante IoU[3], es el cociente entre el área que pertenece a la intersección entre la predicción y la realidad y en el denominador estaría el área suma de la predicción y la realidad.
- La precisión promediada, en adelante AP[4] se calcula para aquellas predicciones consideradas correctas, esto es, que tienen un valor de IoU mayor que un umbral. Dicho de otra manera, la precisión se define como el número de positivos sobre el número de positivos más falsos positivos: $P = \frac{V_p}{V_p+F_p}$.
- La sensibilidad promediada, en adelante AR[5] es el porcentaje promediado de las instancias relevantes que han sido recogidas sobre el total de instancias. Dicho de otra manera, la sensibilidad se define como el número de positivos sobre el número de positivos más falsos negativos: $R = \frac{V_p}{V_p+F_p}$.

La diferencia entre AP y AR es que la AP mide cuántos objetos hemos detectado de forma correcta frente al conjunto de detecciones, mientras que AR mide cuántos objetos hemos detectado de forma correcta frente a todos los objetos posibles.

Finalmente se realiza una comprobación de la calidad de los pesos obtenidos mediante funciones que evalúan la precisión (Averaged Precission) y la sensibilidad promediadas (Averaged Recall) del modelo.

En este caso se obtuvieron los siguientes resultados para un total de 10 imágenes.:

```
DONE (t=0.08s).
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.315
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.569
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.285
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.209
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.336
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.396
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.269
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.321
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.321
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.209
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.342
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.411
Prediction time: 10.019952535629272. Average 1.0019952535629273/image
Total time:  10.24044942855835
```

[3] Por sus siglas en inglés Intersection over Union

[4] Por sus siglas en inglés Averaged Precission

[5] Por sus siglas en inglés Averaged Recall

Con un valor de la media del valor del AP (mAP) para todas las imágenes igual a

```
mAP @ IoU=50:  0.5053373047993296
```

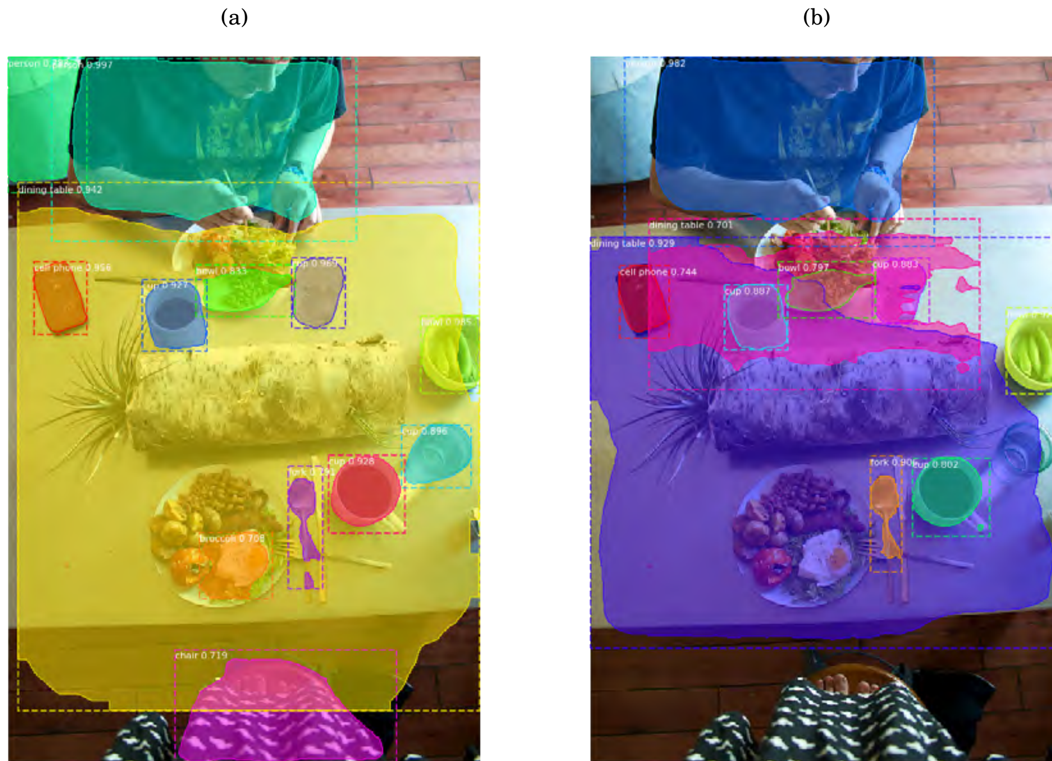


FIGURA 5.16. Detección con los pesos originales 5.16(a) y con los pesos calculados 5.16(b)

5.2.3. Revisión del modelo generado

Otro de los ficheros útiles para trabajar con el modelo Mask R-CNN es el que permite evaluar, depurar y probar dicho modelo. Se trata de una modificación del archivo base del repositorio adaptado a los conjuntos de datos de la memoria.

Una vez entrenados los pesos se podrían repasar con este script a fin de revisar que todo es correcto o si surgen diversos problemas recurrentes que afecten a la potencia final de la red.

Los primeros fragmentos de código corresponden a inicializaciones y configuraciones base. El primer fragmento importante es el que mostraría las clases sobre las que se ha entrenado la red.

```
1 # Build validation dataset
2 if config.NAME == 'shapes':
3 dataset = shapes.ShapesDataset()
4 dataset.load_shapes(500, config.IMAGE_SHAPE[0], config.IMAGE_SHAPE[1])
```

```
5 elif config.NAME == "coco":
6 dataset = coco.CocoDataset()
7 dataset.load_coco(COCO_DIR, "val")
8
9 # Must call before using the dataset
10 dataset.prepare()
11
12 print("Images: {}\nClasses: {}".format(len(dataset.image_ids), dataset.class_names))
```

Con resultado:

```
loading annotations into memory...
Done (t=0.69s)
creating index...
index created!
Images: 4952
Classes: ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
 'bus', 'train' [...],'toothbrush']
```

Se han cargado las 80+1 clases de COCO, siendo la primera clase la correspondiente al fondo de la imagen[6].

La detección de objetos en imágenes no sería diferente de la vista en 5.16(a), mostrando sobre una imagen las cajas y máscaras de cada objeto identificado, así como la puntuación de precisión sobre ellos.

FIGURA 5.17. Imagen original 5.17(a) e imagen con objetos detectados 5.17(b)

[6]Por sus siglás en inglés BackGround

Calculando la precisión de los pesos y su sensibilidad como se indica en A.2:

```
1 # Draw precision-recall curve
2 AP, precisions, recalls, overlaps = utils.compute_ap(gt_bbox[:,:4], gt_bbox[:,4],
3 r['rois'], r['class_ids'], r['scores'])
4 visualize.plot_precision_recall(AP, precisions, recalls)
```

La Figura 5.18 muestra la relación entre la precisión y la sensibilidad para la predicción de objetos. La gráfica que relaciona ambas es una manera de medir el compromiso entre ellas para un límite dado, en este caso de 0.488. Un sistema con una alta sensibilidad pero baja precisión va a detectar muchos objetos pero la mayoría de ellos serán falsos positivos. Por otro lado, un sistema con alta precisión pero baja sensibilidad será el caso totalmente opuesto, detectará muy pocos objetos pero la gran mayoría serán correctos. Un sistema ideal será aquél que detecta muchos objetos con muy pocos errores o falsos positivos. Por tanto, un área grande bajo la curva representa a la vez alta precisión y alta sensibilidad.

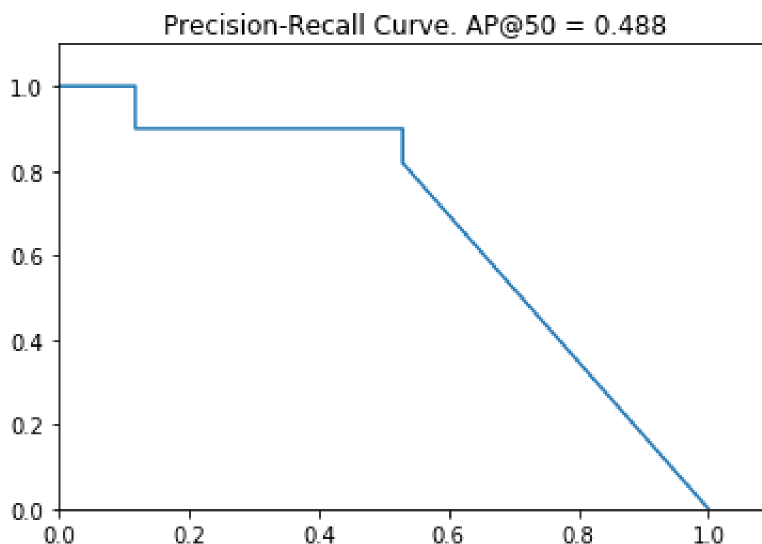


FIGURA 5.18. Curva de sensibilidad y precisión para los pesos calculados

También se puede comprobar la tabla que traza los objetos y sus predicciones:

```
1 # Grid of ground truth objects and their predictions
2 visualize.plot_overlaps(gt_bbox[:, 4], r['class_ids'], r['scores'],
3 overlaps, dataset.class_names)
```

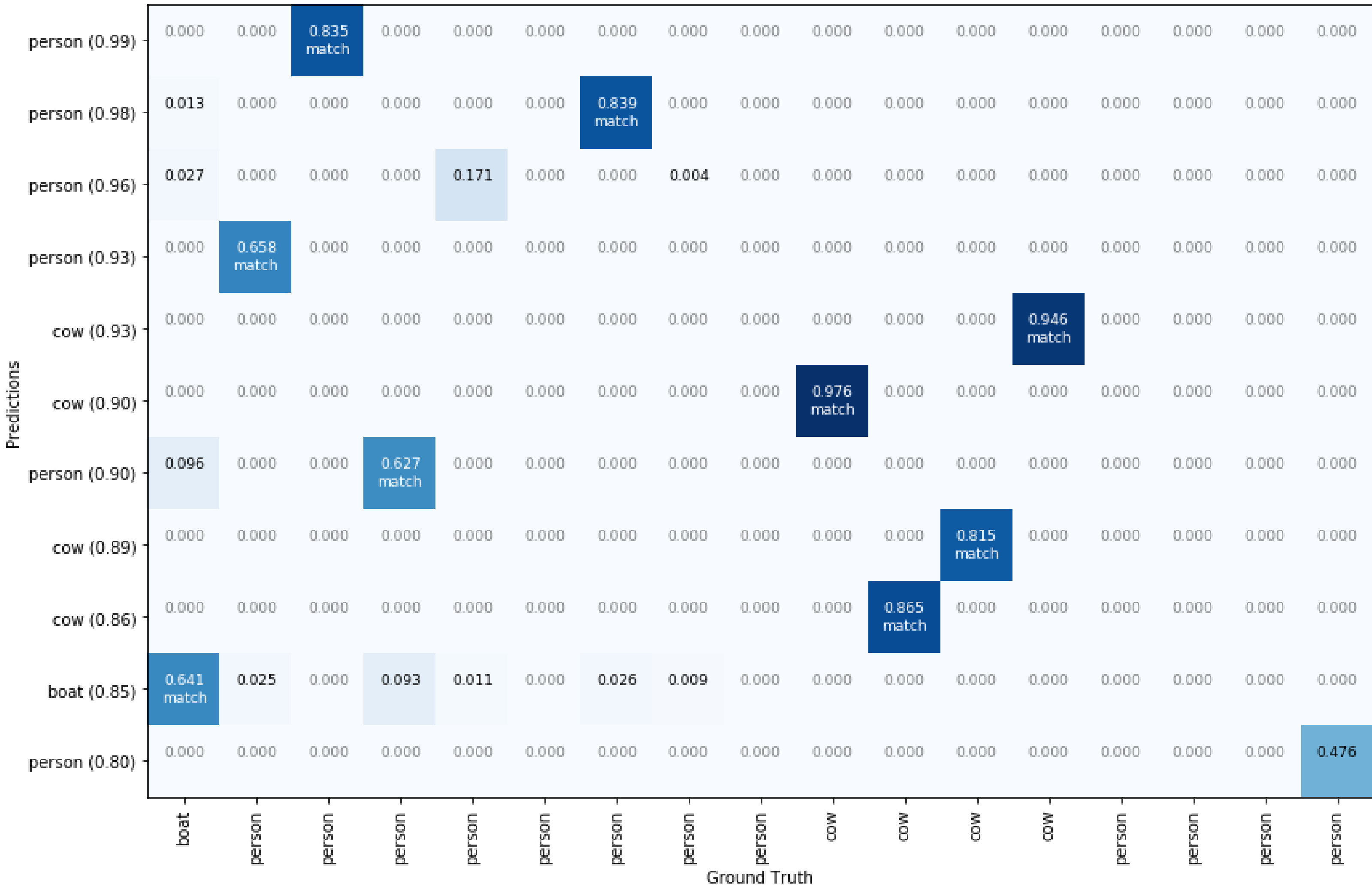


FIGURA 5.19. Tabla de predicciones para los pesos calculados

El cálculo de la precisión promediada según un conjunto de imágenes no diferiría de la que se calculó en el script de entrenamiento.

5.2.3.1. Predicción paso a paso

La red de regiones propuestas o RPN[7] ejecuta un clasificador binario ligero en un conjunto de cajas o anclajes sobre la imagen y devuelve una puntuación objeto/no_objeto.

Los objetivos de la RPN están formados por los valores de entrenamiento para dicha RPN. Para generar los objetivos primero se empieza con una malla de anclajes de forma que cubran totalmente la imagen a diferentes escalas y luego se calcula el IoU de los anclajes con la tabla de verdad del objeto. Anclajes positivos son aquellos con una IoU>=0.7 y anclajes negativos los que no cubren el objeto con al menos una IoU de 0.3. Anclajes con una 0.3<IoU<0.7 se consideran neutros y no son tenidos en cuenta, excluidos por tanto del entrenamiento.

```
1 # Generate RPN trainig targets
2 # target_rpn_match is 1 for positive anchors, -1 for negative anchors
3 # and 0 for neutral anchors.
4 target_rpn_match, target_rpn_bbox = modellib.build_rpn_targets(
5 image.shape, model.anchors, gt_bbox, model.config)
6 log("target_rpn_match", target_rpn_match)
7 log("target_rpn_bbox", target_rpn_bbox)
8
9 positive_anchor_ix = np.where(target_rpn_match[:] == 1)[0]
10 negative_anchor_ix = np.where(target_rpn_match[:] == -1)[0]
11 neutral_anchor_ix = np.where(target_rpn_match[:] == 0)[0]
12 positive_anchors = model.anchors[positive_anchor_ix]
13 negative_anchors = model.anchors[negative_anchor_ix]
14 neutral_anchors = model.anchors[neutral_anchor_ix]
15 log("positive_anchors", positive_anchors)
16 log("negative_anchors", negative_anchors)
17 log("neutral anchors", neutral_anchors)
18
19 # Apply refinement deltas to positive anchors
20 refined_anchors = utils.apply_box_deltas(
21 positive_anchors,
22 target_rpn_bbox[:positive_anchors.shape[0]] * model.config.RPN_BBOX_STD_DEV)
23 log("refined_anchors", refined_anchors, )
24
25 # Display positive anchors before refinement (dotted) and
26 # after refinement (solid).
27 visualize.draw_boxes(image, boxes=positive_anchors, refined_boxes=refined_anchors, ax=
    get_ax())
```

[7] Por sus siglas en inglés Region Proposal Network

Y el resultado sería:

```
target_rpn_match          shape: (65472,)
min:   -1.00000  max:    1.00000
target_rpn_bbox           shape: (256, 4)
min:  -12.04291  max:    4.41942
positive_anchors          shape: (20, 4)
min:  -26.50967  max:  992.00000
negative_anchors          shape: (236, 4)
min: -106.03867  max: 1056.00000
neutral anchors           shape: (65216, 4)
min: -362.03867  max: 1258.03867
refined_anchors           shape: (20, 4)
min:   21.00000  max:  991.00000
```

La ejecución del gráfico de la RPN mostraría las predicciones:

```
# Run RPN sub-graph
pillar = model.keras_model.get_layer("ROI").output  # node to start searching from
rpn = model.run_graph([image], [
("rpn_class", model.keras_model.get_layer("rpn_class").output),
("pre_nms_anchors", model.ancestor(pillar, "ROI/pre_nms_anchors:0")),
("refined_anchors", model.ancestor(pillar, "ROI/refined_anchors:0")),
("refined_anchors_clipped", model.ancestor(pillar, "ROI/refined_anchors_clipped:0")),
("post_nms_anchor_ix", model.ancestor(pillar, "ROI/rpn_non_max_suppression/
    NonMaxSuppressionV2:0")),
("proposals", model.keras_model.get_layer("ROI").output),
])

# Show top anchors by score (before refinement)
limit = 100
sorted_anchor_ids = np.argsort(rpn['rpn_class'][:,:,1].flatten())[::-1]
visualize.draw_boxes(image, boxes=model.anchors[sorted_anchor_ids[:limit]], ax=get_ax())
```

Y el resultado sería:

```
rpn_class                 shape: (1, 65472, 2)
min:    0.00000  max:    1.00000
pre_nms_anchors           shape: (1, 10000, 4)
min: -362.03867  max: 1258.03870
refined_anchors           shape: (1, 10000, 4)
min: -393157.59375  max: 395011.40625
refined_anchors_clipped   shape: (1, 10000, 4)
min:    0.00000  max: 1024.00000
```

```
post_nms_anchor_ix       shape: (1000,)
min:     0.00000  max: 1498.00000
proposals                shape: (1, 1000, 4)
min:     0.00000  max:    1.00000
```

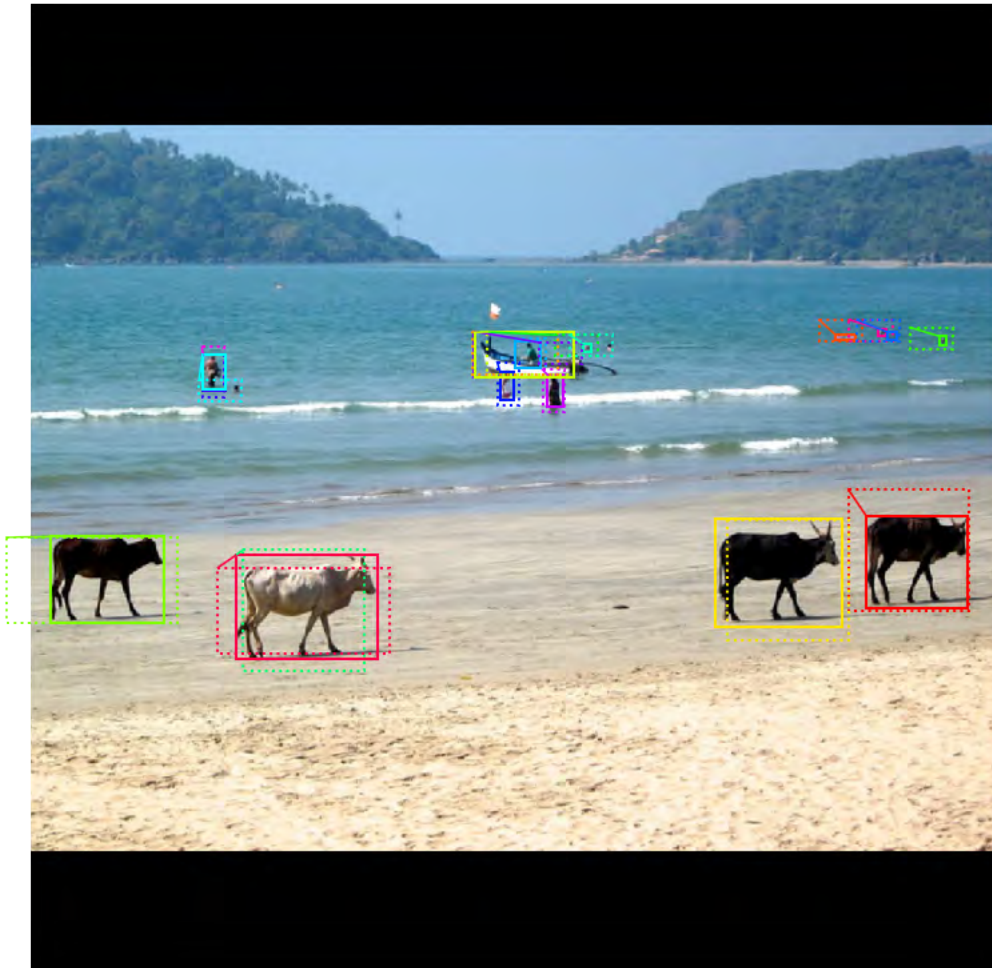


FIGURA 5.20. Anclajes positivos

Tras sucesivos refinados realizados se tendría la imagen final mostrada en la Figura 5.21.

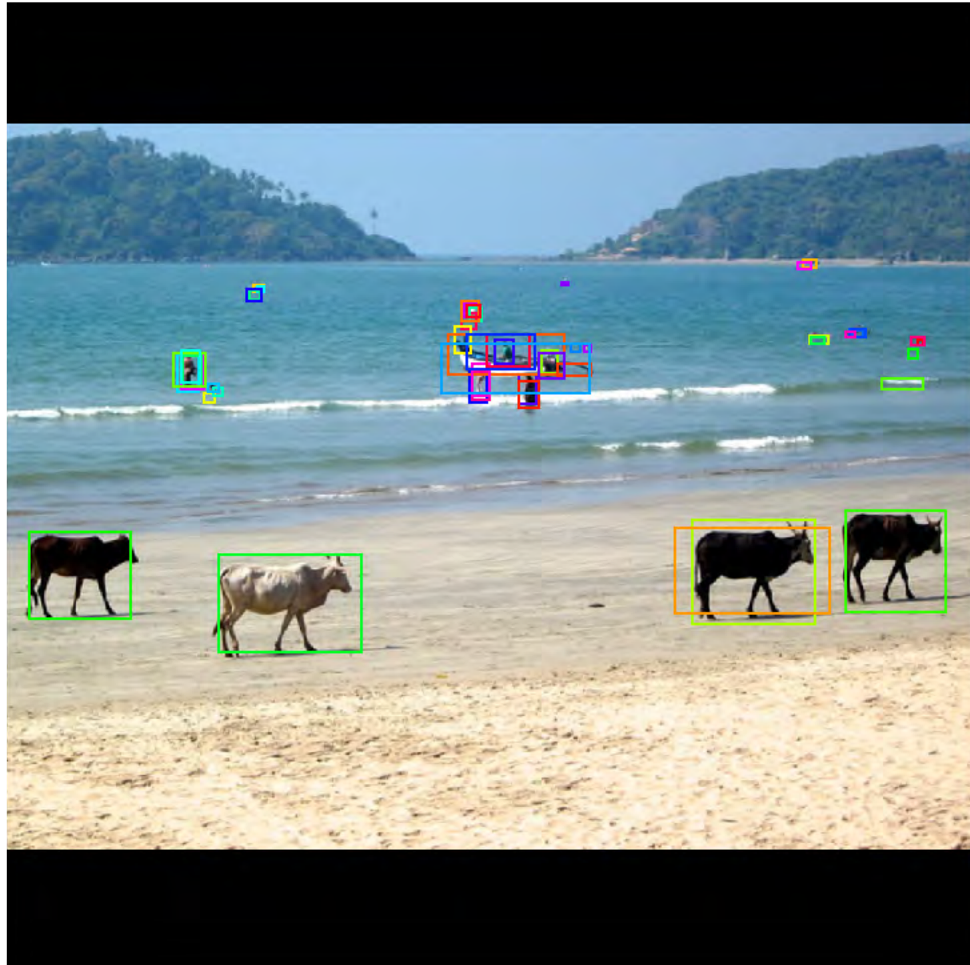


FIGURA 5.21. Propuestas finales de objetos detectados

5.2.3.2. Propuesta de clasificación

Tras las detecciones del apartado anterior el siguiente paso lógico consistiría en clasificar las cajas detectadas que encierran objetos de interés en la imagen.

La salida de este código sería la imagen con cajas alrededor de cada objeto con una puntuación asociada pero sin aparecer todavía las máscaras.

FIGURA 5.22. Propuestas finales de objetos etiquetados

5.2.3.3. Generación de las máscaras

El último paso de la detección es la generación de las máscaras y su colocación encima del objeto ya detectado y etiquetado. La salida sería la imagen totalmente procesada.

FIGURA 5.23. Propuestas finales de máscaras sobre objetos

5.2.3.4. Visualización de las activaciones

Por último, puede ser interesante en ocasiones revisar las activaciones a fin de encontrar patrones extraños o problemas en el procesado de las imágenes.

5.2.4. Revisión de los pesos

Este último script de Jupyter permite revisar los pesos calculados tras el entrenamiento. El código incluido en *inspect_weights.ipynb* revisa las estadísticas de los pesos y dibuja los histogramas.

5.3. Filtro de Kalman

Aunque los Filtros de Kalman [54] no constituyen propiamente una parte de las redes neuronales, por su importancia en la detección y seguimiento de objetos merecen una mención aparte en esta memoria. En la aplicación práctica de la memoria se utilizará un Filtro de Kalman para calcular las posiciones futuras de los objetos detectados y así poder clasificarlos como nuevos en las secuencias o simplemente un desplazamiento de los ya detectados.

El filtro de Kalman es un algoritmo desarrollado por Rudolf E. Kalman en 1960 que sirve para poder identificar el estado oculto (no medible) de un sistema dinámico lineal cuando el sistema está incluso sometido a ruido blanco aditivo. Ya que el Filtro de Kalman es un algoritmo recursivo, éste puede realizarse en tiempo real usando simplemente las nuevas medidas de entrada actuales, el estado calculado previamente y su matriz de incertidumbre, no requiriendo ninguna otra información adicional. Básicamente el objetivo de este filtrado es extraer la información necesaria de una señal, ignorando todo lo demás.

El algoritmo requiere dos tipos de ecuaciones: las que relacionan las variables de estado con las variables observables y las que determinan la estructura temporal de las variables de estado. La calidad de la estimación se medirá con una función de coste o pérdida, como por ejemplo el error cuadrático medio. Por tanto, el objetivo del filtro será minimizar esta función de pérdida.

El algoritmo explicado de forma breve se puede consultar en [55] y [56]. De una forma mas detallada el filtro se diseñaría según [57].

Sea una señal arbitraria descrita como:

$$y_t = a_t x_t + n_t \tag{5.1}$$

donde y_t es la señal temporal observada, x_t es la señal que lleva la información, a_t es el término multiplicativo o de ganancia y n_t es el ruido aditivo. El objetivo del filtrado es obtener una estimación para x_t. Por tanto la diferencia entre la estimación y el valor real será el error del filtrado, el cuál hay que minimizar. Por tanto:

$$f(e_t) = f(x_t - \hat{x}_t) \tag{5.2}$$

Una función de pérdida coherente debe satisfacer las siguientes propiedades[58]:

- $f(e_t) \geqslant 0$ para cualquier e_t
- $f(e_t) = 0$ si $e_t = 0$ que equivale a que $x_t = \hat{x}_t$
- $f(e_t) \geqslant f(h_t)$ si $e_t \geqslant h_t$

Las siguientes funciones de pérdida cumplirían las propiedades anteriores:

- $f_1(e_t) = |x_t - \hat{x}_t|$
- $f_2(e_t) = (x_t - \hat{x}_t)^2$

- $f_3(e_t) = \frac{|x_t - \hat{x}_t|}{x_t}$

Para el detalle del Filtro de Kalmann emplearemos el error cuadrático

$$f(e_t) = (x_t - \hat{x}_t)^2 \tag{5.3}$$

A fin de tener una estimación temporal de la señal de error, se calculará la esperanza de la serie temporal, lo cual dará lugar al cálculo del error cuadrático medio:

$$\begin{aligned} \epsilon(t) &= E(f(e_t)) \\ &= E(e_t^2) \end{aligned} \tag{5.4}$$

Una vez determinado cuál va ser la función de coste empleada para la minimización de la diferencia entre la estimación y el valor real, estamos en posición de deducir las ecuaciones que dan lugar al Filtro de Kalman.

Supongamos que existe un proceso para el cuál queremos determinar el valor de la variable en el tiempo. Sea entonces

$$x_{t+1} = \Phi x_t + w_t \tag{5.5}$$

donde x_t es el vector en el instante t, Φ es la matriz de transición que determina el paso del instante t al instante $t+1$ y por último u_t es el término de error con distribución normal de media nula y covarianza conocida.

Por otro lado, las observaciones de dicha variable se pueden modelar a su vez como

$$z_t = Hx_t + u_t \tag{5.6}$$

donde, en este caso, z_t es el valor de x en un instante t, H es la matriz que relaciona el vector de estado con el de medidas y u_t es el error de medida asociado, ya que H se supone que no posee ruido. Como ocurría en (5.5) el término de error se asume que pertenece a una distribución normal con media nula. De la misma manera se asume que las covarianzas de ambos términos son conocidas y estacionarias en el tiempo con valores

$$Q = E[w_t w_t^T] \tag{5.7}$$

$$R = E[u_t u_t^T] \tag{5.8}$$

Por otro lado, el error cuadrático medio, como se vio en (5.4), se puede expresar de la forma

$$\begin{aligned} P_t &= E[e_t e_t^T] \\ &= E[(x_t - \hat{x}_t)(x_t - \hat{x}_t)^T] \end{aligned} \tag{5.9}$$

Suponiendo que la estimación en el instante anterior $t-1$ para la variable x es conocida y se obtiene haciendo evolucionar el sistema, es posible definir una ecuación para la actualización de las nuevas estimaciones, combinando estimaciones anteriores con nuevos datos. De esta manera

$$\hat{x}_t = \hat{x}'_t + K_t(z_t - H\hat{x}'_t) \tag{5.10}$$

Donde K_t es la llamada Ganancia de Kalman[59]. El término $(z_t - H\hat{x}'_t)$ es el residuo entre la diferencia del valor real del sistema y el valor obtenido por la evolución desde el estado anterior

$$i_t = z_t - H\hat{x}'_t \tag{5.11}$$

Combinando 5.6 junto con 5.10 obtenemos una nueva ecuación de estado para la estimación

$$\hat{x}_t = \hat{x}'_t + K_t(Hx_t + u_t - H\hat{x}'_t) \tag{5.12}$$

Que, a su vez, permite calcular la matriz de covarianzas para el error, P_t, de la siguiente manera

$$\begin{aligned} P_t &= E[(x_t - \hat{x}_t)(x_t - \hat{x}_t)'] \\ &= E[[x_t - [\hat{x}'_t + K_t(Hx_t + u_t - H\hat{x}'_t)]] \\ &\quad [x_t - [\hat{x}'_t + K_t(Hx_t + u_t - H\hat{x}'_t)]]^T] \\ &= E[[(I - K_tH)(x_t - \hat{x}'_t) - K_tu_t] \\ &\quad [(I - K_tH)(x_t - \hat{x}'_t) - K_tu_t]^T] \end{aligned} \tag{5.13}$$

Donde el término $(x_t - \hat{x}'_t)$ es el error en la estimación anterior. Al estar incorrelado con el ruido en la medida, su esperanza se puede reducir a

$$\begin{aligned} P_t &= (I - K_tH)E[(x_t - \hat{x}'_t)(x_t - \hat{x}'_t)^T](I - K_tH) \\ &\quad + K_tE[u_ku_k^T]K_t^T \end{aligned} \tag{5.14}$$

Y, finalmente, agrupando términos

$$P_t = (I - K_tH)P_t'^T(I - K_tH)^T + K_tRK_t^T \tag{5.15}$$

Donde P'_t es la estimación anterior para P_t. Ya se comienza a intuir el carácter recursivo del filtrado, puesto que podemos ir construyendo ecuaciones en base a estados anteriores del sistema.

Operando con 5.15 podemos reescribir la expresión para P_t como

$$P_t = P'_t - K_tHP'_t - P'_tH^TK_t^T + K_t[HP'_tH^T + R]K_t^T \tag{5.16}$$

Según vimos en 5.9, P_t es el error cuadrático medio de la estimación, por lo que si queremos minimizarlo es necesario realizar su derivada e igualarla a 0. Como además buscamos calcular el valor de la ganancia de Kalman K_t para la que se minimiza el error cuadrático medio, la diferenciación se hará con respecto a este parámetro. Además, como P_t es la matriz de covarianzas, minimizar dicha matriz es equivalente a minimizar su traza $Tr(P_t)$, por lo que se calcula la derivada de la traza frente a la ganancia de Kalman, esto es

$$\begin{aligned} \frac{\partial T[P_t]}{\partial K_t} &= -HP'_t - P'_tH^T + 2K_t[HP'_tH^T + R] \\ &= -2(HP'_t)^T + 2K_t(HP'_tH^T + R) \end{aligned} \tag{5.17}$$

Igualando a 0 para hallar el mínimo de la función y resolviendo para K_t se obtiene[8]

$$(HP_t')^T = K_t(HP_t'H^T + R) => K_t = P_t'^T H^T (HP_t'H^T + R)^{-1} \tag{5.18}$$

Por lo que finalmente la ganancia de Kalman vendrá dada por

$$K_t = P_{t-1}H^T(HP_t'H^T + R)^{-1} \tag{5.19}$$

Con el valor de ganancia del filtro que minimiza el error cuadrático medio en la estimación del valor de la señal, podemos actualizar la expresión para P_t obtenida en 5.16 de la siguiente manera

$$\begin{aligned} P_t &= P_t' - P_t'H^T(HP_t'H^T + R)^{-1}HP_t' - P_t'H^T K_t^T \\ &\quad + P_t'H^T(HP_t'H^T + R)^{-1}[HP_t'H^T + R]K_t^T \\ &= P_t' - P_t'H^T(HP_t'H^T + R)^{-1}HP_t' \\ &= (I - K_tH)P_t' \end{aligned} \tag{5.20}$$

La nueva expresión obtenida en 5.20 se corresponde con el valor óptimo de P_t, esto es, con ganancia de Kalman máxima.

Por último, para tener el filtro de Kalman completamente descrito de manera recursiva, es necesario calcular la evolución en el tiempo para P_k'. Las estimaciones del sistema evolucionan en el tiempo según

$$\hat{x}_{t+1}' = \Phi \hat{x}_t \tag{5.21}$$

Por otro lado, el error a priori en la estimación se obtiene por

$$\begin{aligned} e_{t+1}' &= x_{t+1} - \hat{x}_{t+1}' \\ &= (\Phi x_t + w_t) - \Phi \hat{x}_k \\ &= \Phi e_t + w_t \end{aligned} \tag{5.22}$$

Evolucionando a su vez el valor de P_t al instante siguiente $t+1$ tenemos

$$\begin{aligned} P_{t+1}' &= E[e_{t+1}' e_{t+1}'^T] \\ &= E[(\Phi e_t + w_t)(\Phi e_t + w_t)^T] \\ &= E[\Phi e_t(\Phi e_t)^T] + E[w_t w_t^T] \\ &\quad + E[\Phi e_t w_t] + E[(\Phi e_t)^T w_t] + E[\Phi e_t w_t^T] + E[(\Phi e_t)^T w_t^T] \\ &= \Phi P_t \Phi^T + Q \end{aligned} \tag{5.23}$$

[8]Recordemos que $(AB)^t = B^t A^t$

Cabe recordar que los términos e_t y w_t están incorrelados entre sí debido a cómo están definidos. El término e_t es el error hasta el instante t mientras que w_t es el error entre los instantes t y $t+1$. Por esta razón $E[\Phi e_t w_t] = 0$ y se ha podido simplificar de la expresión anterior.

En este punto ya sólo cabe resumir las ecuaciones del filtro de Kalman que determinan cómo evolucionan los estados en el tiempo así como el valor óptimo de ganancia de Kalman. Recopilando las ecuaciones 5.19, 5.10, 5.20, 5.21 y 5.23 se tiene definido el filtro de Kalman de forma recursiva.

Así pues, resumiendo las ecuaciones antes deducidas:

- **Evolución de la estimación**

$$\hat{x}'_{t+1} = \Phi \hat{x}_t$$

- **Evolución de la matriz de covarianzas**

$$P_{t+1} = \Phi P_t \Phi^T + Q$$

- **Ganancia óptima de Kalman**

$$K_t = P_{t-1} H^T (H P'_t H^T + R)^{-1}$$

- **Actualización de la matriz de covarianzas**

$$P_t = (I - K_t H) P'_t$$

- **Actualización de la estimación**

$$\hat{x}_t = \hat{x}'_t + K_t (z_t - H \hat{x}'_t)$$

De esta manera, con las ecuaciones anteriores es posible construir un filtro de Kalman para cualquier sistema dinámico lineal incluso aunque exista ruido aditivo en el mismo.

5.3.1. Implementación del Filtro de Kalman en Python

Aunque es un algoritmo muy sencillo de diseñar y programar, en esta memoria se ha utilizado la implementación que ya existe en Python para este algoritmo. La librería, con un nombre más que intuitivo, llamada *Pykalman* incluye las funciones y manuales necesarios, además de algún ejemplo, para familiarizarse con este algoritmo y poderlo incluir en el desarrollo en Python.

Véase, entre otros, el siguiente ejemplo incluido en pykalman donde se exponen algunas funciones:

```
1 from pykalman import KalmanFilter
2 import numpy as np
3 kf = KalmanFilter(transition_matrices = [[1, 1], [0, 1]], observation_matrices = [[0.1,
     0.5], [-0.3, 0.0]])
4 measurements = np.asarray([[1,0], [0,0], [0,1]])  # 3 observations
```

```
5 kf = kf.em(measurements, n_iter=5)
6 (filtered_state_means, filtered_state_covariances) = kf.filter(measurements)
7 (smoothed_state_means, smoothed_state_covariances) = kf.smooth(measurements)
```

Mediante las funciones incluidas en la librería modelaremos nuestro algoritmo basado en el filtro de Kalman para poder hacer un recuento de objetos más exacto en 5.4.2.3 donde se explicará de forma detallada las funciones y parámetros utilizados.

5.4. Aplicación

Una vez entrenada la red y hechas las comprobaciones pertinentes para evaluar la precisión de los pesos obtenidos se aplicará sobre un conjunto de imágenes extraídos de un vídeo a fin de no sólo procesar la imagen para extraer objetos, sino también calcular metadados que puedan ser de interés. Todas las aplicaciones de esta memoria serán sobre vídeos con contenido de tráfico, ya sea desde una cámara fija o desde una cámara en un vehículo en movimiento. Dependiendo del tipo de vídeo utilizado puede ser interesante calcular el número de coches que atraviesan la imagen o nos adelantan si fuera una cámara móvil o bien el número de personas. Incluso se podría realizar un sistema de frenado automático que en caso de detectar un peatón delante del vehículo lo hiciera detenerse para evitar una colisión.

Durante el detalle de las modificaciones de los archivos incluidos en el repositorio se usarán fotogramas de un vídeo grabado desde una cámara en un coche. No se trata de la aplicación final, sino de una demostración de capacidades de procesado que tendrá la red para luego ser aplicada de forma cuantitativa a otras secuencias. El vídeo se puede ver en este enlace para su revisión.

El primer paso es, mediante el software gratuito mostrado en B.5 convertir el vídeo a fotogramas independientes para trabajar con ellos de forma individual. Estos fotogramas se procesarán según los apartados siguientes de la memoria hasta obtener una secuencia que pueda ser de nuevo convertida en vídeo.

Por último, para estos procesados previos y parciales, se ha bajado el número de frames para que el procesado sea más ligero. En vez de estar trabajando a 30 FPS[9] como tenía el video original, se trabajará a 3 FPS para tener un conjunto de imágenes más manejable. Al final de este apartado se volverá a convertir el vídeo al número de FPS original para ver el resultado final.

5.4.1. Preparación de la etapa de procesado

En primer lugar vamos a preparar el procesado de las imágenes a fin de que detecte y marque de forma correcta los objetos deseados. La imagen siguiente muestra una referencia de la detección sin haber realizado modificaciones del procesado del vídeo.

[9] Por sus siglas en inglés Frames Per Second

Existe, por ejemplo, la clase "backpack"que contendría aquellos objetos identificados como mochilas o bolsos. Aunque en determinados contextos pueda llegar a ser útil, en nuestro caso no lo vamos a considerar como una clase a detectar y la eliminaremos de la lista de clases válidas.

FIGURA 5.24. Imagen de referencia del vídeo procesado

No tiene demasiado sentido que la red detecte las 80 categorías presentes en COCO cuando hay algunas como “cuchillo”[10][52]. Así que lo más lógico es primero eliminar esas clases para que durante el procesado de la imagen se puedan obviar o en caso de identificación correcta no aparezcan.

La lista:

```
# COCO Class names
# Index of the class in the list is its ID.  For example, to get ID of
# the teddy bear class, use: class_names.index('teddy bear')
class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
'bus', 'train', 'truck', 'boat', 'traffic light',
'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
'kite', 'baseball bat', 'baseball glove', 'skateboard',
```

[10] Clase COCO número 49

```
11 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
12 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
13 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
14 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
15 'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
16 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
17 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
18 'teddy bear', 'hair drier', 'toothbrush']
```

resultaría en:

```
1 # COCO Class names
2 # Index of the class in the list is its ID.  For example, to get ID of
3 # the teddy bear class, use: class_names.index('teddy bear')
4 class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
5 'bus', 'train', 'truck', 'NU', 'traffic light',
6 'NU', 'stop sign', 'NU', 'NU', 'NU',
7 'NU', 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
8 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
9 'NU', 'NU', 'NU', 'NU', 'NU',
10 'NU', 'NU', 'NU', 'NU',
11 'NU', 'NU', 'NU', 'NU', 'NU',
12 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
13 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
14 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
15 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
16 'NU', 'NU', 'NU', 'NU', 'NU',
17 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
18 'NU', 'NU', 'NU']
```

Habiendo etiquetado como “NU”[11] las que no deseemos, de esta manera durante el procesado podremos identificar las no deseadas y detener ahí la identificación.

El módulo *visualize.py* incluye también una asignación de colores aleatorios para las máscaras y cajas, sin embargo en el código original se aplicaba imagen a imagen el barajado aleatorio de los colores, por lo que al realizar el vídeo quedaba un tanto molesto para su visionado por el efecto parpadeante.

```
1 def random_colors(N, bright=True):
2 """Generate random colors.
3 To get visually distinct colors, generate them in HSV space then
4 convert to RGB."""
5 brightness = 1.0 if bright else 0.7
6 hsv = [(i / N, 1, brightness) for i in range(N)]
7 colors = list(map(lambda c: colorsys.hsv_to_rgb(*c), hsv))
8 #random.shuffle(colors)
9 return colors
```

[11] No Usada

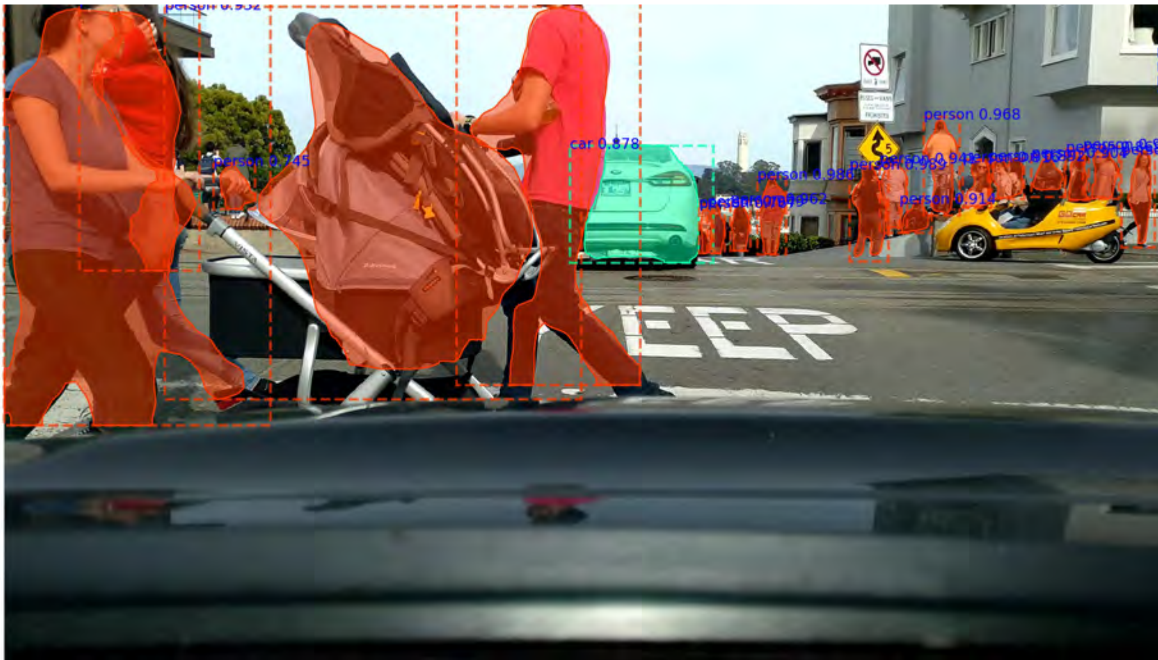


FIGURA 5.25. Imagen con clases recortadas, no aparece el objeto *backpack*

Esta función, en vez de emplearse para cada imagen se ejecutará al principio del procesado del vídeo, de manera que los colores serán uniformes durante todo el vídeo. Además se han dado colores fijos para cada clase COCO, de manera que todas imágenes tendrán un coloreado uniforme.

Este enlace muestra el vídeo reconvertido sin haber hecho la armonización de colores en las clases entre fotogramas, se puede observar que es bastante molesto trabajar con él por la gran diferencia de colores entre cada uno de los fotogramas.

Por otro lado, se fijará un límite mínimo para que un objeto sea realmente válido dentro del procesado, es decir, si detectamos un objeto <coche> con una certeza inferior al 90 % (umbral muy alto, pero válido como demostración) no se incluirá en la imagen resultante ni aparecerá su máscara.

Según se examine el procesado de la imagen se podrá comprobar si el límite impuesto es demasiado agresivo o por contra es suficiente para una buena detección de objetos. Según experiencias previas, un límite de 0.85 (85 %) de validez del objeto suele ser suficiente.

FIGURA 5.26. Imagen con objetos con nivel de aceptación bajo, se ha eliminado el coche central

FIGURA 5.27. Imagen final con etiquetas porcentuales y con límite del 85 % en la aceptación de clases

Con estas modificaciones en los fotogramas iniciales podemos construir una primera versión del vídeo procesado. El vídeo puede verse en el siguiente enlace. Acto seguido se podría utilizar este vídeo/fotogramas para realizar sobre él el post-procesado aplicación final que se desee, aunque por razones puramente didácticas se ha optado por usar vídeos publicados en la web a estos efectos como se verá en las secciones siguientes.

A modo de curiosidad se puede revisar el vídeo anterior ya procesado y con los objetos detectados con un límite del 85 %, resolución total 4K y una velocidad de fotogramas de 30 fps: Mask RCNN 4k 30 fps final version.

5.4.2. Recuento de objetos detectados

Dentro de las posibles aplicaciones de las redes neuronales en el ámbito del procesado de imagen, una posible aplicación sería detectar y contar el número de objetos de una misma clase que aparecen en los sucesivos fotogramas. No sería simplemente el conteo de todos los objetos que aparecen en cada fotograma sino, yendo hacia una mayor complejidad, tener en cuenta las apariciones anteriores para así construir una lista numerada con el número total en la secuencia total del vídeo resultado de unir los fotogramas de manera ordenada.

Para exponer el ejemplo de manera concreta, supongamos una secuencia de vídeo dividida en N fotogramas, de manera que la secuencia completa ordenada iría desde el instante t_0 hasta el instante t_N. En cada fotograma la red neuronal habrá detectado un número de objetos K_c, de manera que el vector de detecciones sería, para cada clase detectada, $D = (K_{c_0}, K_{c_1}, ..., K_{c_N})$. Una primera aproximación, muy sencilla pero con un grado muy alto de imprecisión, sería simplemente añadir al contador del objeto *clase1* el número de detecciones que hay en cada fotograma. Mejorando la técnica de recuento podemos, asumiendo que el número de objetos que aparecen en cada fotograma o abandonan la escena es pequeño, modelarlo mediante un Proceso de Poisson[53]. De forma que, el incremento entre dos fotogramas consecutivos $K_{c_M} - K_{c_{M-1}}$, siempre que sea positivo, nos indicaría el número de objetos *nuevos* para una misma clase. Así pues un contador fuera sumando cada una de las diferencias nos diría para un instante T el número de objetos detectados entre 0 y T.

Este sencillo procedimiento se podría utilizar de forma directa si la red neuronal tuviera un porcentaje de detección y acierto del 100 % que, como hemos visto en los ejemplos de detecciones anteriores no es cierto debido a que cada detección tiene un valor *score* que determina el grado de acierto para cada objeto detectado. En 5.4.1 podemos ver cómo existen diversos objetos “person” detectados con una puntuación desde el límite, en este caso 75 %, hasta el 99 %. Es más, una detección es totalmente incorrecta ya que clasificaría el carrito de bebé del primer plano como “person”. Por otro lado, como el espaciado temporal de los fotogramas no es ’0’ puesto que existe un muestreo de la imagen continua por un medio digital que es la cámara que toma el vídeo, es posible que haya varios objetos que aparecen nuevos en la escena (o que la abandonan) de forma

simultánea. De esta manera una de las condiciones de un Proceso de Poisson

$$\lim_{h \to 0} \frac{o(h)}{h} = 0$$

podría no cumplirse puesto que no vamos a tener una resolución temporal igual a 0. Por poner un ejemplo, una secuencia típica de vídeo puede funcionar a 30 fps, por lo que la resolución temporal sería de $\frac{1}{30}\ segundos = 33\ milisengundos$. En este periodo de tiempo podrían ser varios los objetos que entraran o salieran de la escena.

En resumen, las limitaciones que nos podríamos encontrar a la hora de evaluar el número de objetos encontrados en una secuencia de imágenes ordenadas temporalmente serían:

- Objetos no detectados (o hecho de forma incorrecta) en una imagen que habían sido detectados antes y lo son después.
- Objetos que entran o salen simultáneamente de la imagen
- Objetos mal clasificados

En esta memoria se abordará la solución de la primera de la limitaciones, es decir, cómo eliminar las repeticiones en el conteo de objetos debido a que por alguna razón no aparecen en la detección. Valga un ejemplo visual que ilustre este problema. Si aplicáramos el algoritmo a esta secuencia de imágenes tendríamos en 5.30(c) un nuevo objeto incluido en la cuenta dado que en la imagen anterior 5.30(b) no aparecía. Para poder procesar correctamente esta simple secuencia habría que tener en cuenta la información de 5.30(a) para estimar que ese objeto ya estaba incluido en el cómputo de elementos detectados.

La segunda limitación se podría resolver incrementando el número de *fps* de la secuencia, de manera que la probabilidad de que el número de eventos (entrada o salida de objetos entre imágenes sucesivas) sea igual o mayor que dos, tienda a 0, esto es en un Proceso de Poisson

$$P\{N(h) = 1\} = \lambda h + o(h)$$

y

$$P\{N(h) \geqslant 2\} = o(h)$$

donde se cumple

$$\lim_{h \to 0} \frac{o(h)}{h} = 0$$

Por último, la tercera limitación es independiente de la aplicación de conteo de objetos ya que viene definida por la calidad de la detección de la red neuronal, por lo que al no poderse mejorar si no es modificando la red neuronal o con otros patrones de entrenamiento, está fuera del alcance de esta memoria. La única manera de mitigar este problema es si se ha detectado el objeto erróneo durante un número muy pequeño de imágenes (por ejemplo 1), se podría asumir que ha sido debido a un espurio y eliminarlo de la cuenta final. En los casos que sea así, se realizará un postprocesado para eliminar estos *glitches* en las detecciones.

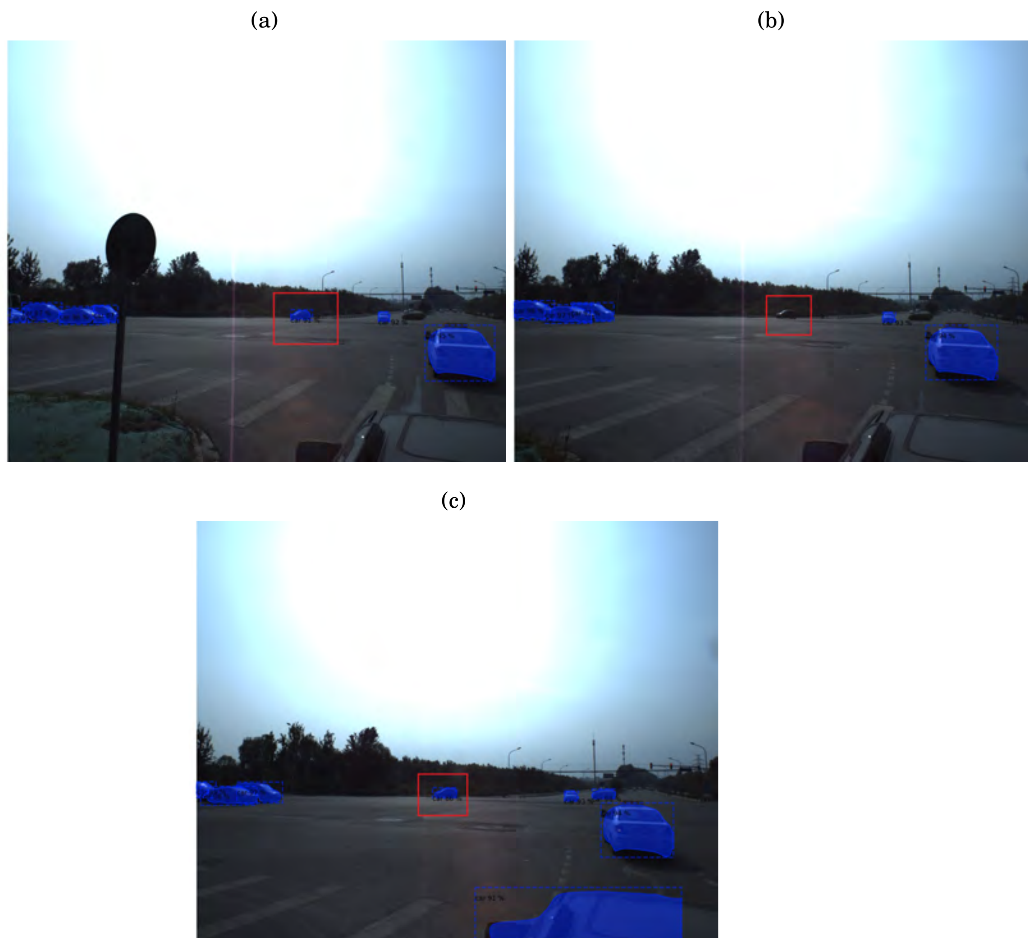


FIGURA 5.28. Se observa cómo el objeto incluido en el cuadro rojo se ha detectado en 5.30(a), pero sin embargo no se ha detectado en 5.30(b) para volver a detectarse en 5.30(c)

En la presentación de resultados de los algoritmos de recuento se usará una secuencia de imágenes obtenida de CVPR 2018 WAD Video Segmentation Challenge. La secuencia ordenada y convertida en vídeo se puede ver en CVPR 2018 test sequences. Para cada uno de los vídeos utilizados en la parte práctica de la memoria se añadirá una tabla resumen con una medida de la precisión en el recuento y detección de objetos.

5.4.2.1. Recuento de objetos: algoritmo base

La primera parte del algoritmo es, utilizando la salida del procesado de la red neuronal, un recuento de objetos asumiendo que dicha salida es ideal, esto es, no ha fallado ninguna detección y todos los objetos se comportan de manera ideal (el número de objetos que entran o salen entre dos fotogramas sucesivos es siempre 1).

Al final del código *procesado_video.py* presentado en D.5 se incluye el fragmento que calcula

la evolución de los objetos en la secuencia de vídeo.

```
images_to_process=len(file_names)
for i in range(0,images_to_process):
image = scipy.misc.imread(os.path.join(VIDEO_DIR, file_names[i]))

#print("image_id ", image_id, dataset.image_reference(image_id))

nombre_imagen = os.path.join(PROCESO_DIR, file_names[i])
print("Nombre imagen: ",nombre_imagen)

# Run detection
results = model_inf.detect([image], verbose=0)

# Visualize results
r = results[0]

num_labels_new, centroide_xn, centroide_yn,centroide_classn =postproc.extract_info(r['
    class_ids'],r['scores'],r['masks'],r['rois'],detect_th,class_names,num_labels)

num_labels_diff=np.subtract(num_labels_new,num_labels)

for j in range(0,len(class_names)):
num_labels_accum[j]=num_labels_accum[j]+num_labels_diff[j]

num_labels=num_labels_new

visualize.display_instances_save(image, r['rois'], r['masks'], r['class_ids'],
    class_names, num_labels_accum, r['scores'],nombre_imagen, colors,detect_th)
```

De esta manera, para cada imagen mediante *postproc.extract_info* de D.6 se resuelve cuántos objetos se han detectado nuevos con respecto al fotograma anterior sin tener en cuenta ningún tipo de análisis estocástico. Dado que la función se ha hecho genérica para ampliaciones posteriores, también resuelve el número de centroides para cada uno de los objetos detectados, en sucesivos apartados veremos la utilización de esa información.

Tras procesar las 50 primeras imágenes del vídeo completo con un límite para la aceptación de cada detección igual a 0.85 (hecho de esta manera para reducir el tiempo de procesado en estas pruebas previas) se obtienen los siguientes resultados finales en el recuento en la Figura 5.29

Recordando las imágenes de la Figura 5.28 en las que se observaba cómo existía un objeto que estaba en los fotogramas inicial y final, pero no en el intermedio, si comprobamos ahora la misma secuencia debería haberse incrementado el contador en una unidad de manera errónea a la realidad ya que ese objeto ya había sido incluido.

Este problema se solucionaría con las modificaciones del algoritmo base contempladas en las próximas secciones de la memoria.

FIGURA 5.29. Recuento de objetos: algoritmo inicial sin mejoras, 50 imágenes y un límite de detección del 85 %

5.4.2.2. Recuento de objetos: aproximación por proximidad de centroides

La primera aproximación para solventar el conteo en exceso de objetos que en un fotograma no son reconocidos para en el siguiente sí entrar en la lista de objetos detectados consiste en calcular si el nuevo objeto tiene una posición próxima a uno ya detectado para, de esta manera, discriminar si es un nuevo objeto o simplemente uno desplazado.

Dado que la salida de la función de detección de Mask RCNN incluye las posiciones de las cajas que engloban cada objeto detectado, es muy sencillo calcular los centroides de cada objeto a partir de estos datos.

```
1 # Run detection
2 results = model_inf.detect([image], verbose=0)
3 num_labels_new, centroide_xn, centroide_yn,centroide_classn =postproc.extract_info(r['
      class_ids'],r['scores'],r['masks'],r['rois'],detect_th,class_names,num_labels)
4
5 #Dentro de la ejecución de extract_info:
6 centroide_y.append((rois[i,0]+rois[i,2])/2)
7 centroide_x.append((rois[i,1]+rois[i,3])/2)
```

En D.6 se recoge la función completa que permite calcular los centroides de todos los objetos que tengan un *score* mayor que un límite y además sean de una clase reconocida. Recordemos que las

FIGURA 5.30. Se observa cómo el objeto incluido en el cuadro azul se ha detectado en 5.30(a), pero sin embargo no se ha detectado en 5.30(b) para volver a detectarse en 5.30(c). Se ha contado por duplicado al detectar un nuevo objeto en la imagen.

clases no válidas se han llamado *'NU'* a fin de eliminar ruido del procesado.

Una vez generada la lista de centroides con *append* se ordena de forma creciente en función de la coordenada X de cada objeto. Esa ordenación guarda los índices para reordenar a su vez la coordenada Y y el vector de clases, siempre en función de los índices generados al ordenar X.

```
centroide_x_orden=np.sort(centroide_x)
for i in range(0,len(centroide_y)):
  centroide_y_orden.append(centroide_y[indices_x[i]])
```

```
4   centroide_class.append(class_filtrada[indices_x[i]])
```

De esta manera, para cada vector de detecciones tendremos asimismo las coordenadas de sus centroides y las clases junto con las coordenadas de los centroides de *frames* pasados y sus clases. Procesando toda esta información en D.7 obtendremos un vector ordenado por clases que incluye el factor a corregir para cada nueva detección. Este factor de corrección se calcula de la siguiente manera: Sea el vector de detecciones, para cada clase detectada $D = (K_{c_0}, K_{c_1}, ..., K_{c_N})$. Fijada una clase C_i se tiene los vectores de centroides tanto en eje X como Y del fotograma actual N y del dos veces anterior $N-1$. Para cada elemento del vector actual de centroides, se realiza la resta con todos los anteriores y se evalúa el error cuadrático medio de las restas.

$$\frac{1}{2}\sqrt{(C_N - C_{N-1})^2_{X_i} + (C_N - C_{N-1})^2_{Y_i}}$$

Si este error es menor que un umbral dado, se considera que el nuevo objeto está próximo a otro ya detectado y por tanto no se debe contabilizar. De esta manera se actualiza una variable que corregirá el número de incrementos de objetos. Además, para mejorar el cálculo de la corrección, se comprueba que cada objeto de la imagen actual o de la dos veces anterior únicamente corrigen una vez el vector de ajuste. De esta manera se evita, por ejemplo, que un objeto esté próximo a dos anteriores y se pueda confundir con ser evoluciones de los dos a la vez. Este proceso, representado en D.7 tendría el siguiente resumen:

```
1     for i in range(0,len(centroid_x)):
2     cx_diff=[]
3     cy_diff=[]
4     sqrtdiff=[]
5     flag=0
6     for j in range(0,len(centroid_xold)):
7       sqrdiff=threshold+100
8       if (centroid_class[i]==index) & (centroid_classold[j]==index):
9         sqrdiff=math.pow(centroid_xold[j]-centroid_x[i],2)+math.pow(centroid_yold[j]-
      centroid_y[i],2)
10        sqrdiff=math.sqrt(sqrdiff)/2
11      else:
12        sqrdiff=100000
13      sqrtdiff.append(sqrdiff)
14      if (sqrdiff<threshold) & (flag==0) & (flag_dest[j]!=1):
15        bias=bias-1
16        flag=1
17        flag_dest[j]=1
```

Finalmente se corregirá el contador para cada clase a fin de mejorar la aproximación del recuento.

Como en el apartado anterior, se ha implementado este algoritmo sobre 50 imágenes del vídeo de ejemplo con el resultado de la Figura 5.31. Se puede comprobar cómo el número de objetos de clase *car* ha descendido notablemente de 36 con el algoritmo base a 25 con este algoritmo modificado. El umbral inferior bajo el cual se asume que un objeto está en la órbita de uno ya

contabilizado es 20, esto es, si se cumple

$$\frac{1}{2}\sqrt{(C_N - C_{N-1})^2_{X_i} + (C_N - C_{N-1})^2_{Y_i}} > 20$$

contabilizaremos un nuevo objeto.

Reduciendo este límite podremos modular la salida del algoritmo para así encontrar el valor numérico que mejor se aproxime a la salida ideal del recuento de objetos. Este paso se hará en el último apartado junto con la comparación cualitativamente de los diversos algoritmos para todas las secuencias de test.

FIGURA 5.31. Recuento de objetos: algoritmo mejorado con detección de posición, 50 imágenes y un límite de detección del 85 %

Por último se muestra a modo de ejemplo la secuencia de tres fotogramas consecutivos de manera similar a la Figura 5.30 donde con esta modificación no debería incrementarse el contador con el objeto detectado en la Figura 5.30(c).

En la Figura 5.32 se puede comprobar cómo en la última imagen del conjunto 5.30(c) no se ha incrementado el contador. El objeto resaltado ha sido identificado como ya contabilizado y no ha dado lugar a ninguna actualización del contador de clases.

Si la densidad de objetos en una sección de la imagen es muy alta, va a ser muy sencillo confundir los centroides ya que todos parecerán estar próximos unos a otros, engañando de esta forma al algoritmo. Este problema está fuera ya del alcance de esta memoria por la complejidad que entraña.

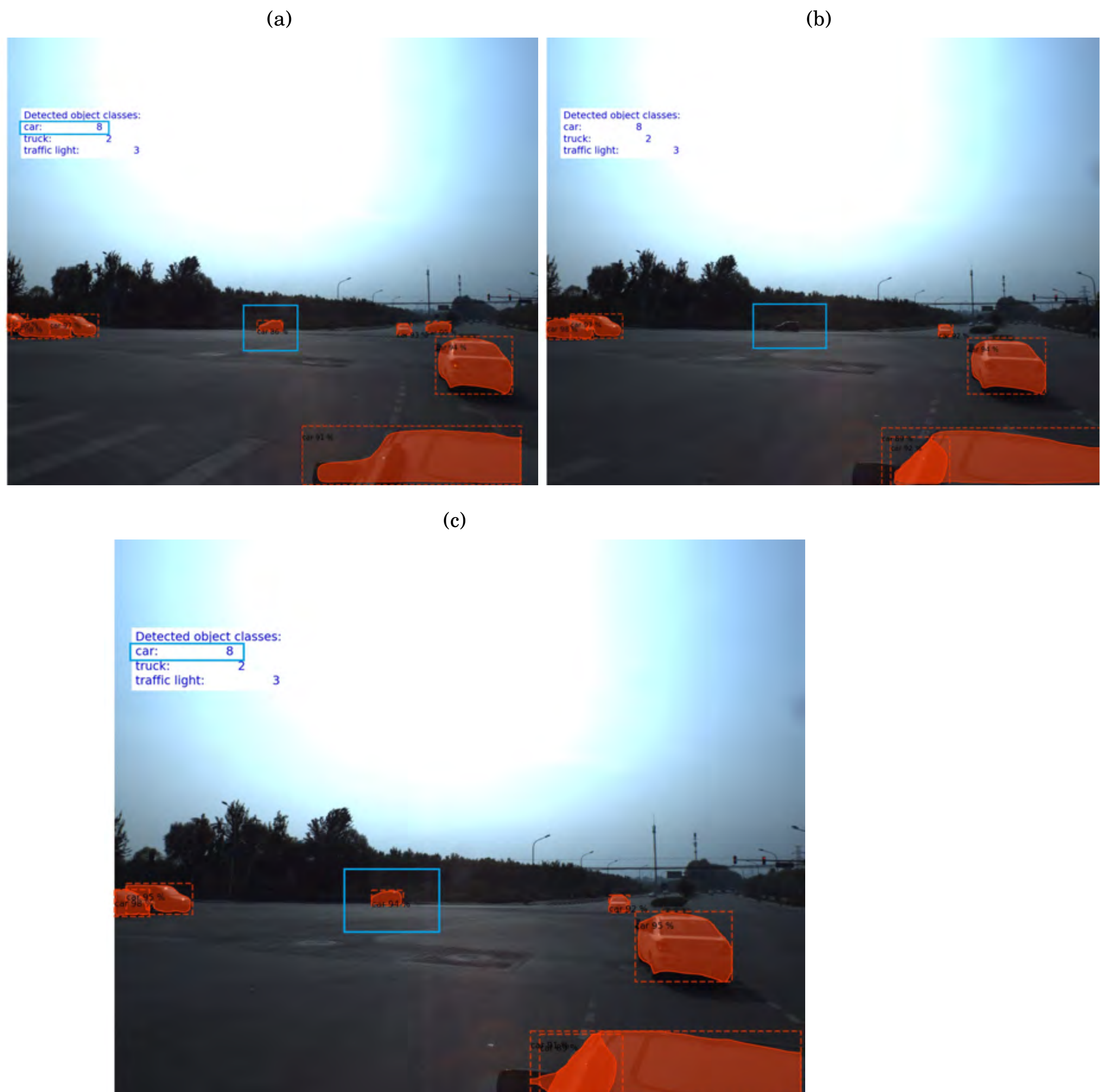


FIGURA 5.32. Se observa cómo el objeto incluido en el cuadro azul se ha detectado en 5.32(a), pero sin embargo no se ha detectado en 5.32(b) para volver a detectarse en 5.32(c).En este caso la mejora del algoritmo de conteo ha detectado el objeto como ya incluido en la cuenta anteriormente y no ha incrementado el contador

El principal problema de este sistema para calcular objetos próximos y así eliminarlos del resultado final es que no tiene en cuenta la velocidad ni la trayectoria. Simplemente comprueba si los objetos han tenido anteriormente alguno en su órbita y en función de un límite infiere si es nuevo o no. Es decir, este algoritmo sólo sería válido en los siguientes supuestos:

- Los objetos están detenidos.
- Los objetos tienen una velocidad muy baja.
- El tiempo entre fotogramas es tan pequeño que permite suponer que los objetos se desplazan una pequeña cantidad entre fotogramas. Es decir, se podría aplicar los supuestos anteriores.

Para poder utilizar de forma general un algoritmo de mejora es necesario pues utilizar algún estimador que tenga en cuenta no sólo las posiciones de manera absoluta, sino también su evolución temporal. Para ello serán necesarias las técnicas del siguiente capítulo.

5.4.2.3. Recuento de objetos: aproximación mediante filtrado estocástico

Esta mejora del algoritmo base se basa en el filtro de Kalman introducido en 5.3 y la implementación que tiene en Python mediante PyKalman. Este algoritmo se utilizará en la aplicación de la memoria para calcular dónde, según la trayectoria que lleve el objeto, debería encontrarse en el fotograma actual tras no haberse detectado en el fotograma anterior.

De esta manera, tenemos la siguiente información extraída de la función detallada en D.6:

- Fotograma N-3: tenemos la posición del objeto guardada en $x3$ e $y3$
- Fotograma N-2: tenemos la posición del objeto guardad en $x2$ e $y2$
- Fotograma N-1: no tenemos información del objeto debido a que no se detectó. Para añadirla a la función de cálculo se hará una estimación en base a las posiciones anteriormente conocidas de la siguiente manera $x1 = x_{est} = 2 * x2 - x3$ e $y1 = y_{est} = 2 * y2 - y3$
- Fotograma N: posición actual del objeto a seguir mediante las variables xn e yn

Así pues con la información de los fotogramas N-1, N-2 y N-3 se construye la función de Kalman:

```
1 kf =KalmanFilter(transition_matrices = [[1, 0], [0, 1]], observation_matrices = [[1, 0],
      [0, 1]], initial_state_mean=[centroid_xold2[k], centroid_yold2[k]])
2 missing_pos=[(2*centroid_xold[j]-centroid_xold2[k]),(2*centroid_yold[j]-centroid_yold2[k
      ])]
3 measurements=np.asarray([[centroid_xold2[k], centroid_yold2[k]],[centroid_xold[j],
      centroid_yold[j]],missing_pos])
4 (filtered_state_means, filtered_state_covariances) = kf.filter(measurements)
```

De acuerdo a las ecuaciones presentadas en el capítulo 5.3 para la definición y trabajo con el Filtro de Kalman y adaptadas a nuestros supuestos de cálculo, tendríamos las siguientes matrices y puntos iniciales:

Matriz de transición $\Phi = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Matriz de covarianzas $P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Estado inicial determinado por la posición del centroide del objeto en el fotograma N-3, o como se ha llamado aquí, *centroid_xold2[k] centroid_yold2[k]*, ya que es el tenemos hace dos fotogramas.

Con estos datos construimos el Filtro de Kalman mediante la función de *pykalman KalmanFilter* que se almacena en la variable kf.

```
1 kf =KalmanFilter(transition_matrices = [[1, 0], [0, 1]], observation_matrices = [[1, 0],
      [0, 1]], initial_state_mean=[centroid_xold2[k], centroid_yold2[k]])
```

Para hacer evolucionar el sistema, que se correspondería con la evolución de la estimación calculada en 5.3 empleamos *np.asarray* y los vectores que van a recoger la información de los fotogramas N-3 y N-2, así como la predicción de la posición del objeto no detectado, almacenada en *missing_pos*.

```
1 missing_pos=[(2*centroid_xold[j]-centroid_xold2[k]),(2*centroid_yold[j]-centroid_yold2[k
      ])]
```

Por último, mediante la función *kf.filter* creamos el filtro de Kalman para todas las medidas anteriores y como resultado extraemos la evolución de tanto el estado del sistema como las covarianzas.

```
1 (filtered_state_means, filtered_state_covariances) = kf.filter(measurements)
```

Calculadas con las siguientes ecuaciones:

$$P_t = (I - K_t H)P'_t$$

$$\hat{x}_t = \hat{x}'_t + K_t(z_t - H\hat{x}'_t)$$

Una vez construida la función de aproximación, es muy sencillo añadir nuevos valores a la misma debido a la propiedad recursiva que posee el Filtro de Kalman. De esta manera:

```
1 (next_state_means, next_state_convariances) = kf.filter_update(filtered_state_means[-1],
      filtered_state_covariances[-1],[centroid_x[i],centroid_y[i]])
```

Y calculamos la diferencia entre los centroides reales del fotograma N y la aproximación de Kalman. Si esta diferencia en valor cuadrático medio es inferior a un límite, consideraremos que el objeto no es nuevo en la escena. Si es superior, incrementaremos el contador en una unidad:

```
1 sqr_diff=math.pow(centroid_x[i]-next_state_means[0],2)+math.pow(centroid_y[i]-
      next_state_means[1],2)
2 sqr_diff=math.sqrt(sqr_diff)/2
3 if (sqr_diff<threshold) & (flag==0) & (flag_dest[j]!=1) & (flag_dest2[k]!=1):
4 bias=bias-1
5 flag=1
6 flag_dest[j]=1
```

```
flag_dest2[k]=1
break
```

Como en 5.4.2.2, cada objeto que actualice el contador *bias* sólo puede hacerlo una vez al igual con los objetos que contribuyan a la trayectoria calculada. Es decir, cada trayectoria de igual manera sólo puede contribuir una vez a actualizar el contador *bias*.

Aunque a priori este método parezca similar al descrito en 5.4.2.2 no es así. En 5.4.2.2 trabajábamos con posiciones absolutas, es decir, que el nuevo objeto esté en la órbita de otro detectado sin tener en cuenta posibles desplazamiento del objeto (o de la cámara si está colocada en un elemento móvil) mientras que este método utiliza métodos para estimar la posición en base a posiciones anteriores y velocidades. Además, mientras el primer método no puede usar información nada más que de un único fotograma anterior (N-2), en este método con el Filtro de Kalman podemos añadir cuantos fotogramas necesitemos para estimar de manera más precisa la trayectoria. En esta memoria se ha usado la información de los fotogramas N-2 y N-3, pero perfectamente se podrían haber empleado más mejorando la trayectoria y por tanto la precisión en la discriminación si es un objeto nuevo en la escena o ya había sido contado anteriormente. Hay que tener en cuenta que cuantas más trayectorias se añadan, mayor será la pérdida de rendimiento del sistema por un número mayor de cálculos.

Con esta modificación de algoritmo y nuevamente aplicada sobre la secuencia de ejemplo, tenemos los siguientes resultados usando un límite de aceptación de 20.

Los resultados con esta variación del algoritmo son significativamente diferentes (todavía no estamos en posición de afirmar si mejores o peores) por lo que en efecto los criterios de cálculo han variado. En el punto 5.4.2.4 se estudiarán los tres algoritmos (base, detección por posición y detección por trayectoria) para tener una valoración cuantitativa del grado de precisión de los mismos.

Con esta modificación está resuelta la problemática que forzaba a que los objetos a detectar debían cumplir unas condiciones de velocidad baja o forzar a que la cadencia de fotogramas de la cámara fuera lo suficientemente alta como para que la diferencia de posición entre dos imágenes fuera muy pequeña. La limitación asociada a una gran densidad de objetos no ha sido resuelta. Como se puede suponer, cuando el número de objetos es muy grande calcular trayectorias puede ser realmente complicado debido a la gran confusión que puede llegar a tenerse por el incremento de variables de cálculo.

Supongamos que queremos comprobar si uno de los M objetos pertenece a uno de los K objetos ya detectados. En este caso necesitaríamos hacer $M \cdot K$ cálculos para hacer todas las comparaciones. Pero si en el caso de trayectorias tenemos K_1 objetos en el fotograma 1 y K_2 en el fotograma 2, necesitaríamos hacer $M \cdot K_1 \cdot K_2$ cálculos. Podemos suponer K_1 y K_2 iguales por lo que la complejidad final sería $M \cdot K_1^2$, valor que crece cuadráticamente frente a la detección de posición.

Si aumentamos el número de fotogramas en los cálculos la complejidad escalará de forma

FIGURA 5.33. Recuento de objetos: algoritmo mejorado con Filtro de Kalman, 50 imágenes y un límite de detección del 85 %

progresiva, por lo que habría que limitar el número de trayectorias a aquellas que se mantengan, es decir, que los nuevos puntos de la trayectoria en verdad se mantengan dentro de la estimación para la inclusión. Si el número de objetos detectados en cada imagen fuera pequeño, las trayectorias tendrían mucha menos complejidad en su cálculo. Por ejemplo, para el objeto usado de muestra en la Figura 5.34 es muy sencillo calcular su trayectoria ya que hay pocos objetos detectados en la imagen, por lo cual vamos a poder estimar de forma muy precisa dónde se va a encontrar en cada punto y si son nuevas apariciones o no.

Lo que sí vamos a comprobar en este apartado es si frente a la situación descrita en la Figura 5.28 el algoritmo es capaz de eliminar del cómputo de objetos detectados aquellos que ya estaban en la lista.

Incluimos en la Figura 5.34 los tres fotogramas de ejemplo y comprobarmos si el vehículo destacado aparece y desaparece y cómo el contador de objetos no debería variar. Se comprueba que en efecto el algoritmo ha sido capaz de detectar que el objeto detectado sigue una trayectoria calculada y por tanto no se trata de un nuevo objeto, sino de un desplazamiento de uno ya detectado anteriormente.

FIGURA 5.34. Se observa cómo el objeto incluido en el cuadro azul se ha detectado en 5.34(a), pero sin embargo no se ha detectado en 5.34(b) para volver a detectarse en 5.34(c).En este caso la mejora del algoritmo de conteo ha detectado el objeto como ya incluido en la cuenta anteriormente y no ha incrementado el contador

5.4.2.4. Comparativa entre los tres algoritmos de conteo de objetos

En esta sección se analizarán los tres algoritmos estudiados en los apartados anteriores. Además, para el caso de estudio se hará un recuento visual de los objetos para detectar el grado de precisión y se indicarán qué errores son fruto de un mal conteo o de una mala detección por parte de la red neuronal. Además, permitirá ajustar los límite de inclusión tanto para el algoritmo que

discierne por posición absoluta como el que analiza las trayectorias.

En el código citado en D.5 se han incluido unas variables que permiten de una forma sencilla tener en cuenta los cálculos por proximidad y por trayectoria en el recuento final.

```
1 use_prox=0
2 use_traj=0
3 for j in range(0,len(class_names)):
4 add[j]=0
5 if (num_labels_diff[j]>0):
6 add[j]=isnew(j,num_labels_diff[j],centroide_xn,centroide_yn,centroide_x2,centroide_y2,
      centroide_classn,centroide_class2)
7 trayec[j]=kalman_tr(j,num_labels_diff[j],centroide_xn,centroide_yn,centroide_x2,
      centroide_y2,centroide_y3,centroide_x3,centroide_classn,centroide_class2,
      centroide_class3)
8 if (num_labels_diff[j]+use_prox*add[j]+use_traj*trayec[j])>0:
9 num_labels_accum[j]=num_labels_accum[j]+num_labels_diff[j]+use_prox*add[j]+use_traj*
      trayec[j]
10 else:
11 num_labels_accum[j]=num_labels_accum[j]
```

Mediante las variables *use_prox* y *use_traj* se puede restar la contribución errónea en la detección al contador de objetos por clase según se asignen a 0 ó 1. Esto permite modificar rápidamente el código para hacer pruebas sobre una u otra configuración de algoritmo.

En la tabla 5.1 se presentan los resultados de los tres algoritmos frente a los datos reales que se habrían obtenido con un examen visual de las imágenes. Para este ejemplo se recuerda que se han tomado los 50 primeros fotogramas de un vídeo extraído del concurso CVPR 2018 WAD. Estos fotogramas se pueden revisar en Vídeo de ejemplo.

La explicación de las diferentes columnas sería:

- **Clase objetos.** Muestra la etiqueta bajo la cuál se clasifican todos los objetos que han aparecido en la secuencia. En particular, para esta lista de fotogramas son: car, truck[12], traffic light, stop sign, person.

- **Recuento visual.** Es el número de elementos de cada clase reales que existen en la secuencia, revisados visualmente con independencia de si han sido detectados o no por la red neuronal. Por ejemplo, no existe ningún elemento de la clase *person* en realidad, se ha detectado por un error de la red neuronal. La detección se hizo con una precisión del 95% por lo cual es complicado eliminarlo de la lista dado su alto valor. La eliminación de este espurio se hará en el siguiente capítulo, usando otras técnicas de filtrado de objetos cuando aparecen de forma esporádica en las secuencias.

- **Recuento visual en función de la detección.** En este caso es el recuento de elementos una vez detectados y con su máscara correspondiente, es decir, las salidas producidas por

[12] Realmente es un autocar

la red neuronal en forma de máscara y etiqueta de objeto. Éste sería el valor al cuál nos hemos de acercar dado que estamos trabajando sobre un post-procesado de las salidas de la red neuronal.

- **Recuento con algoritmo base.** Se trata del recuento con el algoritmo presentado en 5.4.2.1. Es de esperar que, dado que no se hace ninguna corrección, este número sea mayor que el obtenido en el recuento visual.

- **Recuento con algoritmo mejorado por discriminación de posición.** Se trata del recuento con el algoritmo presentado en 5.4.2.2. Un valor normal debería ser superior al del recuento visual.

- **Recuento con algoritmo mejorado por discrimación de trayectoria.** Se trata del recuento con el algoritmo presentado en 5.4.2.3. Un valor normal debería ser superior al del recuento visual.

Clase objetos	Recuento visual	Recuento visual en función de la detección Mask RCNN	Recuento algoritmo base	Recuento algoritmo mejorado por posición	Recuento algoritmo mejorado por trayectoria
		MaskRCNN score=0.85	MaskRCNN score=0.85	MaskRCNN score=0.85 Lím. proximidad posición = 5	MaskRCNN score=0.85 Lím. proximidad trayectoria = 5 Núm. máx. trayectorias = 50
Car	33	23	36	36	34
Truck	1	1	2	2	2
Traffic light	5	6	7	7	7
Stop sign	1	2	2	2	2
Person	0	1	1	1	1
Porcentajes	NA(40)	NA(33)	83 %(48)	83 %(48)	87 %(46)

TABLA 5.1. Comparativa de algoritmos de recuento de objetos para 50 fotogramas

Pasamos ahora a analizar los resultados de la tabla 5.1 de una forma cualitativa:

- **Clase <car>:** en esta clase, la más numerosa de la secuencia, se engloba el recuento de todos los coches detectados durante todos los fotogramas. Partiendo de los 23 coches

detectados y contados visualmente a partir de la salida de la red neuronal, vemos cómo el que mejor aproxima el número real es el algoritmo modificado mediante el Filtro de Kalman. El algoritmo base ha contado 36 coches, lo cual es un acierto del 64 % mientras que el algoritmo con el Filtro de Kalman ha contado 34, siendo un acierto del 68 %. Estaba claro que esta mejora del algoritmo no era óptima puesto que no tenía en cuenta la velocidad de desplazamiento de los objetos.

- **Clase <truck>:** aquí se engloban los camiones detectados en la secuencia. Realmente sólo existe un vehículo de esta clase y que además es un autocar. Los recuentos visuales muestran un único objeto sin embargo los tres algoritmos han contado dos. Esto se debe a que el número de fotogramas entre dos apariciones del objeto es muy grande (concretamente dos) y además el objeto aparece al principio estacionado durante un único fotograma, por lo que es complicado hacer una buena identificación. Esta clase tiene un error del 50 %, pero dado que la muestra es muy pequeña, es un valor muy poco representativo.

- **Clase <Traffic light>:** aquí se engloban los semáforos. Existe un error entre los semáforos reales que existen en la secuencia y los detectados, la red neuronal se confunde en una ocasión, por lo que el número a comparar serían 6 semáforos y no 5. Vemos como en todos los algoritmos los resultados son idénticos.

- **Clase <Stop sign>:** aquí se engloban las señales de stop. Únicamente existe una señal de STOP en la secuencia si bien es cierto que no se distingue claramente pero se adivina por la forma. El resto de señales detectadas son señales de tráfico pero no STOP propiamente dicho. Todos los algoritmos tienen un acierto del 100 % con esta clase

- **Clase <Person>:** aunque los tres algoritmos aciertan con el número de personas en la secuencia, el error viene producido por la red neuronal. Ha clasificado un objeto erróneamente como persona (en concreto es el tronco de un árbol).

- **Porcentajes precisión del recuento:** esta fila, meramente informativa, suma todos los objetos como si fueran de la misma clase y calcula el porcentaje de acierto. Está claro que mezclar todas las clases no tiene un valor representativo alto pero vale como estimación muy superficial de la bondad de cada uno de los algoritmos.

En resumen, el mejor algoritmo de recuento de objetos en fotogramas es el que utiliza un Filtro de Kalman para calcular la posición estimada de objetos en movimiento y así poder discernir si los nuevos objetos son un desplazamiento de otros ya detectados en efecto han entrado en la imagen.

Aunque se podrían haber variado los límites para la aceptación de nuevos objetos como evolución de los antiguos e intentar optimizarlos entre los dos algoritmos de mejora, se ha preferido dejar el mismo valor en ambos para así tener una comparación más justa. Aumentando

el límite de aceptación para el algoritmo de posición se generaría un cuadro más ancho para englobar objetos nuevos, sin embargo esto no es real porque lo que estamos es creando una zona *muerta* en la que los objetos que ahí se encontraran quedaría fuera de cómputo. Con un límite estrecho es poco probable que haya objetos no contados por error, sin embargo es más complicado estimar por posición si el objeto es nuevo o no. Esta razón también apoya que el algoritmo de mejora por trayectoria es un método más eficiente para la estimación de objetos nuevos en las secuencias.

FIGURA 5.35. Resultados finales para los tres algoritmos: base en 5.35(a), mediante ajuste por posición en 5.35(b) y mediante ajuste por trayectoria en 5.35(c)

Por último la tabla 5.2 presenta, para el algoritmo mejorado con Filtro de Kalman, los resultados en función del límite de proximidad. A la vista de esos resultados, se usará para el procesado final un límite de proximidad de 50 que permite un mejor ajuste de los objetos detectados sin perder la capacidad de no añadir aquellos que ya habían aparecido pero debido a un error de la red neuronal no estaban en algún fotograma como ocurre en la Figura 5.28

Clase objetos	Recuento visual en función de la detección Mask RCNN	Recuento algoritmo mejorado por trayectoria			
	MaskRCNN score=0.85	MaskRCNN score=0.85 Lím. proximidad trayectoria = 1	MaskRCNN score=0.85 Lím. proximidad trayectoria = 5	MaskRCNN score=0.85 Lím. proximidad trayectoria = 15;30;40	MaskRCNN score=0.85 Lím. proximidad trayectoria = 50 y mayores
Car	23	36	34	28	27
Truck	1	2	2	2	2
Traffic light	6	7	7	7	7
Stop sign	2	2	2	2	2
Person	1	1	1	1	1
Porcentajes	NA(33)	69 %(48)	72 %(46)	83 %(40)	85 %(39)

TABLA 5.2. Comparativa de algoritmos con Filtro de Kalman para 50 fotogramas y diferentes límites de proximidad de trayectorias

5.4.2.5. Eliminación de objetos espurios

Existe otro efecto en el procesado de las imágenes que afecta al resultado final dado que se añaden objetos que en verdad no han aparecido en la propia imagen. Esto se debe a un error en la detección de la red neuronal, es decir, la red neuronal ha interpretado de forma equivocada que una determinada forma se puede clasificar dentro de las categorías COCO en nuestro caso.

Véase por ejemplo en la Tabla 5.1 la clase <Person> con un único objeto detectado y clasificado en ella. Mediante el recuento visual de las secuencias se descubre que no hay ninguna persona realmente en la misma, es decir, ha sido un error de la red neuronal. De hecho en la Figura 5.36 se comprueba que ese objeto es en verdad un árbol. Dado que los fotogramas anteriores y posteriores no muestran el objeto, sería una buena manera de corregir el error hacer un histograma de los recuentos y a partir de un filtrado de las entradas eliminar esos espurios. El histograma, para que no elimine entradas reales, se hará sobre una ventana móvil de 5 fotogramas, de manera que cuando la imagen central de la secuencia muestre una entrada de objeto y el resto no, eliminará esa entrada al entenderla como un espurio.

Hay que, por otro lado, tener cuidado de no eliminar entradas que ya hayan sido filtradas por las diferentes variaciones del algoritmo de recuento como puedan ser la mejora por posición o por trayectoria mediante Filtro de Kalman. En nuestro caso se hará un filtrado de aquellos objetos que en el histograma móvil sólo aparezcan una única vez, por lo que no entraría en conflicto con el resto de variaciones del algoritmo.

FIGURA 5.36. Se observa cómo el objeto <Person>detectado en 5.36(a) en realidad es un tronco de árbol en 5.36(b)

A fin de calcular la mejora que supone añadir esta función que elimina los objetos espurios en la detección se calculará el ratio entre errores (falsos positivos) y cuántos se eliminarían.

Supongamos que p es la probabilidad de tener un falso positivo en una detección, esto es, que aparezca un objeto nuevo sin que de verdad exista. En la figura 5.36(a) sería el objeto *person*. Si seleccionamos una ventana de fotogramas, la probabilidad de no ser eliminado, es decir, que contemos más de una detección de este objeto de forma errónea sería

$$P_d = 1 - 5p(1-p)^4 = \sum_{i=2}^{5} \binom{5}{i} p^i (1-p)^{5-i} + (1-p)^5$$

O lo que es lo mismo, eliminaremos espurios con una probabilidad

$$P_e = 5p(1-p)^4$$

Teniendo en cuenta que usamos una ventana deslizante de 5 fotogramas. Esta función tiene un máximo en el intervalo $0 \leqslant p \leqslant 1$ que indicaría la probabilidad de eliminar un objetos espurio de forma absoluta, esto es, sin importar la probabilidad de ocurrencia del mismo. El máximo se corresponde con el valor $p = 0{,}2$.

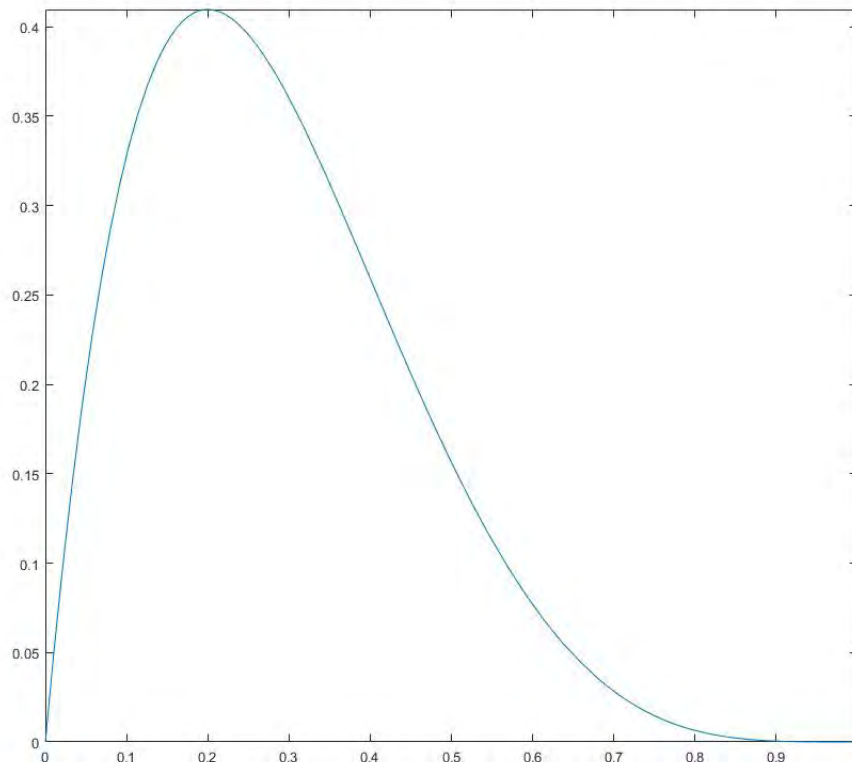


FIGURA 5.37. Probabilidad eliminar detección de objeto espurio

La expresión anterior podría dar lugar a pensar que esta mejora tiene mayor eficiencia cuando $p = 0{,}2$ pero hay que tener en cuenta también la propia probabilidad de tener un falso positivo. De esta manera define un ratio entre falsos positivos y eliminados igual a

$$r = \frac{5p(1-p)^4}{p} = 5(1-p)^4$$

que a su vez tiene una gráfica igual a

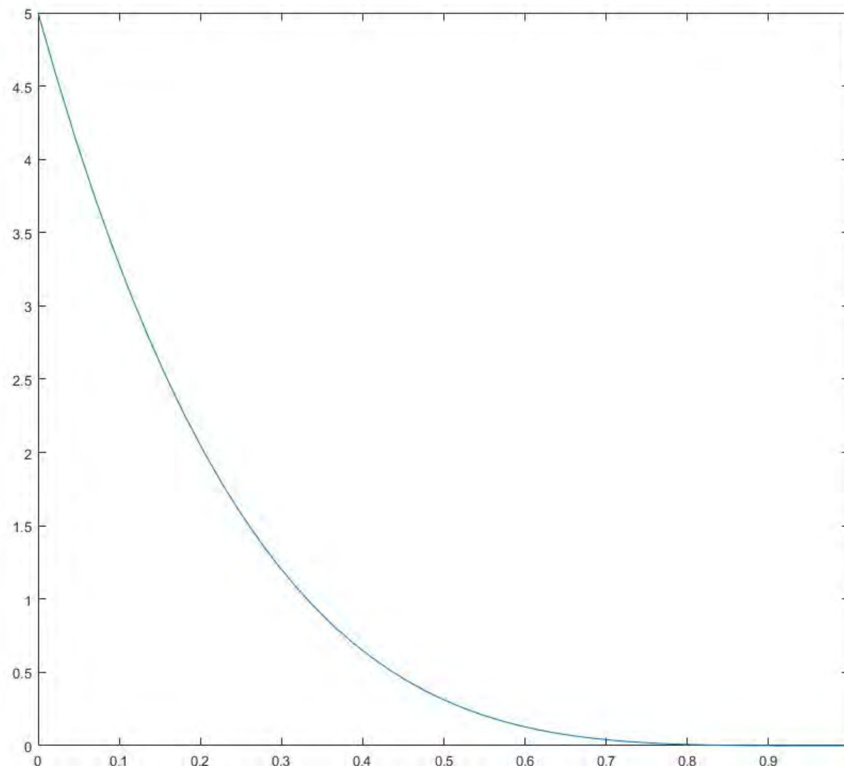


FIGURA 5.38. Eficacia eliminar detección de objeto espurio

Se observa que en este caso el máximo se encuentra en $p = 0$ lo cual es lógico, si no pueden existir falsos positivos, el resultado final tras aplicar el filtro va a ser, nuevamente, sin falsos positivos. Por otro lado el mínimo se encuentra en $p = 1$, si la probabilidad de tener un falso positivo es máxima, el resultado final tras aplicar el filtro seguirá mostrando el falso positivo, ya que va a aparecer en todos y cada uno de los fotogramas, por lo que el filtrado no tendrá ningún efecto.

Tras someter la lista de objetos detectados a la comprobación adicional en el código incluido en D.9 se obtienen los resultados finales en los que la clase <Person> detectada de forma errónea ha sido eliminada. Por tanto la tabla final de detecciones sería la presentada en Tabla 5.3.

FIGURA 5.39. Recuento de objetos: algoritmo final

Clase objetos	Recuento visual	Recuento visual en función de la detección Mask RCNN	Recuento algoritmo final Lím. proximidad trayectoria = 50 Núm. máx. trayectorias = 50
Car	33	23	27
Truck	1	1	2
Traffic light	5	6	7
Stop sign	1	2	2
Person	0	1	0

TABLA 5.3. Comparativa final de clases detectadas

Con este algoritmo, y los parámetros de configuración en lo que a límite de proximidad y número máximo de trayectorias para el que se aplica, se efectuará el análisis de las salidas de la red neuronal para realizar el recuento de objetos en imágenes extraídas de CVPR 2018 en la próxima sección.

5.4.2.6. Aplicación del algoritmo de recuento en secuencias completas

Se ha aplicado el algoritmo y la salida de la red neuronal sobre 11 secuencias de diferente tamaño extraídas de CVPR 2018. Todas estas secuencias se subirán a la red para su comprobación y revisión y en la memoria se incluirán únicamente los resultados finales para cada una de ellas de forma breve. El conjunto de secuencias se puede comprobar en CVPR 2018 test sequences donde se han unido todas las secuencias parciales en una única.

Se presentan a continuación los últimos fotogramas para cada una de las secuencias así como un enlace a la completa con la salida directa de Mask-RCNN y las resultantes de realizar el algoritmo sobre ellas. Esta última presenta varios cortes o saltos en el cuadro resumen debido a que no se ha procesado de una única vez, sino que es la unión de cada una por separado.

- Vídeo 00

FIGURA 5.40. Vídeo 00, valores finales

- Vídeo 01

FIGURA 5.41. Vídeo 01, valores finales

- Vídeo 02

FIGURA 5.42. Vídeo 02, valores finales

- Vídeo 03

FIGURA 5.43. Vídeo 03, valores finales

- Vídeo 04

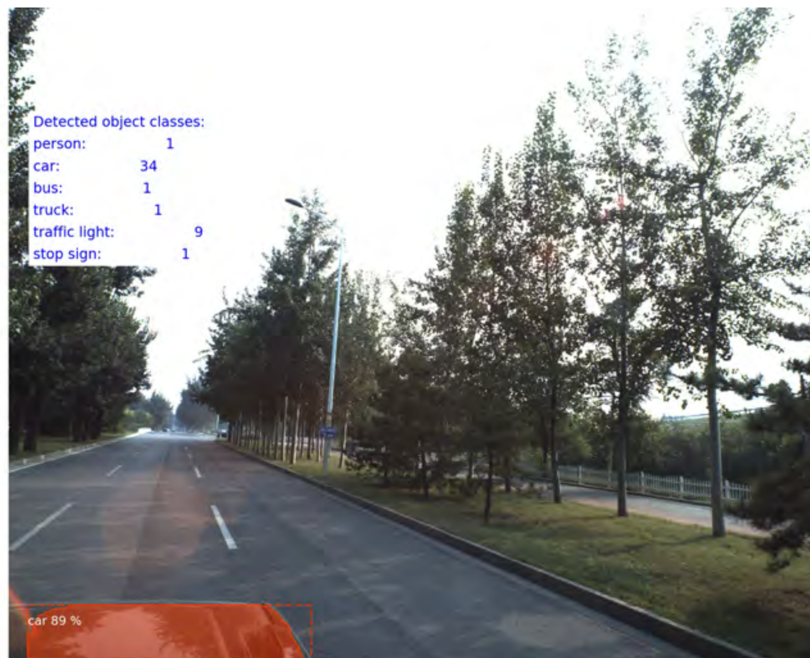


FIGURA 5.44. Vídeo 04, valores finales

- Vídeo 05

FIGURA 5.45. Vídeo 05, valores finales

- Vídeo 06

FIGURA 5.46. Vídeo 06, valores finales

- Vídeo 07

FIGURA 5.47. Vídeo 07, valores finales

- Vídeo 08

FIGURA 5.48. Vídeo 08, valores finales

- Vídeo 09

FIGURA 5.49. Vídeo 09, valores finales

- Vídeo 10

FIGURA 5.50. Vídeo 10, valores finales

- Vídeo 11

FIGURA 5.51. Vídeo 11, valores finales

El vídeo completo se puede visitar en Videos procesados y extraídos de CVPR 2018 WAD Video Segmentation Challenge, el cuál incluye las etiquetas y el recuento de objetos detectados.

Si se quiere revisar el vídeo con simplemente los objetos detectados, se puede visitar Vídeo compuesto con imágenes y objetos detectados de CVPR 2018.

CAPÍTULO 6

Conclusiones

Como el lector habrá ido comprobando a través de las descripciones de esta memoria, el procesado de imagen mediante algoritmos implementados en ordenadores es un tema en completa actualidad y desarrollo. Al margen ya de literatura técnica, prácticamente todos los días aparecen noticias de actualidad sobre inteligencia artificial aplicada en, por ejemplo, conducción autónoma. Esta particular aplicación utiliza de una manera intensa el procesado de imagen para extraer metadatos que permita sustituir a un conductor de un vehículo por una máquina entrenada para suplirlo e incluso eliminar los errores humanos de la conducción.

Las conclusiones principales del trabajo son las siguientes:

1. El estado del arte sobre redes neuronales y procesado de imagen está continuamente variando incluyendo mejoras y diferentes herramientas para lograr que el aprendizaje automatizado tenga cada vez más potencia.

2. Dado que importantes universidades y empresas están apostando por esta rama de las matemáticas, es a día de hoy una de las más demandadas laboralmente y con mucha proyección de futuro ya sea como investigación o aplicada a industria.

3. Existen herramientas *opensource* que permiten trabajar de forma cómoda con numerosas variantes de redes neuronales convolucionales. Además, la comunidad de aportaciones es cada día mayor, logrando comenzar con modelos potentes ya probados como punto de partida para ampliar conocimiento.

4. Mask-RCNN, en sus diversas variantes implementadas por la comunidad, es una potente Red Neuronal Convolucional basada en Regiones que permite un procesado de imágenes para las 80 clases COCO, siendo éstas un resumen de los objetos más importantes que

podemos encontrar en diversos entornos. Además, añadiendo clases adicionales se podría entrenar la red para la detección de diferentes tipos de objetos.

5. Con los resultados de salida de Mask-RCNN se pueden definir numerosos algoritmos de postprocesado para extraer metainformación de las propias imágenes empleado en vigilancia, seguridad, detección y conteo de objetos.

6. La complejidad de estos algoritmos no tiene límite, desde una aplicación sencilla hasta un seguimiento exhaustivo de objetos en movimiento mediante procesado adicional como por ejemplo con un Filtro de Kalman son posibles de forma sencilla en entornos científicos.

7. Cuando el número de objetos detectados y clasificados es muy grande la complejidad del postprocesado hace realmente difícil tener resultados cualitativos con una alta precisión. Por ejemplo en el cálculo de trayectorias cuando existen muchos objetos es muy sencillo que unas enmascaren a otras por la alta densidad. Sería recomendable poder discernir entre diferentes objetos de manera más precisa para así seguir el objeto de interés en cuestión.

8. Aunque Python es un lenguaje muy versátil pierde rendimiento frente a otros lenguajes no interpretados, por lo que para construir una aplicación en tiempo real deberíamos migrar la existente a otras plataformas más optimizadas como podría ser C++ o C#.

A nivel personal este trabajo me ha permitido continuar el estudio que hice durante la ingeniería en redes neuronales tipo MLP enfocadas a compresión de vídeo en tiempo real mediante FPGAs. En los cerca de 12 años que han pasado desde mi estudio inicial este campo ha tenido un avance espectacular de manera que tanto los algoritmos como los resultados que se obtienen hoy en día no tienen nada que ver con lo que en su momento estudié. Durante el desarrollo de la parte narrativa de la memoria y en la parte práctica he podido refrescar y aplicar conocimientos adquiridos, por supuesto, durante el Grado de Matemáticas y el Máster en Matemáticas Avanzadas así como los ya guardados en las neuronas biológicas pero aprendidos en la Ingeniería.

COCO Tools

El conjunto de datos COCO[1] es un conjunto orientado a la detección, segmentación y marcado de objetos a gran escala. COCO se ofrece mediante licencia *Creative Commons Attribution 4.0 License.* y se puede obtener más información en la web oficial `http://cocodataset.org/#home`.

Desde la web de COCO se pueden obtener las bibliotecas de funciones así como todos los conjuntos de imágenes desde el año 2014 hasta 2017, estando a día de hoy pendientes de actualización las imágenes de 2018. Cada año se ofrecen tres tipos de imágenes diferentes: imágenes de entrenamiento, validación y comprobación. La presente memoria se ha realizado con los conjuntos de 2017.

COCO [48] posee las siguientes características:

- Segmentado de objetos
- Reconocimiento en un contexto
- Segmentación de *cosas*
- 330k imágenes (>200k etiquetadas)
- 1.5 millones de objetos
- 80 categorías de objetos
- 5 leyendas por imagen
- 250K partes del cuerpo de personas

[1] Common Objects in COntext

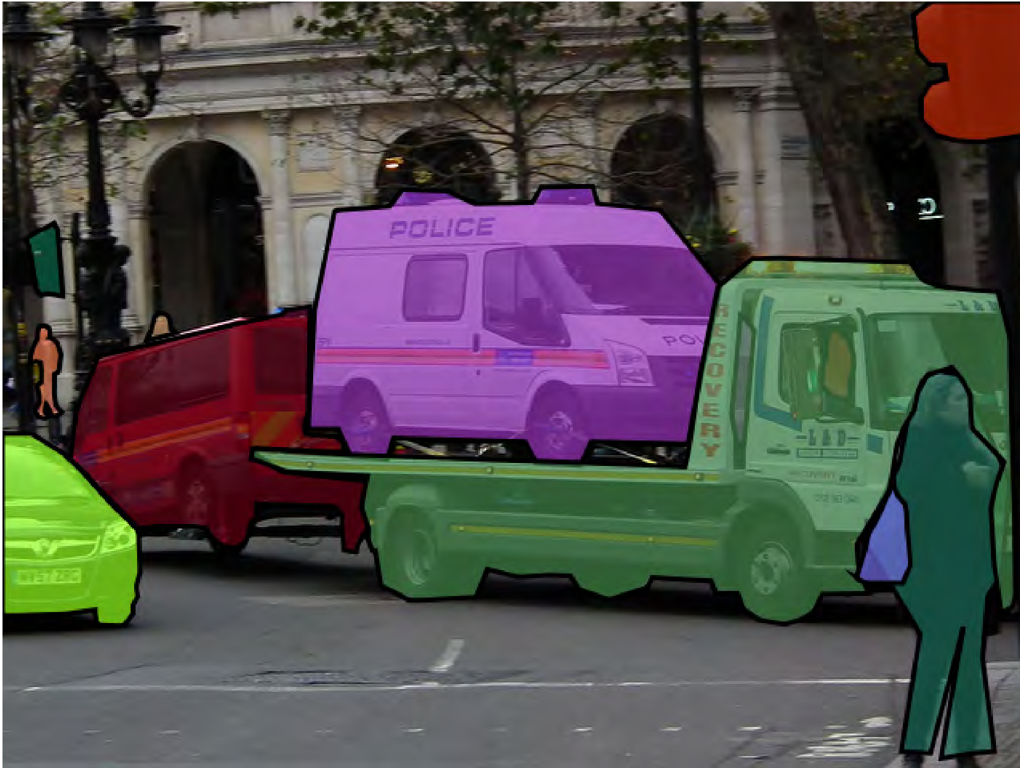


FIGURA A.1. Ejemplo de imagen en COCO

Tanto las imágenes de entrenamiento como las de validación incluyen para cada una de ellas varias entradas de índices llamada *annotations* y la imagen propiamente dicha. Por ejemplo, tomemos la imagen “000000037777"del conjunto de validación.

FIGURA A.2. Imagen “COCO 37777” del conjunto de validación de 2017

Esta imagen tendría las siguientes entradas en los ficheros .json de información:

- Metadatos de la imagen: *´´license": 1,"file_name": ´´000000037777.jpg","coco_url":* http://images.cocodataset.org/val2017/000000037777.jpg*,´´height": 230,"width": 352,´´date_captured": ´´2013-11-14 20:55:31",´´flickr_url":* http://farm9.staticflickr.com/8429/7839199426_f6d48aa585_z.jpg*,´´id": 37777*

- Captions (leyenda) de la imagen: *´´image_id": 37777,´´id": 601877,´´caption": ´´A kitchen and dining area decorated in white."*

- Instancias de objetos: *"segmentation": [[26.5,229.25,63.0,215.25,72.5,215.25,80.5,217.75,88.0, 224.25,86.5,226.25,76.5,229.75]],"area": 546.375,"iscrowd": 0,"image_id": 37777,"bbox": [26.5,215.25, 61.5,14.5],"category_id": 62*[2]*,"id": 100948*

Con todos los elementos contenidos en los ficheros .json y la imagen en sí misma, se construye una figura con metadatos que servirán para o bien entrenar o, en el caso de esta imagen, validar la red neuronal ya entrenada.

FIGURA A.3. Imagen "COCO 37777" con metadatos y máscaras

Todas las imágenes se pueden encontrar en la misma web de COCO en el apartado *explore*: http://cocodataset.org/#explore. Además, al buscar una imagen aparece toda la información y los posibles tipos de objetos que han sido etiquetados.

[2] La categoría 62 se refiere al objeto "silla"

FIGURA A.4. Imagen “COCO 37777” con etiquetas

A.1. Categorías COCO

El conjunto COCO tiene definidas 80 categorías de objetos. Cada categoría tiene un número de índice de forma que si queremos añadir o retirar esa categoría de nuestra red, simplemente hay que trazarla con su número y eliminarlo de la lista. Además existen 91 categorías adicionales para objetos no contables (distinción siempre más destacada en el idioma inglés).

La lista completa se puede consultar aquí: https://github.com/nightrome/cocostuff/blob/master/labels.md

La importación de estas clases al código de detección es muy sencilla:

```
1 # COCO Class names
2 # Index of the class in the list is its ID. For example, to get ID of
3 # the teddy bear class, use: class\_names.index('teddy bear')
4 class\_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
5 'bus', 'train', 'truck', 'boat', 'traffic light',
6 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
7 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
```

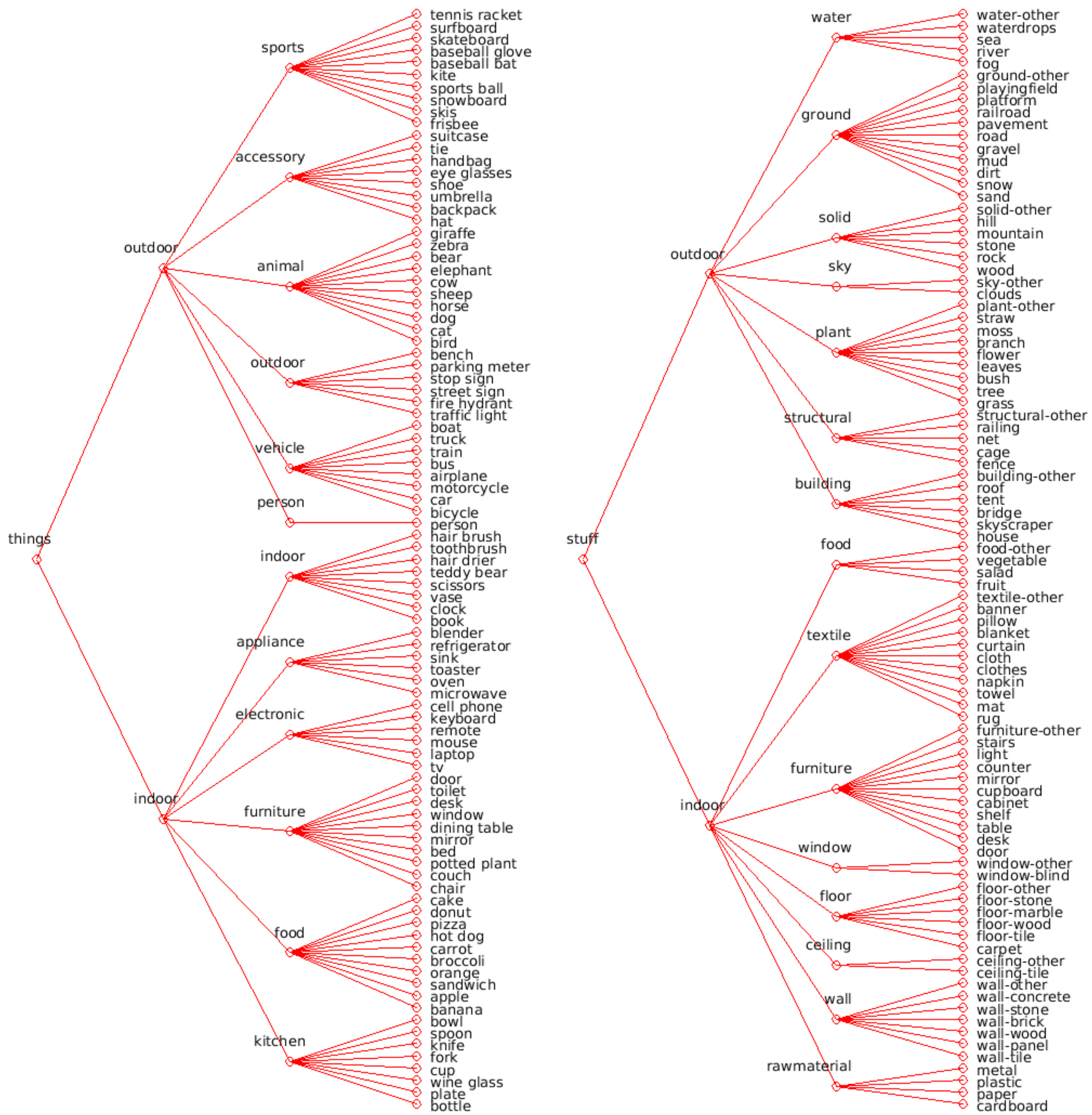


FIGURA A.5. Categorías de COCO

```
'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
'kite', 'baseball bat', 'baseball glove', 'skateboard',
'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
```

```
16 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
17 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
18 'teddy bear', 'hair drier', 'toothbrush']
```

O bien:

```
1 print("Image Count: {}".format(len(dataset.image_ids)))
2 print("Class Count: {}".format(dataset.num_classes))
3 for i, info in enumerate(dataset.class_info):
4 print("{:3}. {:50}".format(i, info['name']))
```

A.2. Métricas usadas en la detección COCO

Existen 12 métricas empleadas para caracterizar las prestaciones de un detector de objetos basados en COCO además de la precisión de las cajas que engloban objetos.

A.2.1. Intersección sobre Unión

En primer lugar se define el parámetro *Intersección sobre Unión (IoU)*[3] que, en la detección de objetos, mide la precisión de un detector en un conjunto de datos particular [49]. Cualquier algoritmo que calcula cajas alrededor de objetos puede tener una salida evaluada usando IoU.

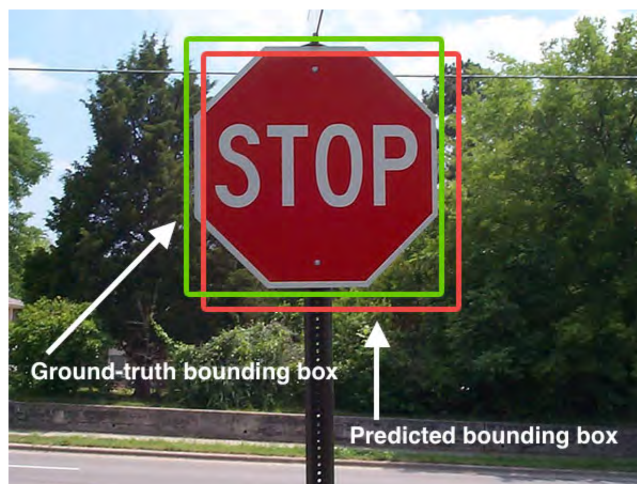


FIGURA A.6. Ejemplo de IoU, en rojo la caja fruto de la predicción y en verde la caja correcta

Numéricamente hablando, $IoU = \frac{AreaofOverlap}{AreaofUnion}$.

Donde *Area of Overlap* es el área que pertenece a la intersección entre la predicción y la realidad, mientras que *Area of Union* es el área suma de la predicción y la realidad.

[3]Por sus siglas en inglés Intersection over Union

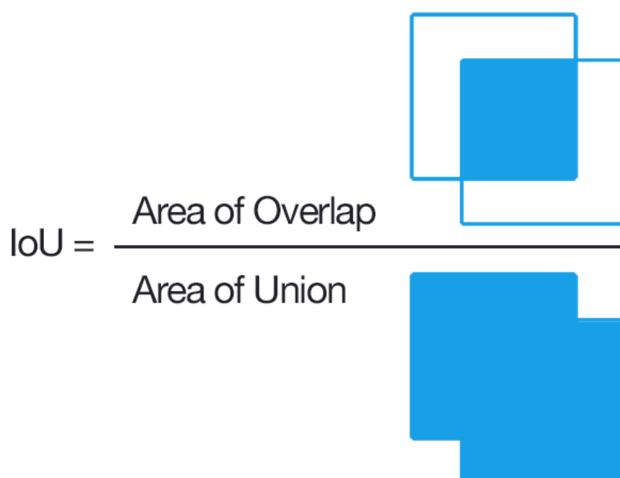


FIGURA A.7. Cálculo de IoU

A.2.2. Precisión promediada

La precisión promediada, en adelante AP[4], se calcula para aquellas predicciones consideradas correctas, esto es, que tienen un IoU mayor que un umbral [50].

En particular, en el conjunto COCO, se calcula para un IoU desde [0.50:0.05:0.95] así como con un IoU=0.50 y 0.75.

A.2.3. AP escalada

Se calcula la AP para tres tipos de objetos en función de su área: área<32, 32<área<96 y área>96; con unas proporciones de 41, 34 y 24 %. El área se calcula a partir del número de píxeles en la máscara de segmentado.

A.2.4. Sensibilidad promediada

La sensibilidad promediada, en adelante AR[5] es el porcentaje promediado de las instancias relevantes que han sido recogidas sobre el total de instancias.

La diferencia entre AP y AR es que la AP mide cuántos objetos hemos detectado de forma correcta frente al conjunto de detecciones, mientras que AR mide cuántos objetos hemos detectado de forma correcta frente a todos los objetos posibles. Por ejemplo, en una imagen hay 10 personas, siendo 5 hombres y 5 mujeres. Nuestro detector identifica y clasifica a 4 de ellas como hombres, habiendo 2 realmente hombres (H_H) y 2 mujeres (H_M). La AP sería $AP = \frac{H_H}{H_H+H_M} = \frac{2}{2+2} = 0{,}5$, mientras que la AR sería $AP = \frac{H_H}{Total} = \frac{2}{10} = 0{,}2$.

En particular, en COCO se calcula AR para 1, 10 y 100 detecciones por imagen.

[4]Por sus siglas en inglés Average Precision

[5]Por sus siglas en inglés Average Recall

A.2.5. AR escalada

Se calcula la AR para tres tipos de objetos en función de su área: área<32, 32<área<96 y área>96; con unas proporciones de 41, 34 y 24 %. El área se calcula a partir del número de píxeles en la máscara de segmentado.

A.2.6. Cálculo de las métricas

Dentro de la biblioteca COCO existen ya programadas las funciones para el cálculo de las métricas que se pueden llamar de forma muy sencilla:

```
# Evaluate
cocoEval = COCOeval(coco, coco_results, eval_type)
cocoEval.params.imgIds = coco_image_ids
cocoEval.evaluate()
cocoEval.accumulate()
cocoEval.summarize()

#IoU: Intersection over Union

print("Prediction time: {}. Average {}/image".format(
t_prediction, t_prediction/len(image_ids)))
print("Total time: ", time.time() - t_start)
```

Para unos pesos arbitrarios y sobre 10 imágenes aleatorias del conjunto de validación, darían los siguientes resultados:

```
Running per image evaluation...
Evaluate annotation type *bbox*
DONE (t=0.02s).
Accumulating evaluation results...
DONE (t=0.06s).
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.400
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.622
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.440
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.181
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.427
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.519
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.337
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.407
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.407
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.186
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.430
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.535
```

```
Prediction time: 9.222646236419678. Average 0.9222646236419678/image
Total time:  9.40375804901123
```

A.3. Uso de la biblioteca COCO

Como se comentó anteriormente, COCO es de uso libre y se puede descargar desde el repositorio github https://github.com/cocodataset/cocoapi para ser usadas en entornos Matlab o Python. En el caso de esta memoria el código se realizó sobre Python, por lo que se instaló únicamente la parte dedicada al entorno Python.

La instalación es muy sencilla:

```
To install:
-For Matlab, add coco/MatlabApi to the Matlab path (OSX/Linux binaries provided)
-For Python, run "make" under coco/PythonAPI
-For Lua, run "luarocks make LuaAPI/rocks/coco-scm-1.rockspec" under coco/
```

APÉNDICE B

GUÍA INSTALACIÓN DE LAS HERRAMIENTAS EMPLEADAS EN LA MEMORIA

Dentro de la información contenida en el repositorio de Git para el proyecto de Karol Majek dedicado a Mask RCNN existe una lista de requisitos a nivel de software necesarios para poder comenzar con los tutoriales o ficheros incluidos en el repositorio [51].

Este apéndice repasa la lista de módulos o programas necesarios para hacer una correcta instalación aunque bien es cierto que debido a las versiones e incompatibilidades entre unas y otras, suele ser más laborioso que simplemente seguir una guía de instalación.

Asímismo se incluye también la lista de software adicional que se ha usado, como por ejemplo, los conversores de video a JPG y viceversa empleados en el procesado de los vídeos.

B.1. Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada *Python Software Foundation License*, que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.

Aunque se indica en [51] que se debe instalar la versión de Pythonn 3.4+, es mi recomendación instalar siempre la última versión disponible. En el momento de la edición de esta memoria, la última versión disponible en `https://www.python.org/` es la versión de Python 3.6.5.

B.1.1. Anaconda

Anaconda es un distribución de los lenguajes Python y R libre y abierto distribución de código abierto de la Python, utilizada en ciencia de datos, y aprendizaje automático (machine learning). Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos. Está orientado a simplificar el despliegue y administración de los paquetes de software. La diferentes versiones de los paquetes se administran mediante el sistema de administración del paquete conda, el cual lo hace bastante sencillo de instalar, ejecutar, y actualizar software de ciencia de datos y aprendizaje automático como ser Scikit-team, TensorFlow y SciPy. La distribución Anaconda es utilizada por 6 millones de usuarios e incluye más de 250 paquetes de ciencia de datos válidos para Windows, Linux y MacOS.

No se indica que se instale Anaconda, pero es altamente recomendable para usar el IDE y tener más simplificada la instalación y configuración del resto de paquetes que se van a usar.

En el momento de la edición de esta memoria, la última versión disponible en https://anaconda.org/ es la versión 5.1 para Python 3.6.

B.2. CUDA

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.

Gracias a millones de GPUs CUDA vendidas hasta la fecha, miles de desarrolladores, científicos e investigadores están encontrando innumerables aplicaciones prácticas para esta tecnología en campos como el procesamiento de vídeo e imágenes, la biología y la química computacional, la simulación de dinámica de fluidos, la reconstrucción de imágenes TC, el análisis sísmico o el trazado de rayos, entre otras.

Los sistemas informáticos están pasando de realizar el procesamiento central en la CPU a realizar coprocesamiento repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las GPUs GeForce, ION Quadro y Tesla GPUs, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones.

Se puede comenzar a trabajar sin usar una GPU nVIDIA (usando la CPU por ejemplo) pero no es nada recomendable por los largos tiempos durante el entrenamiento de la red. Por esta razón la mejor opción es utilizar algún tipo de máquina equipado con una GPU nVIDIA y descargar el conjunto de herramientas CUDA desde https://developer.nvidia.com/cuda-downloads. En el momento de esta memoria y para la gráfica utilizada (GTX 960 para procesado y GTX 1070 para el entrenamiento debido a la mayor demanda de memoria RAM) la versión de CUDA descargada es la 9.0 ya que aunque existen versión más actualizadas, Tensorflow sólo se ejecuta bajo la 9.0 frente a 9.2.

FIGURA B.1. Detalle de modelo de gráfica usada en la memoria para el procesado de imágenes

B.2.1. nVIDIA cuDNN

Como complemento a CUDA existen las cuDNN[1] que son básicamente bibliotecas de primitivas aceleradas para GPU orientadas a redes neuronales profundas.

En el momento de la edición de esta memoria, la última versión disponible compatible con CUDA 9.0 es la versión descargable desde https://developer.nvidia.com/cudnn (atención, registro obligatorio para poder descargarlas). Hay que ser cuidadoso a la hora de elegir la versión ya que existen cuDNN para todas las versiones de CUDA con diferentes fechas y subversiones. Para CUDA 9.0 hay que descargar la versión cuDNN v7.0 y copiar dentro del entorno CUDA los archivos descargados y contenidos en el .zip.

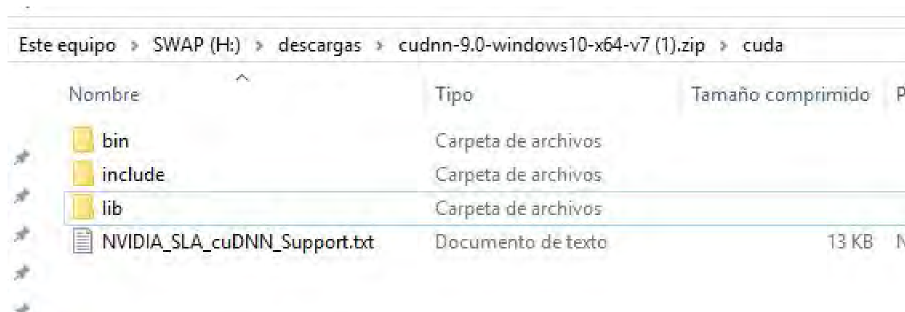


FIGURA B.2. Contenido .zip cuDNN

[1]CUDA Deep Neural Network

B.3. TensorFlow

TensorFlow es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. Actualmente es utilizado tanto para la investigación como para la producción de productos de Google frecuentemente remplazando el rol de su predecesor de código cerrado, DistBelief. TensorFlow fue originalmente desarrollado por el equipo de Google Brain para uso interno en Google antes de ser publicado bajo la licencia de código abierto Apache 2.0 el 9 de noviembre de 2015.

La instalación es muy sencilla y se puede realizar siguiendo los pasos en https://www.tensorflow.org/install/install_windows. Dependiendo de si queremos trabajar con la GPU (opción recomendada) o bien con la CPU habrá que ejecutar una u otra parte del tutorial de la web anterior. Además, en la web de TensorFlow se expone una pequeña lista de comandos a ejecutar para comprobar que se ha instalado correctamente:

```
1 import tensorflow as tf
2 hello = tf.constant('Hello, TensorFlow!')
3 sess = tf.Session()
4 print(sess.run(hello))
```

Si la salida es correcta e igual a

```
Hello, TensorFlow!
```

La instalación habrá sido satisfactoria.

B.4. Resto de módulos necesarios

Aquí se engloban todos aquellos módulos, bibliotecas o archivos necesarios pero que se pueden ir instalando según se necesiten por las funciones de los ficheros .py.

Por ejemplo serían necesarios Keras, Jupyter Notebook, etc. La mejor opción es desde el entorno anaconda ir instalando cada uno mediante

```
pip install keras
pip install Jupyter Notebook
...
...
```

Existe otra opción de instalanción mediante las instrucciones

```
conda install ...
```

Sin embargo en ocasiones me ha dado problemas por no haber instalado correctamente las dependencias, ya que instalaba versiones antiguas o demasiado nuevas no compatibles. Mi recomendación es funcionar mediante *pip install*.

Una vez todos los módulos se hayan instalado correctamente se puede cargar mediante *Jupyter notebook* los archivos demo del repositorio y probar a ejecutarlos todos para comprobar que no existen errores y que los resultados son razonables. Cuando todos los archivos .ipynb se hayan ejectuado correctamente ya se puede proceder a realizar modificaciones, procesado de imágenes y entrenamientos personalizados con los conjuntos de imágenes.

B.5. Free Video to JPG Converter

Se trata de una herramienta totalmente gratuita destinada a extraer fotogramas de los vídeos caseros. De uso muy sencillo e intuitivo permite extraer fotogramas con una cadencia determinada, cada un número de segundos o bien una cantidad fija a partir de un vídeo con independencia de la duración.

Se puede descargar desde la web de DVDVideoSoft e incluye un pequeño tutorial de sobre cómo trabajar con él en el apartado de guías.

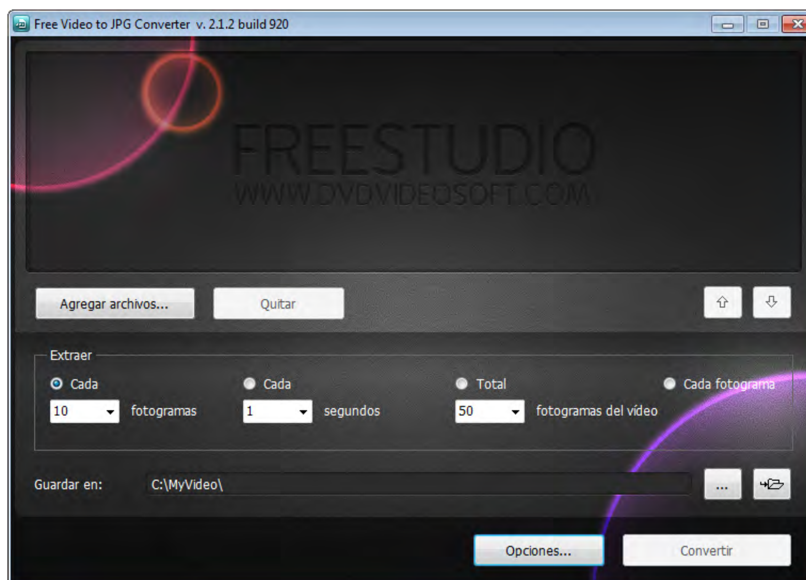


FIGURA B.3. Consola principal *Free Video to JPG Converter*

Las extracciones de esta memoria desde un vídeo a imagen fija .jpg se han realizado con esta herramienta.

B.6. OpenShot Video Converter

OpenShot es un editor de vídeo multi-plataforma con soporte para Linux, Mac, y Windows. OpenShot es compatible con los siguientes sistemas operativos: Linux (la mayoría de distribuciones), Windows (versiones 7, 8, y 10+), y OS X (versiones 10.9+). Los archivos de proyecto también son multiplataforma, lo que significa que puedes guardar un proyecto de vídeo en un SO, y abrirlo en otro distinto. Todas las características de edición de vídeo están disponibles en todas las plataformas. Basado en la potente biblioteca FFmpeg , OpenShot puede leer y escribir la mayoría de los formatos de video e imagen.

Se puede descargar desde la web Openshot donde existen guías y ejemplos de aplicación.

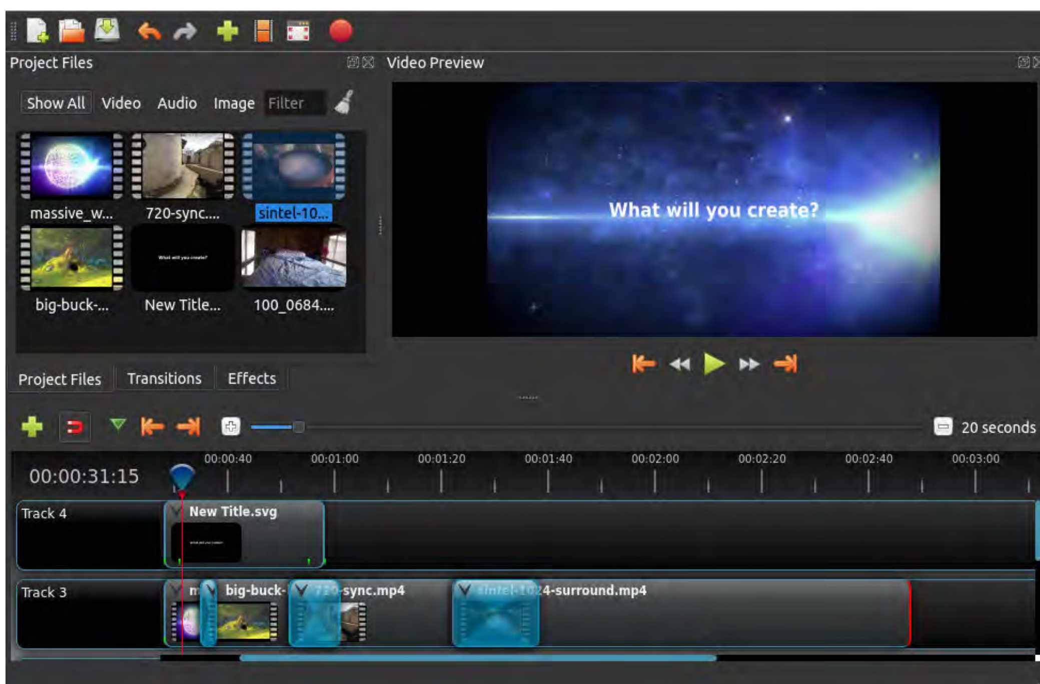


FIGURA B.4. Consola principal *OpenShot Video Converter*

Las conversiones de esta memoria desde formato imagen a vídeo se han realizado con esta herramienta.

APÉNDICE

CONVOLUCIÓN

Se define el operador convolución de dos funciones f y g como la integral del producto de ambas después de invertir una y desplazarla una distancia t, esto es

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\eta)g(t-\eta)d\eta$$

Mientras que en el dominio discreto se definiría de una manera equivalente

$$f[n] * g[n] = \sum_{m} f[m]g[n-m]$$

C.1. Propiedades

El operador convolución, ya sea en el dominio discreto o continuo, tiene las siguientes propiedades (sin demostración):

- Conmutativa

$$f * g = g * f$$

- Asociativa

$$f * (g * h) = (f * g) * h$$

- Distributiva

$$(f + g) * h = (f * h) + (g * h)$$

- Asociativa por un escalar

$$z(f * g) = (zf) * g = f * (zg)$$

$$\forall z \in \mathbb{Z}$$

- Convolución por una distribución Delta[1]

$$f * \delta = f$$

- Inversa

$$S^{(-1)} * S = \delta$$

- Conjugada compleja

$$\overline{f * g} = \overline{f} * \overline{g}$$

- Teorema de la convolución

$$TF(f * g) = kTF(f)TF(g)$$

Donde $TF(f)$ denota la Transformada de Fourier de f y k es una constante que depende de la normalización de dicha transformada

- Conmutativa con la traslación, de importancia en el procesado de imagen ya que de esta manera la traslación se puede aplicar en la imagen o en el filtro convolucional antes de la operación.

$$\tau_x(f * g) = (\tau_x f) * g = f * (\tau_x g)$$

Donde τ_x es la traslación de f definida de la siguiente manera

$$(\tau_x f)(y) = f(y - x)$$

[1]Se define la distribución Delta como $\delta_a(x) = \delta(x-a)$ y $f(a) = \int_{-\infty}^{\infty} \delta(x-a)f(x)$

C.2. Ejemplos de aplicación del operador convolución

La siguiente lista, no exhaustiva, muestra los ejemplos en los que se podría aplicar el operador convolución así como una pequeña descripción.

- **Convolución en sistemas lineales invariantes.** Por ejemplo vamos a ver la convolución de una función escalón con e^{-t}.

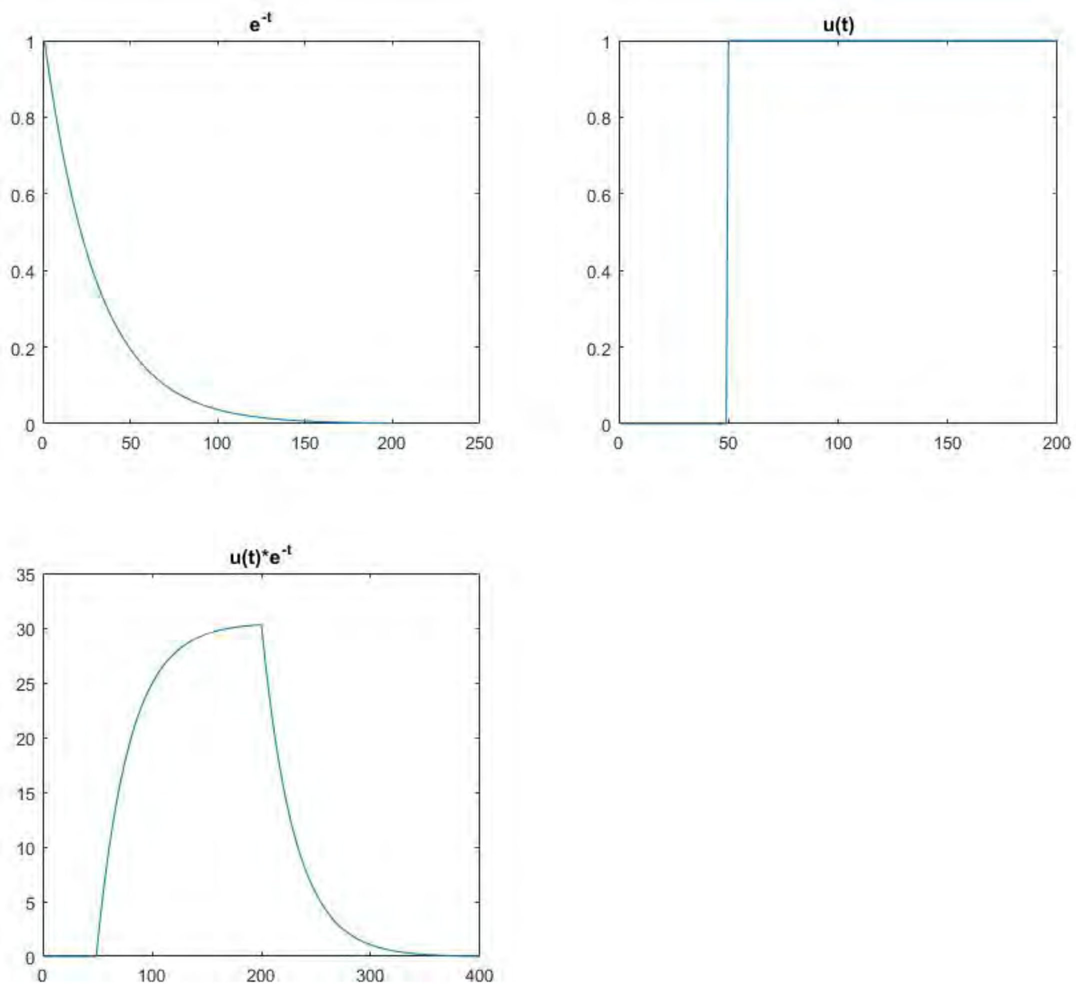


FIGURA C.1. Convolución de $u(t) * e^{-t}$

Esta convolución en sistemas físicos es la producida por una carga/descarga de condensador frente a un inicio del sistema.

- **Convolución en varias dimensiones [60].** Cuando se aplica el operador a imágenes se puede extraer información mediante el uso de un *kernel* específico para cada aplicación concreta. Es similar a las operaciones vistas en 4.2.1.2.
- **Redes Neuronales Convolucionales.** Se puede consultar el capítulo dedicado en esta memoria (4) para un detalle más completo.
- **Teoría de la probabilidad.** La distribución de probabilidad el resultado de la suma de dos variables aleatorias independientes se puede calcular como la convolución de sus distribuciones de forma individual.
- **Efectos producidos por la transferencia de energía entre protones y electrónica.** Dado un patrón de efectos en electrónica de una partícula de la materia se puede calcular cuál sería el efecto de varias mediante la convolución entre la distribución de la llegada de partículas y la distribución de efectos. En electrónica digital, por ejemplo, los efectos serían una distribución de errores en la transmisión.

Código fuente

Este apéndice recoge todos los archivos de código fuente modificados de alguna manera para la redacción de la memoria. El resto de archivos se pueden usar *as is* desde el repositorio de Mask RCNN [51]. Para facilitar su lectura el criterio a la hora de realizar el código ha sido de mejorar la visibilidad y seguimiento frente a una optimización computacional propiamente dicha. De esta manera se pretende que, mediante líneas del tipo *print* en diferentes puntos de test se pudieran ver variables en tiempo real. Por ello algunos cálculos están separados en varias líneas, para mejorar el depurado de las funciones.

D.1. Inspect Data

```
1 import os
2 import sys
3 import itertools
4 import math
5 import logging
6 import json
7 import re
8 import random
9 from collections import OrderedDict
10 import numpy as np
11 import matplotlib
12 import matplotlib.pyplot as plt
13 import matplotlib.patches as patches
14 import matplotlib.lines as lines
15 from matplotlib.patches import Polygon
16 
17 import utils
```

```
18 import visualize
19 from visualize import display_images
20 import model as modellib
21 from model import log
22
23 %matplotlib inline
24
25 ROOT_DIR = os.getcwd()
26
27 # Run one of the code blocks
28
29 # Shapes toy dataset
30 # import shapes
31 # config = shapes.ShapesConfig()
32
33 # MS COCO Dataset
34 import coco
35 config = coco.CocoConfig()
36 COCO_DIR = "../../dataset"  # TODO: enter value here
37
38 # Load dataset
39 if config.NAME == 'shapes':
40 dataset = shapes.ShapesDataset()
41 dataset.load_shapes(500, config.IMAGE_SHAPE[0], config.IMAGE_SHAPE[1])
42 elif config.NAME == "coco":
43 dataset = coco.CocoDataset()
44 dataset.load_coco(COCO_DIR, "train")
45
46 # Must call before using the dataset
47 dataset.prepare()
48
49 print("Image Count: {}".format(len(dataset.image_ids)))
50 print("Class Count: {}".format(dataset.num_classes))
51 for i, info in enumerate(dataset.class_info):
52 print("{:3}. {:50}".format(i, info['name']))
53
54 # Load and display random samples
55 image_ids = np.random.choice(dataset.image_ids, 4)
56 for image_id in image_ids:
57 image = dataset.load_image(image_id)
58 mask, class_ids = dataset.load_mask(image_id)
59 visualize.display_top_masks(image, mask, class_ids, dataset.class_names)
60
61 # Load random image and mask.
62 image_id = random.choice(dataset.image_ids)
63 image = dataset.load_image(image_id)
64 mask, class_ids = dataset.load_mask(image_id)
```

```
65 # Compute Bounding box
66 bbox = utils.extract_bboxes(mask)
67
68 # Display image and additional stats
69 print("image_id ", image_id, dataset.image_reference(image_id))
70 log("image", image)
71 log("mask", mask)
72 log("class_ids", class_ids)
73 log("bbox", bbox)
74 # Display image and instances
75 visualize.display_instances(image, bbox, mask, class_ids, dataset.class_names)
76
77 # Load random image and mask.
78 image_id = np.random.choice(dataset.image_ids, 1)[0]
79 image = dataset.load_image(image_id)
80 mask, class_ids = dataset.load_mask(image_id)
81 original_shape = image.shape
82 # Resize
83 image, window, scale, padding = utils.resize_image(
84 image,
85 min_dim=config.IMAGE_MIN_DIM,
86 max_dim=config.IMAGE_MAX_DIM,
87 padding=config.IMAGE_PADDING)
88 mask = utils.resize_mask(mask, scale, padding)
89 # Compute Bounding box
90 bbox = utils.extract_bboxes(mask)
91
92 # Display image and additional stats
93 print("image_id: ", image_id, dataset.image_reference(image_id))
94 print("Original shape: ", original_shape)
95 log("image", image)
96 log("mask", mask)
97 log("class_ids", class_ids)
98 log("bbox", bbox)
99 # Display image and instances
100 visualize.display_instances(image, bbox, mask, class_ids, dataset.class_names)
101
102 image_id = np.random.choice(dataset.image_ids, 1)[0]
103 image, image_meta, bbox, mask = modellib.load_image_gt(
104 dataset, config, image_id, use_mini_mask=False)
105
106 log("image", image)
107 log("image_meta", image_meta)
108 log("bbox", bbox)
109 log("mask", mask)
110
111 display_images([image]+[mask[:,:,i] for i in range(min(mask.shape[-1], 7))])
```

```
112
113 visualize.display_instances(image, bbox[:,:4], mask, bbox[:,4], dataset.class_names)
114
115 # Add augmentation and mask resizing.
116 image, image_meta, bbox, mask = modellib.load_image_gt(
117 dataset, config, image_id, augment=True, use_mini_mask=True)
118 log("mask", mask)
119 display_images([image]+[mask[:,:,i] for i in range(min(mask.shape[-1], 7))])
120
121 mask = utils.expand_mask(bbox, mask, image.shape)
122 visualize.display_instances(image, bbox[:,:4], mask, bbox[:,4], dataset.class_names)
123
124 # Generate Anchors
125 anchors = utils.generate_pyramid_anchors(config.RPN_ANCHOR_SCALES,
126 config.RPN_ANCHOR_RATIOS,
127 config.BACKBONE_SHAPES,
128 config.BACKBONE_STRIDES,
129 config.RPN_ANCHOR_STRIDE)
130
131 # Print summary of anchors
132 num_levels = len(config.BACKBONE_SHAPES)
133 anchors_per_cell = len(config.RPN_ANCHOR_RATIOS)
134 print("Count: ", anchors.shape[0])
135 print("Scales: ", config.RPN_ANCHOR_SCALES)
136 print("ratios: ", config.RPN_ANCHOR_RATIOS)
137 print("Anchors per Cell: ", anchors_per_cell)
138 print("Levels: ", num_levels)
139 anchors_per_level = []
140 for l in range(num_levels):
141 num_cells = config.BACKBONE_SHAPES[l][0] * config.BACKBONE_SHAPES[l][1]
142 anchors_per_level.append(anchors_per_cell * num_cells // config.RPN_ANCHOR_STRIDE**2)
143 print("Anchors in Level {}: {}".format(l, anchors_per_level[l]))
144
145 ## Visualize anchors of one cell at the center of the feature map of a specific level
146
147 # Load and draw random image
148 image_id = np.random.choice(dataset.image_ids, 1)[0]
149 image, image_meta, _, _ = modellib.load_image_gt(dataset, config, image_id)
150 fig, ax = plt.subplots(1, figsize=(10, 10))
151 ax.imshow(image)
152 levels = len(config.BACKBONE_SHAPES)
153
154 for level in range(levels):
155 colors = visualize.random_colors(levels)
156 # Compute the index of the anchors at the center of the image
157 level_start = sum(anchors_per_level[:level]) # sum of anchors of previous levels
158 level_anchors = anchors[level_start:level_start+anchors_per_level[level]]
```

```
159 print("Level {}. Anchors: {:6}  Feature map Shape: {}".format(level, level_anchors.shape
    [0],
160 config.BACKBONE_SHAPES[level]))
161 center_cell = config.BACKBONE_SHAPES[level] // 2
162 center_cell_index = (center_cell[0] * config.BACKBONE_SHAPES[level][1] + center_cell[1])
163 level_center = center_cell_index * anchors_per_cell
164 center_anchor = anchors_per_cell * (
165 (center_cell[0] * config.BACKBONE_SHAPES[level][1] / config.RPN_ANCHOR_STRIDE**2) \
166 + center_cell[1] / config.RPN_ANCHOR_STRIDE)
167 level_center = int(center_anchor)
168 
169 # Draw anchors. Brightness show the order in the array, dark to bright.
170 for i, rect in enumerate(level_anchors[level_center:level_center+anchors_per_cell]):
171 y1, x1, y2, x2 = rect
172 p = patches.Rectangle((x1, y1), x2-x1, y2-y1, linewidth=2, facecolor='none',
173 edgecolor=(i+1)*np.array(colors[level]) / anchors_per_cell)
174 ax.add_patch(p)
175 
176 # Create data generator
177 random_rois = 2000
178 g = modellib.data_generator(
179 dataset, config, shuffle=True, random_rois=random_rois,
180 batch_size=4,
181 detection_targets=True)
182 
183 # Get Next Image
184 if random_rois:
185 [normalized_images, image_meta, rpn_match, rpn_bbox, gt_boxes, gt_masks, rpn_rois, rois],
     \
186 [mrcnn_class_ids, mrcnn_bbox, mrcnn_mask] = next(g)
187 
188 log("rois", rois)
189 log("mrcnn_class_ids", mrcnn_class_ids)
190 log("mrcnn_bbox", mrcnn_bbox)
191 log("mrcnn_mask", mrcnn_mask)
192 else:
193 [normalized_images, image_meta, rpn_match, rpn_bbox, gt_boxes, gt_masks], _ = next(g)
194 
195 log("gt_boxes", gt_boxes)
196 log("gt_masks", gt_masks)
197 log("rpn_match", rpn_match, )
198 log("rpn_bbox", rpn_bbox)
199 image_id = image_meta[0][0]
200 print("image_id: ", image_id, dataset.image_reference(image_id))
201 
202 # Remove the last dim in mrcnn_class_ids. It's only added
203 # to satisfy Keras restriction on target shape.
```

```
204 mrcnn_class_ids = mrcnn_class_ids[:,:,0]
205
206 b = 0
207
208 # Restore original image (reverse normalization)
209 sample_image = modellib.unmold_image(normalized_images[b], config)
210
211 # Compute anchor shifts.
212 indices = np.where(rpn_match[b] == 1)[0]
213 refined_anchors = utils.apply_box_deltas(anchors[indices], rpn_bbox[b, :len(indices)] *
        config.RPN_BBOX_STD_DEV)
214 log("anchors", anchors)
215 log("refined_anchors", refined_anchors)
216
217 # Get list of positive anchors
218 positive_anchor_ids = np.where(rpn_match[b] == 1)[0]
219 print("Positive anchors: {}".format(len(positive_anchor_ids)))
220 negative_anchor_ids = np.where(rpn_match[b] == -1)[0]
221 print("Negative anchors: {}".format(len(negative_anchor_ids)))
222 neutral_anchor_ids = np.where(rpn_match[b] == 0)[0]
223 print("Neutral anchors: {}".format(len(neutral_anchor_ids)))
224
225 # ROI breakdown by class
226 for c, n in zip(dataset.class_names, np.bincount(mrcnn_class_ids[b].flatten())):
227 if n:
228 print("{:23}: {}".format(c[:20], n))
229
230 # Show positive anchors
231 visualize.draw_boxes(sample_image, boxes=anchors[positive_anchor_ids],
232 refined_boxes=refined_anchors)
233
234 # Show negative anchors
235 visualize.draw_boxes(sample_image, boxes=anchors[negative_anchor_ids])
236
237 # Show neutral anchors. They don't contribute to training.
238 visualize.draw_boxes(sample_image, boxes=anchors[np.random.choice(neutral_anchor_ids,
        100)])
239
240 if random_rois:
241 # Class aware bboxes
242 bbox_specific = mrcnn_bbox[b, np.arange(mrcnn_bbox.shape[1]), mrcnn_class_ids[b], :]
243
244 # Refined ROIs
245 refined_rois = utils.apply_box_deltas(rois[b].astype(np.float32), bbox_specific[:,:4] *
        config.BBOX_STD_DEV)
246
247 # Class aware masks
```

```
248 mask_specific = mrcnn_mask[b, np.arange(mrcnn_mask.shape[1]), :, :, mrcnn_class_ids[b]]
249
250 visualize.draw_rois(sample_image, rois[b], refined_rois, mask_specific, mrcnn_class_ids[b
        ], dataset.class_names)
251
252 # Any repeated ROIs?
253 rows = np.ascontiguousarray(rois[b]).view(np.dtype((np.void, rois.dtype.itemsize * rois.
        shape[-1])))
254 _, idx = np.unique(rows, return_index=True)
255 print("Unique ROIs: {} out of {}".format(len(idx), rois.shape[1]))
256
257 if random_rois:
258 # Dispalay ROIs and corresponding masks and bounding boxes
259 ids = random.sample(range(rois.shape[1]), 8)
260
261 images = []
262 titles = []
263 for i in ids:
264 image = visualize.draw_box(sample_image.copy(), rois[b,i,:4].astype(np.int32), [255, 0,
        0])
265 image = visualize.draw_box(image, refined_rois[i].astype(np.int64), [0, 255, 0])
266 images.append(image)
267 titles.append("ROI {}".format(i))
268 images.append(mask_specific[i] * 255)
269 titles.append(dataset.class_names[mrcnn_class_ids[b,i]][:20])
270
271 display_images(images, titles, cols=4, cmap="Blues", interpolation="none")
272
273 # Check ratio of positive ROIs in a set of images.
274 if random_rois:
275 limit = 10
276 temp_g = modellib.data_generator(
277 dataset, config, shuffle=True, random_rois=10000,
278 batch_size=1, detection_targets=True)
279 total = 0
280 for i in range(limit):
281 _, [ids, _, _] = next(temp_g)
282 positive_rois = np.sum(ids[0] > 0)
283 total += positive_rois
284 print("{:5} {:5.2f}".format(positive_rois, positive_rois/ids.shape[1]))
285 print("Average percent: {:.2f}".format(total/(limit*ids.shape[1])))
```

D.2. Entrenamiento de los pesos en terminal Anaconda

```
import os
import sys
import random
import math
import re
import time
import numpy as np
import cv2
import matplotlib
#import matplotlib.pyplot as plt
import scipy.misc
import sys


from config import Config
import utils
import model as modellib
import visualize
from model import log
import time
import coco

from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from pycocotools import mask as maskUtils
from coco import evaluate_coco

print("Fin de importaciones")



# Root directory of the project
ROOT_DIR = os.getcwd()

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")

# Path to COCO trained weights
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")

# Carpeta de imágenes para hacer el entrenamiento
TRAIN_DIR = os.path.join(ROOT_DIR, "../../dataset/train2017")

# Carpeta de imágenes para hacer la validación del modelo
```

```
46 VAL_DIR = os.path.join(ROOT_DIR, "../../dataset")
47
48 # Carpeta de imágenes para hacer la visualización de imágenes con objetos detectados
49 VAL_DIR2 = os.path.join(ROOT_DIR, "../../dataset/val2017")
50
51 COCO_DIR = "../../dataset"
52
53 TRAIN="YES"
54
55 #Número de EPOCHS por fase
56 PHASE_EPOCH = 100
57 #Número de steps por epoch
58 STEPS = 100
59
60
61 ############################################################
62 #  Configurations
63 ############################################################
64
65 class CocoConfig(Config):
66 """Configuration for training on MS COCO.
67 Derives from the base Config class and overrides values specific
68 to the COCO dataset.
69 """
70 # Give the configuration a recognizable name
71 NAME = "coco"
72
73 # We use a GPU with 12GB memory, which can fit two images.
74 # Adjust down if you use a smaller GPU.
75 IMAGES_PER_GPU = 1
76
77 # Uncomment to train on 8 GPUs (default is 1)
78 GPU_COUNT = 1
79
80 #Lo dejo a 2 para aligerar, lo está haciendo de 1000
81 # Use a small epoch since the data is simple
82 STEPS_PER_EPOCH = STEPS
83
84 # Number of classes (including background)
85 NUM_CLASSES = 1 + 80  # COCO has 80 classes
86
87
88 # Device to load the neural network on.
89 # Useful if you're training a model on the same
90 # machine, in which case use CPU and leave the
91 # GPU for training.
92 DEVICE = "/gpu:0"  # /cpu:0 or /gpu:0
```

```
93
94
95 ############################################################
96 #   Dataset
97 ############################################################
98
99 class CocoDataset(utils.Dataset):
100 def load_coco(self, dataset_dir, subset, class_ids=None,
101 class_map=None, return_coco=False):
102 """Load a subset of the COCO dataset.
103 dataset_dir: The root directory of the COCO dataset.
104 subset: What to load (train, val, minival, val35k)
105 class_ids: If provided, only loads images that have the given classes.
106 class_map: TODO: Not implemented yet. Supports maping classes from
107 different datasets to the same class ID.
108 return_coco: If True, returns the COCO object.
109 """
110 # Path
111 #He cambiado de a 2017!
112 image_dir = os.path.join(dataset_dir, "train2017" if subset == "train"
113 else "val2017")
114
115 # Create COCO object
116 #Cambiado de 2014 a 2017
117 json_path_dict = {
118 "train": "annotations/instances_train2017.json",
119 "val": "annotations/instances_val2017.json",
120 "minival": "annotations/instances_minival2017.json",
121 "val35k": "annotations/instances_valminusminival2017.json",
122 }
123 coco = COCO(os.path.join(dataset_dir, json_path_dict[subset]))
124
125 # Load all classes or a subset?
126 if not class_ids:
127 # All classes
128 class_ids = sorted(coco.getCatIds())
129
130 # All images or a subset?
131 if class_ids:
132 image_ids = []
133 for id in class_ids:
134 image_ids.extend(list(coco.getImgIds(catIds=[id])))
135 # Remove duplicates
136 image_ids = list(set(image_ids))
137 else:
138 # All images
139 image_ids = list(coco.imgs.keys())
```

```
140
141 # Add classes
142 for i in class_ids:
143 self.add_class("coco", i, coco.loadCats(i)[0]["name"])
144
145 # Add images
146 for i in image_ids:
147 self.add_image(
148 "coco", image_id=i,
149 path=os.path.join(image_dir, coco.imgs[i]['file_name']),
150 width=coco.imgs[i]["width"],
151 height=coco.imgs[i]["height"],
152 annotations=coco.loadAnns(coco.getAnnIds(imgIds=[i], iscrowd=False)))
153 if return_coco:
154 return coco
155
156 def load_mask(self, image_id):
157 """Load instance masks for the given image.
158
159 Different datasets use different ways to store masks. This
160 function converts the different mask format to one format
161 in the form of a bitmap [height, width, instances].
162
163 Returns:
164 masks: A bool array of shape [height, width, instance count] with
165 one mask per instance.
166 class_ids: a 1D array of class IDs of the instance masks.
167 """
168 # If not a COCO image, delegate to parent class.
169 image_info = self.image_info[image_id]
170 if image_info["source"] != "coco":
171 return super(self.__class__).load_mask(image_id)
172
173 instance_masks = []
174 class_ids = []
175 annotations = self.image_info[image_id]["annotations"]
176 # Build mask of shape [height, width, instance_count] and list
177 # of class IDs that correspond to each channel of the mask.
178 for annotation in annotations:
179 class_id = self.map_source_class_id(
180 "coco.{}".format(annotation['category_id']))
181 if class_id:
182 m = self.annToMask(annotation, image_info["height"],
183 image_info["width"])
184 # Some objects are so small that they're less than 1 pixel area
185 # and end up rounded out. Skip those objects.
186 if m.max() < 1:
```

```
187 continue
188 instance_masks.append(m)
189 class_ids.append(class_id)
190 
191 # Pack instance masks into an array
192 if class_ids:
193 mask = np.stack(instance_masks, axis=2)
194 class_ids = np.array(class_ids, dtype=np.int32)
195 return mask, class_ids
196 else:
197 # Call super class to return an empty mask
198 return super(self.__class__).load_mask(image_id)
199 
200 def image_reference(self, image_id):
201 """Return a link to the image in the COCO Website."""
202 info = self.image_info[image_id]
203 if info["source"] == "coco":
204 return "http://cocodataset.org/#explore?id={}".format(info["id"])
205 else:
206 super(self.__class__).image_reference(self, image_id)
207 
208 # The following two functions are from pycocotools with a few changes.
209 
210 def annToRLE(self, ann, height, width):
211 """
212 Convert annotation which can be polygons, uncompressed RLE to RLE.
213 :return: binary mask (numpy 2D array)
214 """
215 segm = ann['segmentation']
216 if isinstance(segm, list):
217 # polygon -- a single object might consist of multiple parts
218 # we merge all parts into one mask rle code
219 rles = maskUtils.frPyObjects(segm, height, width)
220 rle = maskUtils.merge(rles)
221 elif isinstance(segm['counts'], list):
222 # uncompressed RLE
223 rle = maskUtils.frPyObjects(segm, height, width)
224 else:
225 # rle
226 rle = ann['segmentation']
227 return rle
228 
229 def annToMask(self, ann, height, width):
230 """
231 Convert annotation which can be polygons, uncompressed RLE, or RLE to binary mask.
232 :return: binary mask (numpy 2D array)
233 """
```

```
234 rle = self.annToRLE(ann, height, width)
235 m = maskUtils.decode(rle)
236 return m
237
238
239 config=CocoConfig()
240
241 # Create model in training mode
242 model = modellib.MaskRCNN(mode="training", config=config,
243 model_dir=MODEL_DIR)
244
245
246 # Which weights to start with?
247 init_with = "last"  # imagenet, coco, or last
248
249 if init_with == "imagenet":
250 model.load_weights(model.get_imagenet_weights(), by_name=True)
251 elif init_with == "coco":
252 # Load weights trained on MS COCO, but skip layers that
253 # are different due to the different number of classes
254 # See README for instructions to download the COCO weights
255 model.load_weights(COCO_MODEL_PATH, by_name=True,
256 exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
257 "mrcnn_bbox", "mrcnn_mask"])
258 elif init_with == "last":
259 # Load weights trained
260 model_path = model.find_last()[1]
261 # Load the last model you trained and continue training
262 model.load_weights(model_path, by_name=True)
263 #assert model_path != "", "Provide path to trained weights"
264 print("Loading weights from ", model_path)
265
266
267
268 if TRAIN=="YES":
269 dataset_train = CocoDataset()
270 dataset_train.load_coco(COCO_DIR, "train")
271 dataset_train.prepare()
272
273 # Validation dataset
274 dataset_val = CocoDataset()
275 dataset_val.load_coco(VAL_DIR,"val")
276 dataset_val.prepare()
277
278
279 # This training schedule is an example. Update to fit your needs.
280 # Training - Stage 1
```

```
281 # Adjust epochs and layers as needed
282 print("Stage 1")
283 model.train(dataset_train, dataset_val,
284 learning_rate=config.LEARNING_RATE,
285 epochs=PHASE_EPOCH*1,
286 layers='heads')
287
288 # Training - Stage 2
289 # Finetune layers from ResNet stage 4 and up
290 print("Stage 2")
291 model.train(dataset_train, dataset_val,
292 learning_rate=config.LEARNING_RATE / 10,
293 epochs=PHASE_EPOCH*2,
294 layers='4+')
295 #layers='heads')
296
297 # Training - Stage 3
298 # Finetune layers from ResNet stage 3 and up
299 print("Stage 3")
300 model.train(dataset_train, dataset_val,
301 learning_rate=config.LEARNING_RATE / 100,
302 epochs=PHASE_EPOCH*3,
303 layers='all')
304 #layers='heads')
305
306 """# Training - Stage 4
307 # Adjust epochs and layers as needed
308 print("Stage 4")
309 model.train(dataset_train, dataset_val,
310 learning_rate=config.LEARNING_RATE/1000,
311 epochs=PHASE_EPOCH*4,
312 layers='heads')
313
314 # Training - Stage 5
315 # Finetune layers from ResNet stage 4 and up
316 print("Stage 5")
317 model.train(dataset_train, dataset_val,
318 learning_rate=config.LEARNING_RATE / 10000,
319 epochs=PHASE_EPOCH*5,
320 #layers='4+')
321 layers='heads') """
322
323
324
325 print("Fin del entrenamiento")
```

D.3. Procesado sobre la imagen

```
1 def display_instances_save(image, boxes, masks, class_ids, class_names,num_labels_accum,
2 scores, title="",colors=3, th_class=0.01,
3 figsize=(16, 16), ax=None):
4 """
5 boxes: [num_instance, (y1, x1, y2, x2, class_id)] in image coordinates.
6 masks: [num_instances, height, width]
7 class_ids: [num_instances]
8 class_names: list of class names of the dataset
9 scores: (optional) confidence scores for each box
10 figsize: (optional) the size of the image.
11 """
12 # Number of instances
13 N = boxes.shape[0]
14 if not N:
15 print("\n*** No instances to display *** \n")
16 else:
17 assert boxes.shape[0] == masks.shape[-1] == class_ids.shape[0]
18
19 if not ax:
20 _, ax = plt.subplots(1, figsize=figsize)
21
22 # Generate random colors
23 #colors = random_colors(81)
24
25 # Show area outside image boundaries.
26 height, width = image.shape[:2]
27 ax.set_ylim(height + 10, -10)
28 ax.set_xlim(-10, width + 10)
29 ax.axis('off')
30 ax.set_title(title)
31
32 masked_image = image.astype(np.uint32).copy()
33 #print("colores",colors)
34 #print("clases",class_ids)
35 #print("tamaño colores",len(colors))
36 #print("N=",N)
37 #print("tamaño clases",len(class_ids))
38 for i in range(N):
39 #print("clase",class_names[class_ids[i]])
40 if (scores[i]>th_class) & (class_names[class_ids[i]]!='NU'):
41 #print("i=",i)
42 #print("clases de i",class_ids[i])
43 #print("colores_i=",colors[class_ids[i]])
44 color = colors[class_ids[i]]
45
```

```
46 # Bounding box
47 if not np.any(boxes[i]):
48 # Skip this instance. Has no bbox. Likely lost in image cropping.
49 continue
50 y1, x1, y2, x2 = boxes[i]
51 p = patches.Rectangle((x1, y1), x2 - x1, y2 - y1, linewidth=2,
52 alpha=0.7, linestyle="dashed",
53 edgecolor=color, facecolor='none')
54 ax.add_patch(p)
55
56 # Label
57 class_id = class_ids[i]
58 score = scores[i] if scores is not None else None
59 label = class_names[class_id]
60 x = random.randint(x1, (x1 + x2) // 2)
61 caption = "{} {:.0f} %".format(label, score*100) if score else label
62 ax.text(x1, y1 + 80, caption,
63 color='w', size=13, backgroundcolor="none")
64
65 # Mask
66 mask = masks[:, :, i]
67 masked_image = apply_mask(masked_image, mask, color)
68
69 # Mask Polygon
70 # Pad to ensure proper polygons for masks that touch image edges.
71 padded_mask = np.zeros(
72 (mask.shape[0] + 2, mask.shape[1] + 2), dtype=np.uint8)
73 padded_mask[1:-1, 1:-1] = mask
74 contours = find_contours(padded_mask, 0.5)
75 for verts in contours:
76 # Subtract the padding and flip (y, x) to (x, y)
77 verts = np.fliplr(verts) - 1
78 p = Polygon(verts, facecolor="none", edgecolor=color)
79 ax.add_patch(p)
80 ax.imshow(masked_image.astype(np.uint8))
81 ax.text(100,500,"Detected object classes:",color='b',size=15,backgroundcolor="w")
82 j=1
83 for i in range(0,len(class_names)):
84 if (num_labels_accum[i]>0):
85 caption = "{}:                    {} ".format(class_names[i],num_labels_accum[i])
86 ax.text(100,500+j*90,caption,color='b',size=15,backgroundcolor="w")
87 j=j+1
88 plt.savefig(title,dpi=150,bbox_inches='tight',pad_inches=-0.3, bbox_extra_artists=[])
89 plt.close()
```

D.4. Preparación de vídeos de CVPR 2018 WAD

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using System.IO;
7 using System.Collections;
8 using System.Diagnostics;
9 using System.Threading;
10
11
12 namespace organizador_videos_test2
13 {
14 class Program
15 {
16 static void Main(string[] args)
17 {
18 string path = "D:/UNED/TFM/list_test_mapping";
19 string video_path = "D:/UNED/TFM/test_video_kaggle";
20 string video_path_save = "D:/UNED/TFM/videos_kaggle";
21 string[] Documents = System.IO.Directory.GetFiles(path, "*.txt");
22 int index = 0;
23
24 Console.WriteLine("Organizando imágenes");
25 Console.WriteLine();
26 Console.WriteLine("Hay {0} archivos en la carpeta", Documents.Length);
27 Console.WriteLine();
28
29 foreach (string dato in Documents)
30 {
31 ProcessFile(dato,path,video_path,video_path_save,index);
32 index++;
33 }
34
35 Thread.Sleep(3000);
36 }
37
38 public static void ProcessFile(string file, string path, string video_path, string
       video_path_save, int num_file)
39 {
40 int index = 0;
41 byte[] content = System.IO.File.ReadAllBytes(file);
42 char[] image_name = new char[32];
43 int image_index = 1;
44 string fmt = "00000";
```

```
45 string fold = "00";
46
47 Console.WriteLine("Procesando archivo {0}",file);
48
49
50 for (int i = 0; i < 32; i++)
51 image_name[i] = Convert.ToChar(content[i]);
52
53 string FileJPG = new string(image_name);
54
55 FileJPG = FileJPG + ".jpg";
56
57 index = 32;
58
59 System.IO.Directory.CreateDirectory(video_path_save + "/video_"+ num_file.ToString(fold))
       ;
60 string sourceFile = System.IO.Path.Combine(video_path, FileJPG);
61 string destFile = System.IO.Path.Combine(video_path_save+ "/video_"+ num_file.ToString(
       fold),"video_"+ num_file.ToString(fold) + "_seq_"+image_index.ToString(fmt)+".jpg");
62 System.IO.File.Copy(sourceFile, destFile, true);
63 image_index++;
64
65 try
66 {
67 while(true)
68 {
69 while (content[index]!=46)
70 {
71 index++;
72 }
73 index = index + 5;
74
75 for (int i = 0; i < 32; i++)
76 image_name[i] = Convert.ToChar(content[i+index]);
77
78 FileJPG = new string(image_name);
79
80 FileJPG = FileJPG + ".jpg";
81
82 index = 32+index;
83
84 System.IO.Directory.CreateDirectory(video_path_save + "/video_" + num_file.ToString(fold)
       );
85 sourceFile = System.IO.Path.Combine(video_path, FileJPG);
86 destFile = System.IO.Path.Combine(video_path_save + "/video_" + num_file.ToString(fold),
       "video_" + num_file.ToString(fold) + "_seq_" + image_index.ToString(fmt) + ".jpg");
87 System.IO.File.Copy(sourceFile, destFile, true);
```

```
88 image_index++;
89
90 }
91
92 }
93 catch (Exception e)
94 {
95 Console.WriteLine("Terminado el procesado del archivo {0}",file);
96 }
97
98 }
99 }
100 }
```

D.5. Procesado de video

```
import os
from visualize import random_colors
import sys
import random
import math
import re
import time
import numpy as np
import cv2
import matplotlib
import matplotlib.pyplot as plt
import scipy.misc
import sys

from config import Config
import utils
import model as modellib
import visualize
from model import log
import time
import coco

from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from pycocotools import mask as maskUtils
from coco import evaluate_coco
import postproc
from postproc import isnew
from postproc import kalman_tr
from postproc import espurios


# Root directory of the project
ROOT_DIR = os.getcwd()

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")

# Path to COCO trained weights
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")

# Carpeta de imágenes para hacer el entrenamiento
TRAIN_DIR = os.path.join(ROOT_DIR, "../../dataset/train2017")

# Carpeta de imágenes para hacer la validación del modelo
```

```
46 VAL_DIR = os.path.join(ROOT_DIR, "../../dataset")
47
48 # Carpeta de imágenes para hacer la visualización de imágenes con objetos
49 # detectados
50 VAL_DIR2 = os.path.join(ROOT_DIR, "../../dataset/val2017")
51
52 COCO_DIR = "../../dataset"
53
54 VIDEO_DIR = os.path.join(ROOT_DIR, "video_fotos")
55
56 PROCESO_DIR = os.path.join(ROOT_DIR, "fotos_procesadas")
57
58 ############################################################
59 #  Configurations
60 ############################################################
61 class CocoConfig(Config):
62 """Configuration for training on MS COCO.
63 Derives from the base Config class and overrides values specific
64 to the COCO dataset.
65 """
66 # Give the configuration a recognizable name
67 NAME = "coco"
68
69 # We use a GPU with 12GB memory, which can fit two images.
70 # Adjust down if you use a smaller GPU.
71 IMAGES_PER_GPU = 1
72
73 # Uncomment to train on 8 GPUs (default is 1)
74 GPU_COUNT = 1
75
76
77 # Number of classes (including background)
78 NUM_CLASSES = 1 + 80  # COCO has 80 classes
79
80
81 # Device to load the neural network on.
82 # Useful if you're training a model on the same
83 # machine, in which case use CPU and leave the
84 # GPU for training.
85 DEVICE = "/gpu:0"  # /cpu:0 or /gpu:0
86
87
88 ############################################################
89 #  Dataset
90 ############################################################
91 class CocoDataset(utils.Dataset):
92 def load_coco(self, dataset_dir, subset, class_ids=None,
```

```
93 class_map=None, return_coco=False):
94 """Load a subset of the COCO dataset.
95 dataset_dir: The root directory of the COCO dataset.
96 subset: What to load (train, val, minival, val35k)
97 class_ids: If provided, only loads images that have the given classes.
98 class_map: TODO: Not implemented yet. Supports maping classes from
99 different datasets to the same class ID.
100 return_coco: If True, returns the COCO object.
101 """
102 # Path
103 #He cambiado de a 2017!
104 image_dir = os.path.join(dataset_dir, "train2017" if subset == "train"
105 else "val2017")
106
107 # Create COCO object
108 #Cambiado de 2014 a 2017
109 json_path_dict = {
110 "train": "annotations/instances_train2017.json",
111 "val": "annotations/instances_val2017.json",
112 "minival": "annotations/instances_minival2017.json",
113 "val35k": "annotations/instances_valminusminival2017.json",
114 }
115 coco = COCO(os.path.join(dataset_dir, json_path_dict[subset]))
116
117 # Load all classes or a subset?
118 if not class_ids:
119 # All classes
120 class_ids = sorted(coco.getCatIds())
121
122 # All images or a subset?
123 if class_ids:
124 image_ids = []
125 for id in class_ids:
126 image_ids.extend(list(coco.getImgIds(catIds=[id])))
127 # Remove duplicates
128 image_ids = list(set(image_ids))
129 else:
130 # All images
131 image_ids = list(coco.imgs.keys())
132
133 # Add classes
134 for i in class_ids:
135 self.add_class("coco", i, coco.loadCats(i)[0]["name"])
136
137 # Add images
138 for i in image_ids:
139 self.add_image("coco", image_id=i,
```

```
140 path=os.path.join(image_dir, coco.imgs[i]['file_name']),
141 width=coco.imgs[i]["width"],
142 height=coco.imgs[i]["height"],
143 annotations=coco.loadAnns(coco.getAnnIds(imgIds=[i], iscrowd=False)))
144 if return_coco:
145 return coco
146 
147 def load_mask(self, image_id):
148 """Load instance masks for the given image.
149 
150 Different datasets use different ways to store masks. This
151 function converts the different mask format to one format
152 in the form of a bitmap [height, width, instances].
153 
154 Returns:
155 masks: A bool array of shape [height, width, instance count] with
156 one mask per instance.
157 class_ids: a 1D array of class IDs of the instance masks.
158 """
159 # If not a COCO image, delegate to parent class.
160 image_info = self.image_info[image_id]
161 if image_info["source"] != "coco":
162 return super(self.__class__).load_mask(image_id)
163 
164 instance_masks = []
165 class_ids = []
166 annotations = self.image_info[image_id]["annotations"]
167 # Build mask of shape [height, width, instance_count] and list
168 # of class IDs that correspond to each channel of the mask.
169 for annotation in annotations:
170 class_id = self.map_source_class_id("coco.{}".format(annotation['category_id']))
171 if class_id:
172 m = self.annToMask(annotation, image_info["height"],
173 image_info["width"])
174 # Some objects are so small that they're less than 1 pixel area
175 # and end up rounded out.  Skip those objects.
176 if m.max() < 1:
177 continue
178 instance_masks.append(m)
179 class_ids.append(class_id)
180 
181 # Pack instance masks into an array
182 if class_ids:
183 mask = np.stack(instance_masks, axis=2)
184 class_ids = np.array(class_ids, dtype=np.int32)
185 return mask, class_ids
186 else:
```

```
187 # Call super class to return an empty mask
188 return super(self.__class__).load_mask(image_id)
189
190 def image_reference(self, image_id):
191 """Return a link to the image in the COCO Website."""
192 info = self.image_info[image_id]
193 if info["source"] == "coco":
194 return "http://cocodataset.org/#explore?id={}".format(info["id"])
195 else:
196 super(self.__class__).image_reference(self, image_id)
197
198 # The following two functions are from pycocotools with a few changes.
199
200 def annToRLE(self, ann, height, width):
201 """
202 Convert annotation which can be polygons, uncompressed RLE to RLE.
203 :return: binary mask (numpy 2D array)
204 """
205 segm = ann['segmentation']
206 if isinstance(segm, list):
207 # polygon -- a single object might consist of multiple parts
208 # we merge all parts into one mask rle code
209 rles = maskUtils.frPyObjects(segm, height, width)
210 rle = maskUtils.merge(rles)
211 elif isinstance(segm['counts'], list):
212 # uncompressed RLE
213 rle = maskUtils.frPyObjects(segm, height, width)
214 else:
215 # rle
216 rle = ann['segmentation']
217 return rle
218
219 def annToMask(self, ann, height, width):
220 """
221 Convert annotation which can be polygons, uncompressed RLE, or RLE to binary mask.
222 :return: binary mask (numpy 2D array)
223 """
224 rle = self.annToRLE(ann, height, width)
225 m = maskUtils.decode(rle)
226 return m
227
228 # COCO Class names
229 # Index of the class in the list is its ID. For example, to get ID of
230 # the teddy bear class, use: class_names.index('teddy bear')
231 """class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
232 'bus', 'train', 'truck', 'boat', 'traffic light',
233 'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
```

```
234 'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
235 'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
236 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
237 'kite', 'baseball bat', 'baseball glove', 'skateboard',
238 'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
239 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
240 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
241 'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
242 'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
243 'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
244 'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
245 'teddy bear', 'hair drier', 'toothbrush']"""
246
247 class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
248 'bus', 'train', 'truck', 'NU', 'traffic light',
249 'NU', 'stop sign', 'NU', 'NU', 'NU',
250 'NU', 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
251 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
252 'NU', 'NU', 'NU', 'NU', 'NU',
253 'NU', 'NU', 'NU', 'NU',
254 'NU', 'NU', 'NU', 'NU', 'NU',
255 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
256 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
257 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
258 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
259 'NU', 'NU', 'NU', 'NU', 'NU',
260 'NU', 'NU', 'NU', 'NU', 'NU', 'NU',
261 'NU', 'NU', 'NU']
262
263 config = CocoConfig()
264
265 # Comprobamos cómo detecta imágenes con los pesos originales O entreados
266 # Create model object in inference mode.
267 model_inf = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR, config=config)
268
269 # Load weights trained on MS-COCO
270 #print("Loading weights ", COCO_MODEL_PATH)
271 #model_inf.load_weights(COCO_MODEL_PATH, by_name=True)
272
273
274 # Load weights trained
275 model_path = model_inf.find_last()[1]
276 #assert model_path != "", "Provide path to trained weights"
277 print("Loading weights from ", model_path)
278 model_inf.load_weights(COCO_MODEL_PATH, by_name=True)
279
280 model_inf.load_weights(model_inf.find_last()[1], by_name=True)
```

```
281
282
283
284 # Load a random image from the images folder
285 file_names = next(os.walk(VIDEO_DIR))[2]   #Crea el registro de ficheros (fotos) en la
        carpeta
286 #print("Lista de imágenes: ",file_names)
287 print("Tamaño de la lista",len(file_names))
288 # Generate random colors
289 colors = random_colors(81)
290 #print("colores =",colors)
291 detect_th=0.85
292 num_labels=[0]*len(class_names)
293 num_labels_old=[0]*len(class_names)
294 num_labels_new=[0]*len(class_names)
295 num_labels_accum=[0]*len(class_names)
296 centroide_xn=[]
297 centroide_yn=[]
298 centroide_x1=[]
299 centroide_y1=[]
300 centroide_x2=[]
301 centroide_y2=[]
302 centroide_x3=[]
303 centroide_y3=[]
304 centroide_classn=[]
305 centroide_class1=[]
306 centroide_class2=[]
307 centroide_class3=[]
308 num_labels_diff=[0]*len(class_names)
309
310 th_espur=1
311
312 images_to_process=len(file_names)
313 #images_to_process=50
314 class_names_matrix=[[0 for x in range(len(class_names))] for y in range(images_to_process
        )]
315
316 for i in range(0,images_to_process):
317 add=[0]*len(class_names)
318 trayec=[0]*len(class_names)
319 espur=[0]*len(class_names)
320 image = scipy.misc.imread(os.path.join(VIDEO_DIR, file_names[i]))
321
322 #print("image_id ", image_id, dataset.image_reference(image_id))
323
324 nombre_imagen = os.path.join(PROCESO_DIR, file_names[i])
325 print("Nombre imagen: ",nombre_imagen)
```

```
326
327 # Run detection
328 results = model_inf.detect([image], verbose=0)
329
330 # Visualize results
331 r = results[0]
332
333
334 centroide_x3=centroide_x2[:]
335 centroide_y3=centroide_y2[:]
336 centroide_class3=centroide_class2[:]
337 centroide_x2=centroide_x1[:]
338 centroide_y2=centroide_y1[:]
339 centroide_class2=centroide_class1[:]
340 centroide_x1=centroide_xn[:]
341 centroide_y1=centroide_yn[:]
342 centroide_class1=centroide_classn[:]
343
344
345 num_labels_new, centroide_xn, centroide_yn,centroide_classn =postproc.extract_info(r['
        class_ids'],r['scores'],r['masks'],r['rois'],detect_th,class_names,num_labels)
346
347 class_names_matrix[i][:]=num_labels_new
348
349 #print("Etiquetas: ", num_labels_new)
350 #print("Matriz de clases: ",class_names_matrix)
351
352 #print("centroides x",centroide_xn,centroide_x1,centroide_x2,centroide_x3)
353 #print("centroides y",centroide_yn,centroide_y1,centroide_y2,centroide_y3)
354
355 num_labels_diff=np.subtract(num_labels_new,num_labels)
356 num_labels_diff_old=np.subtract(num_labels,num_labels_old)
357
358 #print("Num labels: ", num_labels_old, num_labels, num_labels_new)
359 #print("Num labels diff: ",num_labels_diff_old, num_labels_diff)
360
361 use_prox=0
362 use_traj=1
363 use_espur=1
364
365 for j in range(0,len(class_names)):
366 add[j]=0
367 trayec[j]=0
368 espur[j]=0
369 if (num_labels_diff[j]>0):
370 if (num_labels_diff_old[j]<0):
371 add[j]=isnew(j,num_labels_diff[j],centroide_xn,centroide_yn,centroide_x2,centroide_y2,
```

```
        centroide_classn ,centroide_class2 )
372 trayec[ j]=kalman_tr(j ,num_labels_diff[ j ] ,centroide_xn ,centroide_yn ,centroide_x1 ,
        centroide_y1 ,centroide_x2 ,centroide_y2 ,centroide_x3 ,centroide_y3 ,centroide_classn ,
        centroide_class1 ,centroide_class2 ,centroide_class3 )
373 if (num_labels_diff[ j ]+use_prox*add[ j ]+use_traj*trayec[ j ]) >0:
374 num_labels_accum[ j ]=num_labels_accum[ j ]+num_labels_diff[ j ]+use_prox*add[ j ]+use_traj*
        trayec[ j ]
375 else:
376 num_labels_accum[ j ]=num_labels_accum[ j ]
377 if (i >3) & (i<images_to_process -2) & (class_names_matrix[ i -2][ j]==th_espur) & (use_espur)
        :
378 extracted_data=[row[ j ] for row in class_names_matrix]
379 espur[ j]=espurios(i ,j ,extracted_data)
380 #print("Matriz de clases: ",num_labels_new)
381 #print("Vector de espurios: ",espur)
382 num_labels_accum[ j ]=num_labels_accum[ j ]+espur[ j ]
383 
384 """print("Ajuste desaparecidos: ",add)
385 print("Ajuste trayectorias:", trayec)
386 print("lista acumulada: ",num_labels_accum)"""
387 
388 
389 
390 num_labels_old=num_labels[:]
391 num_labels=num_labels_new[:]
392 
393 
394 
395 visualize.display_instances_save(image, r['rois'], r['masks'], r['class_ids'],
        class_names, num_labels_accum, r['scores'],nombre_imagen, colors,detect_th)
```

D.6. Postrocesado: extracción de información

```
def extract_info(class_ids,scores,masks,rois,threshold,class_names,num_labels):

"""print("clases ", class_ids)
print("scores ",scores)
print("máscaras ",masks)
print("rois ",rois)"""

centroide=[[]]
centroide_x=[]
centroide_y=[]
centroide_x_orden=[]
centroide_y_orden=[]
indices_x=[]
class_filtrada=[]
class_filtrada_labels=[]
centroide_class=[]
j=1
num_labels_new=[0]*len(class_names)
#print("etiquetas: ",num_labels)

for i in range(0,len(class_ids)):
#print("esquinas:",rois[i,1])
if (scores[i]>threshold) & (class_names[class_ids[i]]!='NU'):
centroide_y.append((rois[i,0]+rois[i,2])/2)
centroide_x.append((rois[i,1]+rois[i,3])/2)
class_filtrada.append(class_ids[i])
class_filtrada_labels.append(class_names[class_ids[i]])
num_labels_new[class_ids[i]]=num_labels_new[class_ids[i]]+1

"""print("centroides x",centroide_x)
print("centroides y",centroide_y)"""

indices_x=np.argsort(centroide_x)
#print("Indices x ordenados ",indices_x)

centroide_x_orden=np.sort(centroide_x)
for i in range(0,len(centroide_y)):
centroide_y_orden.append(centroide_y[indices_x[i]])
centroide_class.append(class_filtrada[indices_x[i]])

"""print("centroides x",centroide_x_orden)
print("centroides y",centroide_y_orden)"""
"""print("class_raw: ",class_ids)"""
#print("class_filtrada: ",class_filtrada)
"""print("class_names: ",class_filtrada_labels)
```

```
46 print("lista incrementada: ",num_labels)
47 print("lista incrementada nueva: ",num_labels_new)
48
49 print("resta:",np.subtract(num_labels_new,num_labels))"""
50
51
52 return num_labels_new, centroide_x_orden, centroide_y_orden,centroide_class;
```

D.7. Postrocesado: discriminación de nuevos objetos mediante posición

```
def isnew(index,labels_diff,centroid_x,centroid_y,centroid_xold,centroid_yold,
    centroid_class,centroid_classold):

"""print("centroides x: ",centroid_x,centroid_xold)
print("centroides y: ",centroid_y,centroid_yold)
print("indice: ",index, centroid_class,centroid_classold)"""
bias=0
threshold=5
flag_dest=[0]*len(centroid_xold)


for i in range(0,len(centroid_x)):
cx_diff=[]
cy_diff=[]
sqrtdiff=[]
flag=0
for j in range(0,len(centroid_xold)):
sqrdiff=threshold+100
if (centroid_class[i]==index) & (centroid_classold[j]==index):
sqrdiff=math.pow(centroid_xold[j]-centroid_x[i],2)+math.pow(centroid_yold[j]-centroid_y[i
    ],2)
sqrdiff=math.sqrt(sqrdiff)/2
cx_diff.append(centroid_xold[j]-centroid_x[i])
cy_diff.append(centroid_yold[j]-centroid_y[i])
else:
sqrdiff=100000
cx_diff.append(10000)
cy_diff.append(10000)
sqrtdiff.append(sqrdiff)
if (sqrdiff<threshold) & (flag==0) & (flag_dest[j]!=1):
bias=bias-1
flag=1
flag_dest[j]=1
"""print("Clase:", centroid_class[i])
print("Bias",bias)
print("restas x ",cx_diff)
print("restas y ",cy_diff)
print("Diferencias ",sqrtdiff)"""

return bias
```

D.8. Postprocesado: discriminación de nuevos objetos mediante trayectoria y Filtro de Kalman

```
def kalman_tr(index,labels_diff,centroid_x,centroid_y,centroid_xold,centroid_yold,
    centroid_xold2,centroid_yold2,centroid_xold3,centroid_yold3,centroid_class,
    centroid_classold,centroid_classold2,centroid_classold3):

"""print("centroides x: ",centroid_x,centroid_xold,centroid_xold2,centroid_xold3)
print("centroides y: ",centroid_y,centroid_yold,centroid_yold2,centroid_yold3)
print("indice: ",index, centroid_class,centroid_classold,centroid_classold2,
    centroid_classold3)"""
bias=0
th_tr=50
flag_dest=[0]*len(centroid_xold)
flag_dest2=[0]*len(centroid_xold2)
flag_dest3=[0]*len(centroid_xold3)
threshold=50

"""kf = KalmanFilter(transition_matrices = [[1, 0], [0, 1]], observation_matrices = [[1,
    0], [0, 1]])
measurements = np.asarray([[10,20], [1,1], [0,1]])  # 3 observations
measurements[1]=ma.masked
kf = kf.em(measurements, n_iter=2)
(filtered_state_means, filtered_state_covariances) = kf.filter(measurements)
print("Medidas prueba: ",measurements)
(smoothed_state_means, smoothed_state_covariances) = kf.smooth(measurements)
print("KF prueba = ",filtered_state_means)
(next_state_means, next_state_convariances) = kf.filter_update(filtered_state_means[-1],
    filtered_state_covariances[-1],[0, 0])
print("Salida futura= ",next_state_means)"""

if len(centroid_x)<th_tr:
for i in range(0,len(centroid_x)):
cx_diff=[]
cy_diff=[]
nsm0=[]
nsm1=[]
sqrt=[]
sqrt2=[]
flag=0
traj_min=1000
for j in range(0,len(centroid_xold)):
for k in range(0,len(centroid_xold2)):
for l in range(0,len(centroid_xold3)):
sqr_diff=threshold+100
sqr_diff2=threshold+100
if (centroid_class[i]==index) & (centroid_classold[j]==index)& (centroid_classold2[k]==
```

```
    index) & (centroid_classold3[l]==index):
40 #la clave es comprobar la trayectoria en función de los fotogramas N-2 y N-3, asumiendo
    que el fotograma N-1 está en la recta. Si el nuevo punto es próximo al calculado,
    está alineado y no lo resto
41 missing_pos=[(2*centroid_xold2[k]-centroid_xold3[l]),(2*centroid_yold2[k]-centroid_yold3[
    l])]
42 kf =KalmanFilter(transition_matrices = [[1, 0], [0, 1]], observation_matrices = [[1, 0],
    [0, 1]], initial_state_mean=[centroid_xold3[l], centroid_yold3[l]])
43 measurements=np.asarray([[centroid_xold3[l], centroid_yold3[l]],[centroid_xold2[k],
    centroid_yold2[k]],missing_pos])
44 (filtered_state_means, filtered_state_covariances) = kf.filter(measurements)
45 (next_state_means, next_state_covariances) = kf.filter_update(filtered_state_means[-1],
    filtered_state_covariances[-1],[centroid_x[i],centroid_y[i]])
46 sqr_diff=math.pow(centroid_x[i]-next_state_means[0],2)+math.pow(centroid_y[i]-
    next_state_means[1],2)
47 sqr_diff=math.sqrt(sqr_diff)/2
48 #Si sqrt_diff es menor de un threshold, quiere decir que el móvil está en la trayectoria
    del identificado anteriormente con la estimación
49 nsm0.append(next_state_means[0])
50 nsm1.append(next_state_means[1])
51 sqrt.append(sqr_diff)
52
53 #El siguiente punto es comprobar que no podemos encontrar un punto alineado con los tres
    anteriores, esto es, que de verdad desapareció. Si se cumple que está alineado, no
    restamos, porque no hemos contado el móvil
54 kf =KalmanFilter(transition_matrices = [[1, 0], [0, 1]], observation_matrices = [[1, 0],
    [0, 1]], initial_state_mean=[centroid_xold3[l], centroid_yold3[l]])
55 measurements=np.asarray([[centroid_xold3[l], centroid_yold3[l]],[centroid_xold2[k],
    centroid_yold2[k]],[centroid_xold[j], centroid_yold[j]]])
56 (filtered_state_means, filtered_state_covariances) = kf.filter(measurements)
57 (next_state_means, next_state_covariances) = kf.filter_update(filtered_state_means[-1],
    filtered_state_covariances[-1],[centroid_x[i],centroid_y[i]])
58 sqr_diff2=math.pow(centroid_x[i]-next_state_means[0],2)+math.pow(centroid_y[i]-
    next_state_means[1],2)
59 sqr_diff2=math.sqrt(sqr_diff2)/2
60 sqrt2.append(sqr_diff2)
61 if (sqr_diff2<traj_min):
62 traj_min=sqr_diff2
63
64 if (sqr_diff<threshold) & (traj_min>threshold) & (flag==0) & (flag_dest[j]==0) & (
    flag_dest2[k]==0) & (flag_dest3[l]==0):
65 bias=bias-1
66 flag=1
67 flag_dest[j]=1
68 flag_dest2[k]=1
69 flag_dest3[l]=1
70 break
```

```
71 """print("Centroides", centroid_x[i],centroid_y[i])
72 print("Centroides medidos ",centroid_x,centroid_y)
73 print("Centroides estimados ",nsm0,nsm1)
74 print("Diferencias ",sqrt,sqrt2)
75 print("Flags ",flag_dest,flag_dest2,flag_dest3,bias)"""
76
77
78 return bias
```

D.9. Postprocesado: eliminación de objetos espurios

```
1 def espurios(i,j,matrix):
2 array_subs=0
3 #print("Columna entradas: ",matrix)
4
5 th=matrix[i-2]
6
7 if (matrix[i-4]<th) & (matrix[i-3]<th) & (matrix[i-1]<th) & (matrix[i]<th):
8 array_subs=-1
9 else:
10 array_subs=0
11
12
13 return array_subs
```

Bibliografía

[1] *AI winter*

[2] Bernard Marr, *A Short History of Machine Learning Every Manager Should Read*

[3] Sancho Caparrini F., *Introducción al Aprendizaje Automático*

[4] *Aprendizaje Supervisado y no Supervisado*

[5] *Aprendizaje no Supervisado*

[6] R.A. Johnson, D.V. Wichern,*Applied Multivariate Statistical Analysis 6th Ed*

[7] *Redes Neuronales*

[8] MacQueen, J. B. *Some Methods for Classification and Analysis of Multivariate Observations* Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, 1, Berkeley, CA: University of California Press (1967), 281-297.

[9] Myerson, R. B. *Game Theory. Analisis of Conflict* Harvard University Press, First Edition 1997.

[10] Morris, P. *Introduction to Game Theory* Springer, Second edition 1994.

[11] Piatetsky-Shapiro, G., *Discovery, analysis, and presentation of strong rules*, in G. Piatetsky-Shapiro and W. J. Frawley, eds, Knowledge Discovery in Databases, AAAI/MIT Press, Cambridge, MA. 1991.

[12] Agrawal R.; Imielinski T.; A. Swami *Mining Association Rules Between Sets of Items in Large Databases*, SIGMOD Conference 1993: 207-216

[13] Hinojosa Gutiérrez A. P. et alii *Introducción al lenguaje de programación Python (3º Edición)*, Universidad de Granada 2011.

[14] *Anaconda, distribución de Python*

[15] *Libro Machine Learning PDF sobre TensorFlow*

[16] *Procesamiento Paralelo Cuda*

[17] Torres, J. *Primeros pasos en Keras*

[18] Castellanos A. *¿Qué son las redes neuronales?*.

[19] Ballesteros A. *Historia de las Redes Neuronales*

[20] Blasco J. *Verificación y entrenamiento de un predictor no lineal de vídeo mediante entorno Matlab-System Generator-Placa FPGA/PCI* URSI 2005

[21] Blasco J., Gadea R., Aliaga R. *Verificación y entrenamiento de redes neuronales mediante entorno Matlab-System Generator-Placa FPGA/PCI* JCRA 2005

[22] Trentin E. *Multilayer Perceptron (MLP): the Backpropagation (BP) Algorithm* Università di Siena, 2008

[23] Syrine Ben Driss, M Soua, Rostom Kachouri, Mohamed Akil. *A comparison study between MLP and Convolutional Neural Network models for character recognition* SPIE Conference on Real-Time Image and Video Processing, Apr 2017, Anaheim, CA, United States. SPIE Proceedings 10223, Real-Time Image and Video Processing 2017.

[24] *Redes Neuronales Convolucionales*

[25] Fukushima, K.*Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position.* Biological Cybernetics 36 (4): 193?202 1980.

[26] Nielsen M. *Neural Networks and Deep Learning* Free online book

[27] Deshpande A. *A Beginner's Guide To Understanding Convolutional Neural Networks*

[28] Doukkali F. *Convolutional Neural Networks (CNN, or ConvNets)*

[29] Geitgey A. *Machine Learning is Fun! Part 3: Deep Learning and Convolutional Neural Networks*

[30] Hinton G. *Rectified Linear Units Improve Restricted Boltzmann Machines* Department of Computer Science, University of Toronto.

[31] Hinton G. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* Department of Computer Science, University of Toronto.

[32] Lin M. *Network In Network* Marzo 2014.

[33] Deshpande A. *The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3)*

[34] Krizhevsky A. *ImageNet Classification with Deep Convolutional Neural Networks*

[35] Zeiler M., Fergus R. *Visualizing and Understanding Convolutional Networks Noviembre 2013*

[36] Simonyan K., Zisserman A. *VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION Abril 2015*

[37] Szegedy C. et ali *Going Deeper with Convolutions CVPR2015*

[38] Kaiming H. et ali *Deep Residual Learning for Image Recognition Diciembre 2015*

[39] Girshick R. et ali *Rich feature hierarchies for accurate object detection and semantic segmentation Octubre 2014*

[40] Hui J. *What do we learn from region based object detectors (Faster R-CNN, R-FCN, FPN)?*

[41] Gao H. *Faster R-CNN Explained*

[42] Kaiming H. et ali *Mask R-CNN arXiv:1703.06870, Enero 2018*

[43] Goodfellow I. et ali *Generative Adversarial Nets Junio 2014*

[44] Karpathy A., Fei-Fei L. *Deep Visual-Semantic Alignments for Generating Image Descriptions Abril 2015*

[45] Jaderberg M. et ali *Spatial Transformer Networks Febrero 2016*

[46] *CS231n Convolutional Neural Networks for Visual Recognition*

[47] Girshick R. *Facebook open sources Detectron Enero 2018*

[48] Lin T. et ali *Microsoft COCO: Common Objects in Context Febrero 2015*

[49] Rosebrock A. *Intersection over Union (IoU) for object detection*

[50] Everingham M. et ali *The PASCAL Visual Object Classes (VOC) Challenge International Journal of Computer Vision*

[51] Majek K. *Mask RCNN*

[52] *Labels in COCO-Stuff*

[53] *Proceso de Poisson*

[54] Welch G., Bishop G. *An Introduction to the Kalman Filter. University of North Carolina at Chapel Hill, 2001*

[55] El Filtro de Kalman. Mplanaslasa, 2014

[56] Esme B. *Kalman Filter For Dummies*

[57] Thacker N.A., Lacey A.J. *Tutorial: The Kalman Filter. Imaging Science and Biomedical Engineering Division. University of Manchester. Diciembre 1998*

[58] Vélez R., García A., *Principios de inferencia estadística, Unidad Didáctica UNED, enero 2009, pp. 142-143*

[59] *How to understand Kalman gain intuitively?*

[60] Colah C.,*Understanding Convolutions*