ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Trabajo Fin de Grado

Estudio práctico de técnicas AI para reconocimiento de objetos

Autor: Andrea Martínez Martínez

Director: María Dolores Cano Baños

Agradecimientos

A mi familia y amigos, por ser la base de quien soy hoy.

A Lola, por abordar conmigo este proyecto.

Índice de contenidos

Relación de figuras

Capítulo 1. Introducción y Objetivos

1.1 Introducción

Este proyecto está basado en el estudio de técnicas de inteligencia artificial en el ámbito de la visión artificial, que trata de conseguir sistemas capaces de comprender una imagen o secuencia de imágenes y actuar según convenga en una determinada situación.

Las principales bases de este proyecto son el aprendizaje automático y el aprendizaje profundo (más conocidos por su nombre en inglés: Machine Learning y Deep Learning) y su aplicación en el mundo de la inteligencia artificial. Antes de pasar a los objetivos del proyecto necesitamos saber un poco más sobre estos conceptos para poder identificarlos y diferenciarlos a lo largo de esta memoria.

1.2 Conceptos clave

- Inteligencia Artificial:

  Subdisciplina del campo de la informática, que busca la creación de máquinas que puedan imitar comportamientos inteligentes. Esta disciplina trata de crear sistemas que puedan ser capaces de "aprender" o "razonar" como un ser humano. En nuestro ámbito de estudio una definición más correcta de inteligencia artificial sería: estudio, desarrollo y aplicación de técnicas informáticas que permiten a cierto sistema adquirir habilidades propias de la inteligencia humana, como, por ejemplo:

  - Entender situaciones y contextos.
  - Identificar objetos y reconocer su significado.
  - Analizar y resolver problemas.
  - Aprender a realizar nuevas tareas.
  - Comprender el lenguaje natural.
  - Reconocer imágenes.

- Machine Learning:

  Machine learning, o aprendizaje automático, es el conjunto de algoritmos informáticos que mejoran automáticamente a través de la experiencia. Los algoritmos de Machine Learning construyen un modelo matemático basado en datos de muestra conocidos como "datos de entrenamiento" con la finalidad de hacer predicciones o decisiones sobre estos datos sin estar específicamente programados para ello, es decir, en lugar de escribir un código para una tarea en concreto, proporcionaremos al algoritmo los datos de entrenamiento, y éste construirá su propia lógica basada en ellos.

- Deep Learning:

  Deep Learning, o aprendizaje profundo, es un subconjunto de algoritmos de Machine Learning que construye redes neuronales artificiales para imitar la estructura y función del cerebro humano. En la práctica, utiliza un gran número de capas ocultas de procesamiento no lineal para extraer características de los datos, cada una de las capas se corresponde con un nivel más alto de abstracción que las capas previas.

A modo de resumen aclaratorio diremos que el conjunto de algoritmos de Machine Learning son las técnicas que usamos para dotar a máquinas de inteligencia artificial, y los algoritmos de Deep Learning son un subconjunto de algoritmos de Machine Learning que toman como principio el funcionamiento de las redes neuronales.

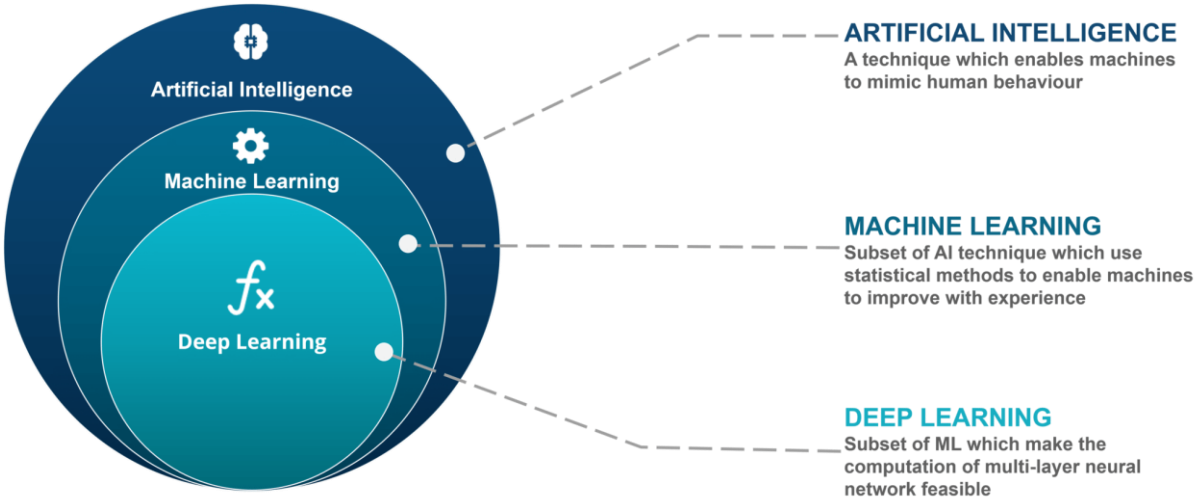


Figura 1. Esquema de niveles AI, ML, DL

1.3 Justificación del proyecto: el auge de la AI

Para justificar la elección de este proyecto es necesario hablar sobre el auge que está teniendo la inteligencia artificial en los últimos años. Según el periódico digital ComputerWorld.es, Arvind Krishna, nombrado recientemente CEO de IBM, ha reafirmado su creencia de que "la nube híbrida y la AI son las dos fuerzas dominantes que impulsan la transformación digital hoy en día".

La inteligencia artificial surge como concepto en los años 50. Tras las teorías que ya anticipaba Alan Turing en su artículo "Computing Machinery and Intelligence" la AI nacía como concepto en 1956 gracias a John McCarthy, Marvin Minsky y Claude Shannon, que la definieron como "la ciencia e ingenio de hacer máquinas inteligentes". Pero el gran avance en este campo no se ha producido hasta hace unos años atrás, debido a que la inteligencia artificial requiere grandes conjuntos de datos y gran capacidad de cómputo. Afortunadamente, la creación y disponibilidad de datos ha crecido exponencialmente, esto, junto con las mejoras en hardware de almacenamiento, ha permitido disponer de enormes conjuntos de datos para entrenar modelos de inteligencia artificial.

El mundo de la inteligencia artificial está creciendo muchísimo y permite avances en gran variedad de campos, algunos de los más usados hoy en día son:

- AI incorporada al software de fotografía de los smartphones para ayudar en fotografías con malas condiciones de luz, enfoque de rostros, detección y ajuste de escenas, etc.

- Personaje no jugador en videojuegos, que toman patrones de jugadores reales para que el juego sea más realista en comparación al uso de bots para modos de juego offline.

- Vehículos autónomos que son capaces de aprender por donde circular o como analizar las señales y objetos que encuentran a su paso.

- Traductores tanto de texto como de voz, utilizan AI para aprender y recoger cada vez más expresiones que permitan mayor precisión en las traducciones con el paso del tiempo.

- Reconocimiento facial a tiempo real, usado para el desbloqueo facial en smartphones.

Otros usos más curiosos y novedosos son:

- Uso de cámaras de video con AI para comprobar el uso de mascarillas y el cumplimiento de la distancia de seguridad reglamentaria.

- Reconstrucción de imágenes de rostros pixelados, con aplicación en el mundo policial.

- Herramientas que permiten el diagnóstico de enfermedades.

- Extracción de información clínica útil a partir de imágenes médicas que permitirán conocer más acerca de las enfermedades.

En definitiva, la AI tiene innumerables usos que con el tiempo influirán en casi todas las facetas de nuestras vidas para ayudar a mejorar la eficiencia y aumentar nuestras capacidades humanas.

1.4 Objetivos

Desde un punto de vista generalizado el objetivo principal del proyecto es iniciarse en las bases de la ciencia de datos y la inteligencia artificial mediante la exploración de los principales algoritmos de aprendizaje automático y aprendizaje profundo.

En cuanto a los objetivos concretos, tenemos dos enfoques principales:

- El desarrollo de un modelo que permita el seguimiento o trazabilidad de personas mediante vídeo.

  Este modelo se encargará de procesar la entrada de vídeo detectando en cada fotograma las personas que aparecen y pudiendo realizar un seguimiento de estas, siendo capaz de reconocer que, por ejemplo, si aparece una persona andando en fotogramas sucesivos, se trata en realidad de la misma persona. Además, registrar a las personas detectadas con un identificador y ubicarlas con marcas temporales y espaciales para poder realizar un seguimiento.

- Crear un sistema de identificación de estados de ánimo en caras:

  Entre las funcionalidades principales estarían conseguir ubicar en la imagen el rostro de la persona y una vez ubicado conseguir examinar las características de este para predecir un estado de ánimo. Para conseguir esto la red se entrenará con imágenes de personas de piel oscura. La decisión de usar como datos de entrenamiento a personas con piel oscura se debe a que los algoritmos de reconocimiento facial están sufriendo graves dificultades a la hora de ser imparciales con los sesgos, esto hace que incluso los sistemas de reconocimiento facial de alto rendimiento identifiquen erróneamente a personas de piel oscura con tasas muy elevadas [39].

Además de los objetivos mencionados cabe destacar algunos otros como son:

- Aprendizaje del lenguaje Python y el uso de bibliotecas como Tensorflow y Keras para el desarrollo de algoritmos.

- Iniciación al uso de repositorios para la búsqueda de bibliotecas y otros frameworks.

- Uso de servicios cloud que proporcionan grandes empresas como Google y Amazon para el desarrollo de aplicaciones con inteligencia artificial y otros servicios como el uso gratuito de GPU.

Capítulo 2. Técnicas de detección de objetos y reconocimiento facial

2.1 Introducción a algoritmos Machine Learning

Dentro de los algoritmos de machine learning hay tres tipos fundamentales:

- Aprendizaje supervisado:

  Se basa en la generación de conocimiento a través del análisis de datos etiquetados. Como datos de entrenamiento usaremos un conjunto de ejemplos con resultados conocidos, el modelo aprenderá los parámetros de la muestra para después poder usarlos para obtener resultados acertados de nuevos datos. El aprendizaje supervisado emplea técnicas de clasificación y regresión para desarrollar modelos predictivos.

  Las técnicas de clasificación predicen respuestas discretas; por ejemplo, si un correo electrónico es legítimo o es spam, o bien si un tumor es cancerígeno o benigno. Los modelos de clasificación organizan los datos de entrada en categorías. Algunos algoritmos habituales para realizar la clasificación son: máquina de vectores de soporte (SVM), k-vecino más cercano, clasificadores bayesianos, regresión logística y redes neuronales.

  Las técnicas de regresión predicen respuestas continuas; por ejemplo, cambios de temperatura o fluctuaciones en la demanda energética. Las aplicaciones más habituales son la predicción de la carga eléctrica y el trading algorítmico. Algunos algoritmos habituales de regresión son: modelo lineal y no lineal, regularización, regresión por pasos, redes neuronales y aprendizaje neurodifuso adaptativo.

- Aprendizaje no supervisado:

  En este caso los datos de ejemplo no están etiquetados, sino que el algoritmo buscará patrones ocultos o estructuras intrínsecas en los datos para crear agrupaciones.

  El clustering es la técnica de aprendizaje no supervisado más común. Se emplea para el análisis de datos exploratorio para encontrar patrones o agrupaciones ocultos en los datos. Entre sus aplicaciones están el análisis de secuencias genéticas, la investigación de mercados y el reconocimiento de objetos. Algunos algoritmos habituales de clustering son: k-means y k-medoids, clustering jerárquico, modelos de mezclas gaussianas, modelos de Markov ocultos, mapas autoorganizados, c-means y clustering sustractivo.

- Aprendizaje por refuerzo:

  Este caso funciona a través de ensayo y error en un algoritmo que se retroalimenta. El algoritmo recibe como entrada un dato de primeras desconocido y dará un resultado, si el resultado es correcto recibirá un premio, si no, una sanción. De esta manera el algoritmo podrá percibir que características de cada uno de los datos se corresponden con las soluciones premiadas para así mejorar sus predicciones.

En este proyecto nos centraremos en el aprendizaje supervisado. Trataremos con los dos casos prácticos mencionados anteriormente:

- Detección y seguimiento de personas.

- Identificación de rostros y estados de ánimo.

En los próximos apartados se describen los principales algoritmos de detección de objetos, que es la base para los dos casos, y como se utilizan estos algoritmos en conjunto con otras técnicas para conseguir propósitos más concretos.

2.2 Técnicas de detección de objetos

- Region-based Convolutional Neural Networks (R-CNN) [12]

  Antes de explicar cómo funciona el R-CNN hay que explicar qué es una CNN (red neuronal convolucional). Las CNN son un tipo de algoritmos de Deep learning usados principalmente para la clasificación de imágenes, que generalmente se componen de las siguientes capas:

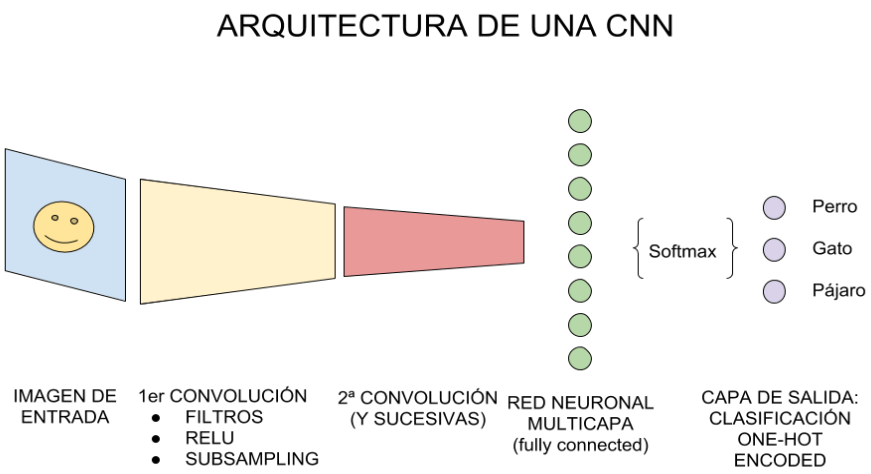


Figura 2. CNN

- Capa de convolución (CONV): va escaneando la imagen de entrada y utiliza filtros (conjunto de kernels) que realizan operaciones de convolución. Tras ello se utilizan funciones de activación, la más usada es ReLu (Rectified Linear Unit) que elimina los valores negativos. La salida resultante se llama mapa de características o de activación.

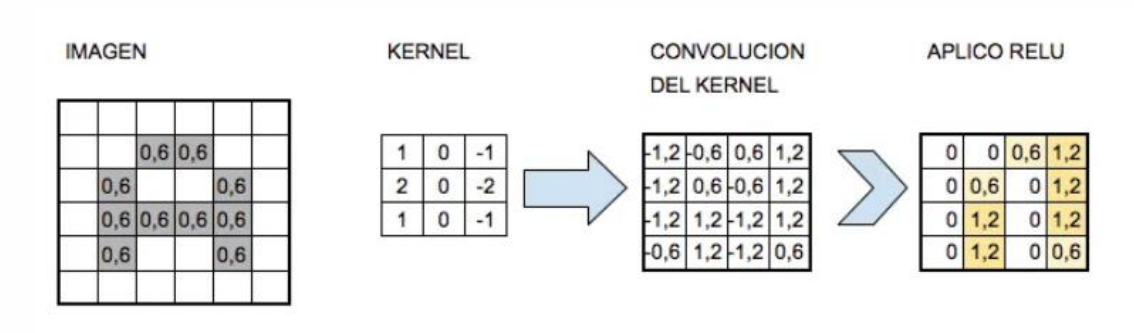


Figura 3. CNN-CONV layer

- Capa de pooling (POOL): es una operación de submuestreo que se suele aplicar tras la convolución, se encarga de reducir las dimensiones que recibirá la próxima capa, hay dos tipos principales:

Max pooling: Cada operación de agrupación selecciona el valor máximo de la vista actual (más común que average pooling).

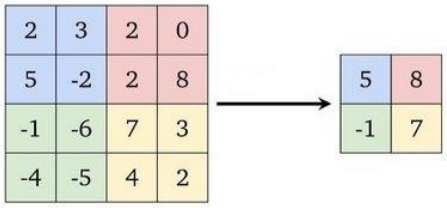


Figura 4. CNN-Max Pooling

Average pooling: Cada operación de agrupación selecciona el valor medio de los valores de la vista actual.

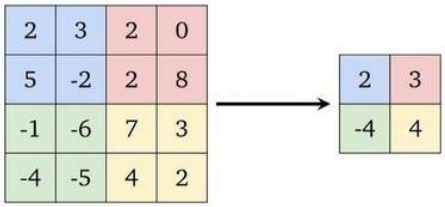


Figura 5. CNN-Average Pooling

- Capa fully connected (FC): Cada entrada está conectada a todas las neuronas y tendrá el mismo número de neuronas que de clases que pueda predecir, estas capas suelen encontrarse al final de las arquitecturas CNN.

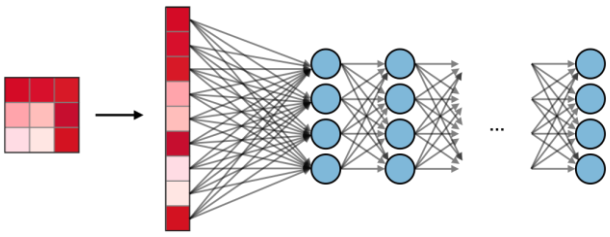


Figura 6. CNN-FC layer

El problema de las CNN es que permiten clasificar la imagen, pero no permiten ubicar diferentes objetos en una imagen, como solución para este problema surgen las R-CNN.

Una red R-CNN se compone de tres módulos principales. El módulo inicial se encarga de extraer unas 2000 propuestas de región en la imagen de entrada a partir de un algoritmo de segmentación llamado búsqueda selectiva, este algoritmo escanea la imagen con ventanas de varios tamaños, y busca píxeles adyacentes que compartan colores y texturas, además tiene en cuenta las condiciones de la luz.

El segundo módulo es una CNN que extrae un vector de características por cada propuesta de región. La región propuesta sufrirá deformaciones para adaptarse al módulo final, un clasificador (Support Vector Machines, SVM), que clasifica cada región.

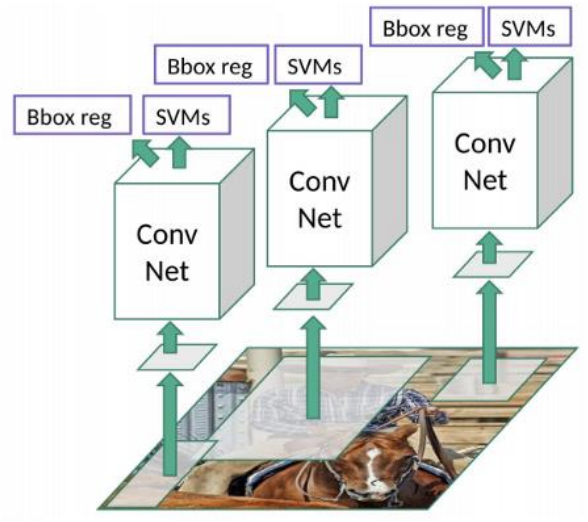


Figura 7. R-CNN

- Fast R-CNN [13]

  Fue propuesto por uno de los autores de R-CNN. Esta red también usa un algoritmo para generar propuestas de región. La diferencia es que Fast R-CNN calcula un mapa de características convolucional para toda la imagen de entrada en lugar de para cada una de las regiones, lo que lo hace mucho más rápido.
  A continuación, para cada propuesta de objeto, se extrae un vector de características de longitud fija del mapa de características utilizando una capa de agrupación de región de interés (Region of Interest, RoI). Fast R-CNN mapea cada uno de estos RoI en un vector de características utilizando capas fully connected, para finalmente generar la probabilidad softmax y el cuadro delimitador, que son la clase y la posición del objeto, respectivamente.

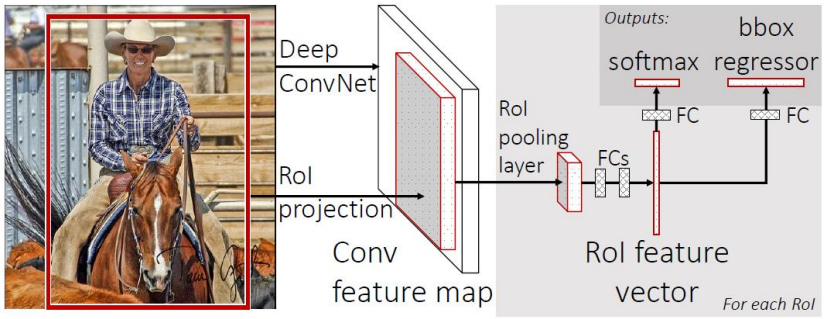


Figura 8. Fast R-CNN

- Faster R-CNN [14]

  La búsqueda selectiva es un proceso lento que afecta al rendimiento de la red, por tanto, Fater R-CNN abandona el método tradicional de propuestas de región y apuesta por un enfoque de aprendizaje completamente profundo. Consta de dos módulos una CNN llamada red de propuesta de región (RPN) y el detector de Fast R-CNN. Los dos módulos se fusionan en una sola red.

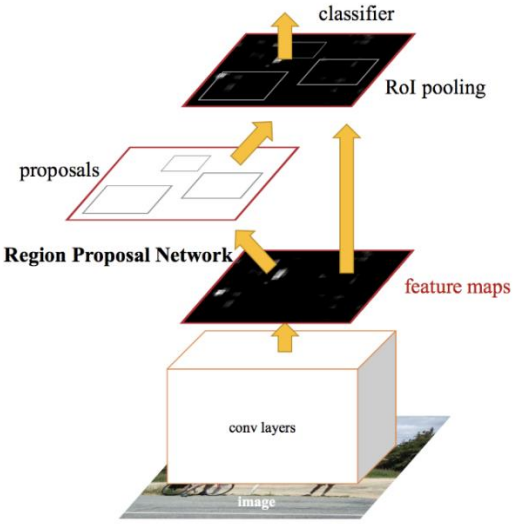


Figura 9. Faster R-CNN

- Histogram of Oriented Gradients (HOG) [4]

  Un histograma de gradientes orientados es un descriptor de características que se utiliza en técnicas de visión por computador para el procesamiento de imágenes con el fin de detectar objetos. La idea es reducir la complejidad de la imagen y mantener solamente las características más descriptivas. Empezamos por pasar la imagen a blanco y negro ya que podemos detectar la ubicación de un objeto sin necesidad de color. Nuestro objetivo ahora es comparar cada píxel con sus pixeles contiguos y sustituirlos por flechas que indican la dirección en la que la imagen se hace más oscura, obteniendo así los gradientes de la imagen. Pero hacerlo píxel a píxel sigue siendo demasiada información, por tanto, se hace en cuadrículas de 16x16 píxeles. En cada cuadrícula, contaremos cuántos gradientes apuntan en cada dirección principal, luego reemplazaremos esa cuadrícula en la imagen con la dirección con más ocurrencias.

  Este método se usa en detección de objetos para encontrar determinados objetos creando el HOG de la imagen y buscando sobre el mismo, patrones de HOG conocidos.

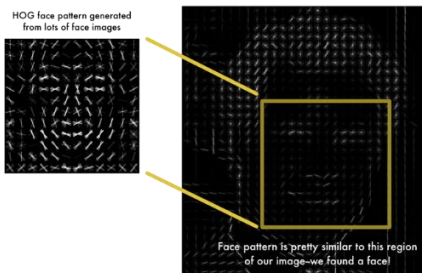


Figura 10. Comparación de patrón HOG facial en esquema HOG de una imagen

- Spatial Pyramid Pooling (SPP-net) [15]

  Se trata de una estructura de red que puede generar una representación de longitud fija independientemente del tamaño de la imagen. La combinación de pirámides resiste a las deformaciones de los objetos, y SPP-net mejora a los métodos basados en CNN, ya que permite calcular los mapas de características de la imagen completa solo una vez y luego agrupar las características en regiones arbitrarias (subimágenes) para generar representaciones de longitud fija para entrenar a los detectores, así evita calcular repetidamente las características convolucionales.

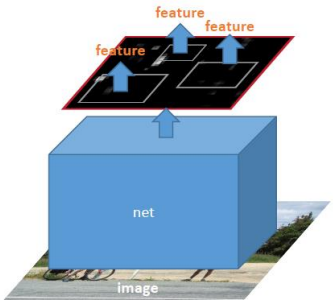


Figura 11. SPP-Net

- YOLO (You Only Look Once) [16]

  Este es uno de los algoritmos más populares para detección de objetos. Este algoritmo utiliza un enfoque totalmente diferente a los anteriores, esta red divide la imagen en regiones y predice cuadros delimitadores y probabilidades para cada región. La mayor diferencia ante otros algoritmos y lo que lo hace mucho más eficiente es que puede realizar predicciones con una sola evaluación de la red.

  Su funcionamiento es el siguiente, dividimos la imagen en una cuadrícula de dimensiones SxS, dentro de cada cuadrícula seleccionamos B cuadros delimitadores para cada uno de los cuales, la red genera una probabilidad de clase y valores de compensación para el cuadro delimitador. De entre los cuadros delimitadores establecidos se buscan aquellos cuya probabilidad de clase esté por encima de un valor umbral, estos son seleccionados y utilizados para localizar el objeto dentro de la imagen.

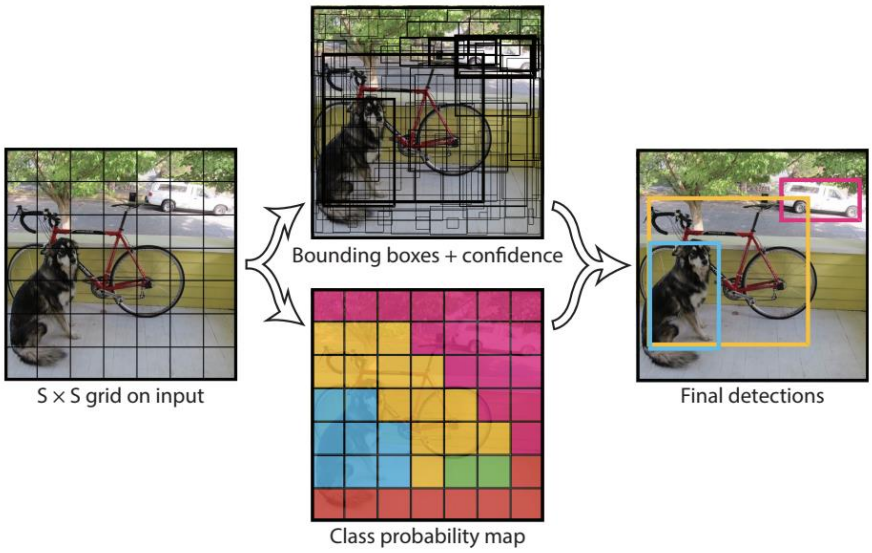


Figura 12. YOLO

- Single Shot Multibox Detector (SDD) [17]

  Este algoritmo se presentó un par de meses después de YOLO como una alternativa digna. De manera similar a YOLO, la detección de objetos se realiza en una única propagación hacia adelante de la red. Este modelo de CNN de extremo a extremo pasa la imagen de entrada a través de una serie de capas convolucionales, generando cuadros delimitadores candidatos de diferentes escalas en el camino. SSD considera los objetos etiquetados como ejemplos positivos, y cualquier otro cuadro delimitador que no se superponga con los positivos son ejemplos negativos. Resulta que construir el conjunto de datos de esta manera lo hace muy desequilibrado. Por esa razón, SSD aplica un método llamado minería negativa dura, que se basa en elegir solamente los

ejemplos negativos con la mayor pérdida de confianza, esto conduce a una optimización más rápida y una fase de entrenamiento más estable.

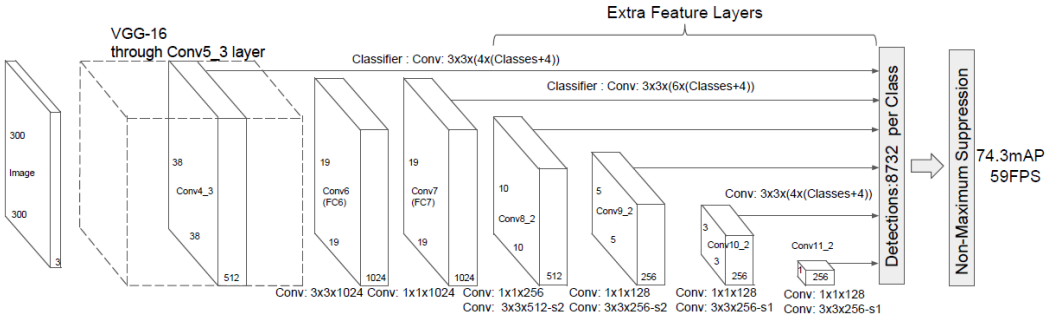


Figura 13. SSD

2.3 Aplicación a la trazabilidad de objetos

El seguimiento o trazabilidad de objetos es una disciplina dentro de la visión por computador, cuyo objetivo es rastrear objetos a medida que se mueven a través de una serie de cuadros de video. Los objetos suelen ser personas, pero también pueden ser animales, vehículos u otros objetos de interés, como la pelota en un partido de fútbol.

El seguimiento de objetos comienza con la detección de objetos: identificar objetos en una imagen y asignarles cuadros delimitadores. El algoritmo de seguimiento de objetos asigna un ID a cada objeto identificado en la imagen, y en los cuadros subsiguientes intenta transmitir este ID e identificar la nueva posición del mismo objeto.

Estos son algunos de los principales algoritmos de trazabilidad de objetos:

- SORT [20]

  SORT es un algoritmo de seguimiento de objetos que se basa principalmente en el análisis de un motor de detección de objetos subyacente. Puede conectarse a cualquier algoritmo de detección de objetos. El algoritmo rastrea múltiples objetos en tiempo real, asociando los objetos en cada fotograma con los detectados en fotogramas anteriores usando heurísticas simples. Por ejemplo, SORT maximiza la métrica IOU (Intersection Over Union) entre cuadros delimitadores en fotogramas vecinos.

- Deep SORT [21]

  Toma como base SORT, pero reemplaza la métrica de asociación con un nuevo aprendizaje de métricas de coseno, un método para aprender un espacio de características donde la similitud del coseno se optimiza de manera efectiva a través de la reparametrización del régimen softmax.

  En Deep SORT los cuadros delimitadores se calculan utilizando una red neuronal convolucional previamente entrenada, entrenada en un conjunto de datos de reidentificación de personas a gran escala. Este método tiene muchas ventajas en la detección de múltiples objetos, ya que es simple de implementar, ofrece una precisión sólida y se ejecuta en tiempo real.

- Track R-CNN [22]

  Para la parte de detección se utiliza un modelo Fast R-CNN para inicializar una trayectoria. Cuando un objeto es detectado, se agrega una nueva trayectoria a la lista de trayectorias con un estado inicial oculto calculado a partir del cuadro delimitador detectado.

  La parte de seguimiento se compone de una parte de movimiento previo y una red de comparación de apariencia. La parte de movimiento previo hace propuestas de región en el fotograma actual de una trayectoria dados los resultados de búsqueda selectiva y el historial de seguimiento previo. Mientras que la red de comparación de apariencia genera una puntuación de seguimiento dada la propuesta de región y el historial de seguimiento. Elegiremos el cuadro delimitador que corresponde a la puntuación de seguimiento más grande en cada fotograma.

- Tracktor ++ [23]

  La idea principal de Tracktor ++ es utilizar la rama de regresión de Faster R-CNN para el seguimiento fotograma a fotograma extrayendo características del fotograma actual y luego utilizando las ubicaciones de los objetos del fotograma anterior como entrada para que el proceso de agrupación de RoI pueda reubicarlos en el fotograma actual.

  Este algoritmo también utiliza algunos modelos de movimiento tales como la compensación de movimiento de cámara basado en el registro de imágenes, y reidentificación a corto plazo. El método de reidentificación almacena en caché los registros para un número fijo de fotogramas y luego compara los recién detectados con lo almacenado en caché para una posible reidentificación.

2.4 Aplicación al reconocimiento facial y la detección de emociones

El reconocimiento facial es una tarea que se divide en varios problemas sucesivos:

1. Dada una imagen conseguir detectar los rostros en ella.

2. Centrarnos en un rostro de los detectados e intentar que la inclinación o rotación del mismo y las condiciones de luz afecten lo mínimo posible.

3. Hallar las características que distinguen unívocamente dicho rostro de cualquier otro.

4. Finalmente utilizaríamos las características halladas para poder diferenciar ese rostro de otros cuyas características conocemos.

Estos problemas se resuelven de la siguiente forma. En primer lugar, haciendo uso del ya mencionado histograma de gradientes orientados. Una vez obtenida la imagen de entrada aplicamos HOG para obtener los gradientes de la imagen y poder buscar las caras usando un patrón de HOG conocido para ello.

Una vez ubicados los rostros nos centraríamos solamente en el recuadro que delimita el rostro detectado. Es muy probable que la cara de dicha persona no se encuentre en posición frontal, y que cada imagen que recibamos sea en una postura diferente, por lo que necesitaremos reajustar esas imágenes en función de un patrón para poder ser lo más precisos posible en los posteriores pasos. Para este cometido se utiliza un algoritmo que es capaz de ubicar los puntos de referencia de un rostro, algo similar a los mostrados en la siguiente imagen.

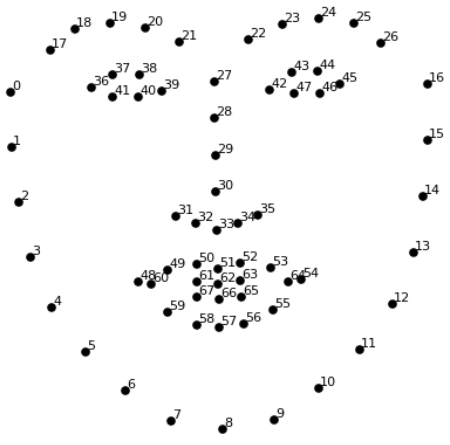


Figura 14. Puntos de referencia faciales

Tras ubicar los puntos se aplicarán transformaciones afines (como rotaciones y escalas) para conseguir reajustar la posición de esos puntos obteniendo así una imagen transformada que pretende centrar al máximo el rostro.

El penúltimo paso es calcular las características que nos permitan diferenciar a una persona de otra, para nosotros como personas es lógico pensar en medidas biométricas,

ya que visualmente somos capaces de diferenciarlo, pero para un computador una imagen no es más que un conjunto de bits, no posee el suficiente nivel de abstracción como para "aprender a ver". Este problema se ha solucionado de una forma muy sencilla y compleja a la vez, dejando a la propia máquina "decidir" qué medidas son cruciales para el reconocimiento facial. Se trata de entrenar un modelo con tres imágenes, dos de ellas de la misma persona conocida, y otra de una persona totalmente diferente, la red generará unas medidas (en sus comienzos aleatorias y desacertadas), tras ello deberá comparar las medidas generadas para las tres imágenes y ajustar la red para asegurarse de que las medidas de las imágenes que pertenecían a la misma persona están un poco más cerca, y estas dos a su vez más alejadas de la imagen de la persona desconocida. Tras millones de pasos de entrenamiento y millones de imágenes obtendremos una red capaz de generar medidas confiables que identifiquen a una persona.

Por último, entrenar un clasificador que tome las medidas generadas por el modelo anterior y sea capaz de compararlas con las medidas de rostros conocidos para realizar una predicción. Una forma de comparación sencilla sería calcular la distancia euclídea para cada una de las medidas y comprobar si dichas distancias se encuentran por debajo de un valor umbral.

Pasemos ahora a hablar de la detección de emociones en rostros, el proceso más lógico a seguir es bastante similar al proceso seguido para el reconocimiento facial. El primer paso sería detectar el rostro en la imagen como ya hemos explicado anteriormente, seguiríamos también el mismo proceso para centrar y alinear la cara.

Una vez hecho esto habría que entrenar una red capaz de abstraer las características comunes a rostros que comparten la misma emoción, para ello utilizaremos una red neuronal y la entrenaremos con conjuntos de imágenes de rostros correspondientes a cada emoción diferente que se quiera identificar.

Capítulo 3. Caso práctico: Trazabilidad de objetos

En este capítulo hablaremos sobre las técnicas utilizadas para realizar una aplicación capaz de detectar, indexar y seguir personas en vídeos o en imagen a tiempo real.

3.1 Algoritmos escogidos

Teniendo en cuenta las bases analizadas en el capítulo anterior debemos escoger dos algoritmos, un algoritmo encargado de realizar la detección de objetos y otro algoritmo capaz de hacer un seguimiento, para después complementarlos en una única aplicación, de modo que la salida del algoritmo de detección se utiliza como entrada para el algoritmo encargado de la trazabilidad.

En la trazabilidad de objetos orientada a personas es interesante que los algoritmos utilizados sean capaces de ejecutarse a tiempo real para así poder ser usado en cámaras en vivo, los algoritmos serán elegidos en función de este criterio, la velocidad de procesamiento en fotogramas por segundo.

Analizando comparativas de precisión y velocidad de algoritmos de detección de objetos la conclusión obtenida es que el algoritmo más adecuado para este propósito es YOLO debido a que, aunque su precisión en las detecciones sea menor que la proporcionada por otros algoritmos como Faster R-CNN, su rapidez es mucho mayor lo que nos permite realizar detecciones en tiempo real. En la Figura 15 se muestra un gráfico de barras que permite comparar diversos algoritmos de detección de objetos en función de su precisión. En la Figura 16 encontramos un gráfico similar que compara los algoritmos según su velocidad de procesado medida en fotogramas por segundo.

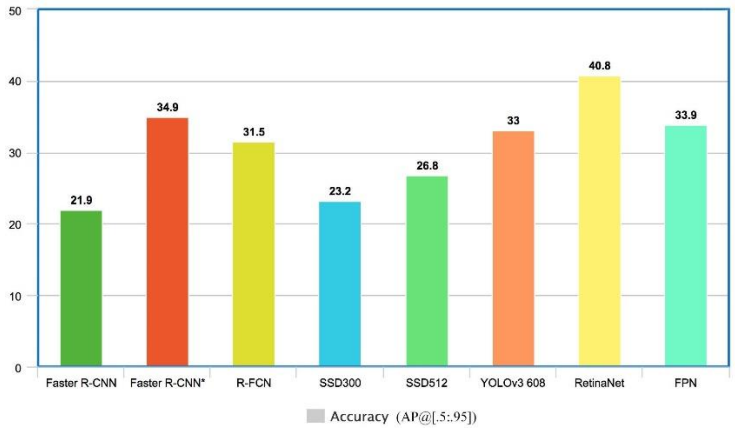


Figura 15. Comparativa de algoritmos según precisión [38]

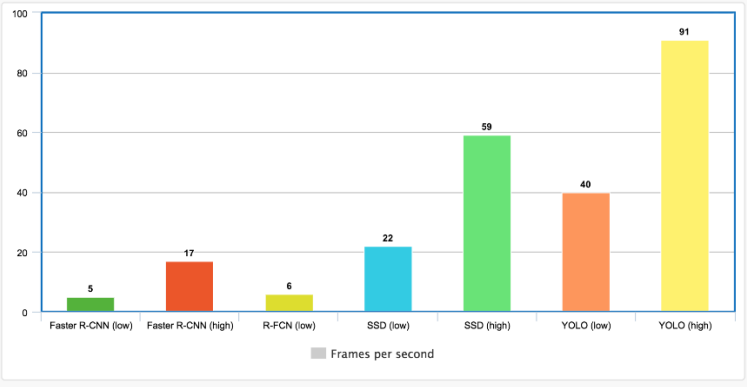


Figura 16. Comparativa de algoritmos según FPS [38]

En cuanto a los algoritmos de seguimiento mencionados en el capítulo anterior:

- Tracktor ++ es bastante preciso, pero un gran inconveniente es que no es viable para el seguimiento en tiempo real. Los resultados muestran una ejecución promedio de 3 FPS.
- TrackR-CNN es bueno porque proporciona segmentación como beneficio adicional. Pero al igual que con Tracktor ++, es difícil de utilizar para el seguimiento en tiempo real, ya que tiene una ejecución promedio de 1.6 FPS.
- DeepSORT es más rápido gracias a su simplicidad. Puede llegar a 16FPS de media manteniendo una buena precisión, lo que definitivamente lo convierte en una opción sólida para la detección y el seguimiento de múltiples objetos.

En conclusión, usaremos DeepSORT usando como detector YOLO en su versión más reciente (YOLOv4) ya que nos permitirán una velocidad de detección mucho mayor.

3.2 Descripción detallada

YOLOv4:

Como ya explicamos en apartados anteriores la rapidez de este algoritmo es debida a que es capaz de realizar predicciones en un solo paso de la red, lo que da sentido a su nombre (You Only Look Once).

La imagen de entrada es dividida en una cuadrícula que en este ejemplo será de 13x13 celdas:

Figura 17. Cuadrícula YOLO

Cada una de las celdas es responsable de predecir 5 cuadros delimitadores, de los que se genera una puntuación de confianza, es decir, la probabilidad de que dicho cuadro delimitador encierre verdaderamente un objeto.

Los cuadros delimitadores propuestos por cada celda tendrían un aspecto similar a la siguiente imagen.

Figura 18. Cuadros delimitadores YOLO

Las líneas más gruesas se corresponden con los cuadros que poseen mayor puntuación de confianza.

Para cada cuadro delimitador la celda predice una clase, utilizando una distribución de probabilidad sobre todas las clases posibles, dado que el modelo está entrenado con el conjunto de datos PASCAL VOC, las clases disponibles en éste serán las clases que puede detectar.

La puntuación de confianza para el cuadro y la predicción de la clase se combinan en una puntuación final que nos indica la probabilidad de que un cuadro delimitador contenga un tipo específico de objeto, lo que se ve reflejado en la siguiente imagen.

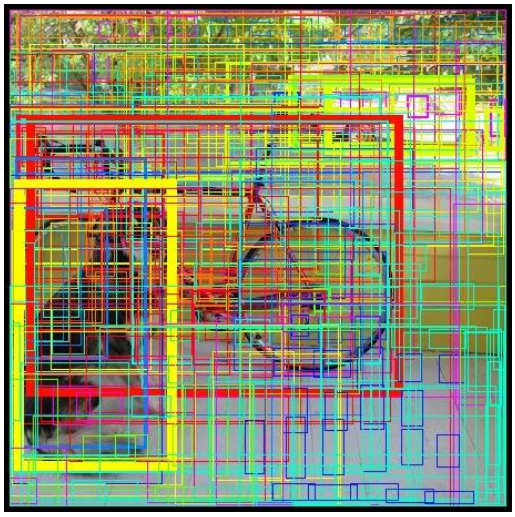


Figura 19. Cuadros y clases YOLO

Como teníamos una cuadrícula de 13x13 celdas y cada celda predice 5 cuadros delimitadores, el resultado eran 845 cuadros en total, la gran mayoría de ellos obtienen puntuaciones muy bajas, por lo que solo se mantienen aquellos con puntuaciones finales mayores o iguales al 30% (este umbral es variable en función de la precisión que se desee para el detector). Por tanto, aplicando este filtro, los cuadros en nuestra imagen final quedarían reducidos a lo que se muestra a continuación.

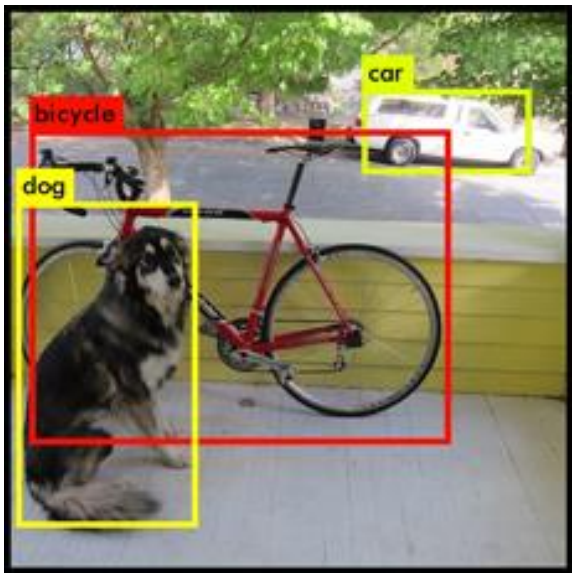


Figura 20. Predicciones YOLO

De las 845 propuestas solamente se conservan las tres predicciones finales que dieron mejores resultados, pero lo más relevante de esto es que todas esas predicciones se realizan al mismo tiempo, para todo este proceso la red neuronal solo se ha ejecutado una vez.

En cuanto a la arquitectura de la red, YOLO utiliza redes neuronales convolucionales simples. La red neuronal de YOLO en su primera versión constaba de 24 capas de convolución que hacen la función de extractores de características, estas capas de convolución se intercalan con capas de agrupación o pooling, y finalmente se utilizan dos capas densas encargadas de realizar las predicciones.

Existe también una versión reducida de YOLO que, aunque es menos precisa, requiere de menos capacidad de cómputo debido a que solo utiliza 9 capas de convolución. A esta versión reducida se le llama Tiny - YOLO.

La siguiente imagen muestra la arquitectura de las capas de la red de Tiny-YOLO.

Layer	kernel	stride	output shape
Input			(416, 416, 3)
Convolution	3×3	1	(416, 416, 16)
MaxPooling	2×2	2	(208, 208, 16)
Convolution	3×3	1	(208, 208, 32)
MaxPooling	2×2	2	(104, 104, 32)
Convolution	3×3	1	(104, 104, 64)
MaxPooling	2×2	2	(52, 52, 64)
Convolution	3×3	1	(52, 52, 128)
MaxPooling	2×2	2	(26, 26, 128)
Convolution	3×3	1	(26, 26, 256)
MaxPooling	2×2	2	(13, 13, 256)
Convolution	3×3	1	(13, 13, 512)
MaxPooling	2×2	1	(13, 13, 512)
Convolution	3×3	1	(13, 13, 1024)
Convolution	3×3	1	(13, 13, 1024)
Convolution	1×1	1	(13, 13, 125)

Figura 21. Capas de la CNN de Tiny-YOLO

Como podemos apreciar, el algoritmo procesa una imagen de entrada redimensionada a 416x416 píxeles y además 3 canales que se refieren a los canales de color RGB, esta imagen atravesará la red y saldrá como un vector de 13x13x125, 13x13 se corresponde con el tamaño de cuadrícula en que se divide la imagen, los 125 canales son el resultado de multiplicar los cinco cuadros que predice cada celda por los 25 datos que describen un cuadro delimitador (Posición X, Posición Y, alto, ancho, puntuación de confianza y las distribuciones de probabilidad sobre las 20 clases posibles).

En sus versiones posteriores se han realizado las siguientes mejoras sobre el modelo inicial:

- YOLOv2: aparece en diciembre de 2016 mejorando algunas deficiencias de la primera versión, como su problema en la detección de objetos pequeños. En esta segunda versión la red tiene 30 capas de convolución y aparecen los cuadros de anclaje, que son cuadros predefinidos por el usuario que dan una idea sobre la posición relativa y las dimensiones de los objetos a detectar.
- YOLOv3: se lanzó en abril de 2018, esta vez la red neuronal consta de 53 capas de convolución y utiliza detección en 3 escalas para detectar objetos desde tamaños muy pequeños a muy grandes.
- YOLOv4: lanzada en abril de 2020. Recibe la influencia de nuevos métodos de detección de objetos como “Bag of Freebies” y “Bag of Specials” para la fase de entrenamiento. Estos métodos de última generación, que incluyen CBN (Cross-iteration batch normalization), PAN (Path aggregation network), etc., son más eficientes para el entrenamiento con una única GPU, además aumentan la rapidez del modelo.

DeepSORT:

Como ya explicamos anteriormente DeepSORT es un algoritmo de seguimiento de objetos (popularmente conocido por su nombre en inglés “Object Tracking”) que ha mostrado notables resultados en el problema Multiple Object Tracking (MOT).

El funcionamiento del algoritmo sigue los siguientes pasos:

- Primero se detectan todos los objetos en la imagen usando un detector de objetos.
- En segundo lugar, las posiciones de los registros existentes previamente se actualizan mediante un filtro de Kalman.
- Luego, agrupan los registros en función de su edad (tiempo que llevan sin ser asociados a una nueva detección) y se ejecuta el algoritmo húngaro en cada uno de los grupos en orden de edad creciente.
- Todas los registros no coincidentes y no confirmados de primera edad se procesan utilizando el algoritmo SORT original.
- Finalmente, las detecciones sin coincidencias se establecen como nuevos registros.

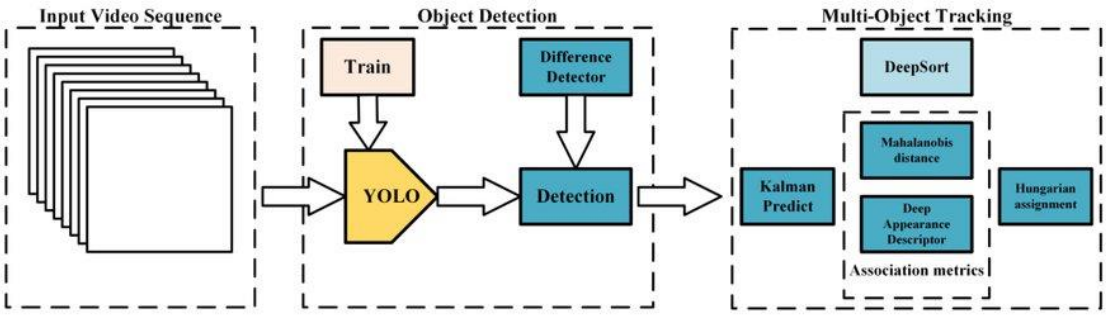


Figura 22. Esquema DeepSORT

El filtrado de Kalman es un componente crucial para DeepSORT. El escenario de seguimiento de Kalman se define en el espacio de estados de las siguientes ocho variables (u, v, a, h, u ', v', a ', h') donde “u,v” corresponde a las coordenadas del centro del cuadro delimitador, “a” es la relación de aspecto, “h” es la altura de la imagen, y el resto de variables corresponden respectivamente a la velocidad con la que varían las anteriores.

Las variables solo tienen factores de posición y velocidad absolutos, ya que asumimos que un modelo de velocidad lineal simple. El filtro de Kalman nos ayuda a tener en cuenta el ruido en la detección y utiliza el estado anterior para predecir un buen ajuste para los cuadros delimitadores.

Para cada detección creamos un registro que tiene toda la información de estado necesaria, este contiene el parámetro de edad necesario para detectar y eliminar aquellos objetos que fueron detectados por última vez hace mucho tiempo ya que estos son objetos que han abandonado la escena.

Una vez tenemos los nuevos cuadros delimitadores proporcionados por el filtro de Kalman el siguiente problema será asociar nuevas detecciones con las predicciones. Dado que se procesan de manera independiente, surge el problema de asignación. Para resolverlo necesitamos dos cosas: una métrica de distancia que permita cuantificar la asociación y un algoritmo eficiente para asociar los datos.

Como métrica de distancia los autores de DeepSORT decidieron utilizar la distancia de Mahalanobis, que se utiliza para incorporar información sobre el movimiento. Esta métrica es más precisa que la distancia euclidiana, ya que estamos midiendo efectivamente la distancia entre dos distribuciones de probabilidad.

Como algoritmo de asociación los autores usaron el "algoritmo húngaro", que es un algoritmo de optimización combinatoria muy eficaz y simple que resuelve el problema de asignación.

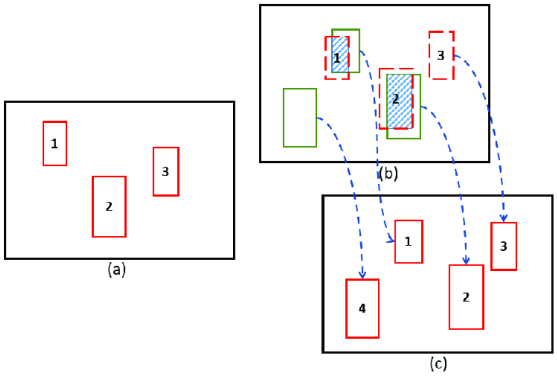


Figura 23. Asociación

A pesar de la efectividad del filtro de Kalman, falla en muchos de los escenarios del mundo real como son problemas de oclusión y de variación del punto de vista, etc.

Para mejorar esto, introdujeron otra métrica de distancia basada en la apariencia del objeto. Primero construyeron un clasificador sobre el conjunto de datos, lo entrenaron hasta que logró una precisión razonablemente buena y luego eliminaron la capa de clasificación final. Esto produce un único vector de características, a la espera de ser clasificado. Ese vector de características se convierte en nuestro descriptor de apariencia del objeto.

Name	Patch Size/Stride	Output Size
Conv 1	$3 \times 3/1$	$32 \times 128 \times 64$
Conv 2	$3 \times 3/1$	$32 \times 128 \times 64$
Max Pool 3	$3 \times 3/2$	$32 \times 64 \times 32$
Residual 4	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 5	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 6	$3 \times 3/2$	$64 \times 32 \times 16$
Residual 7	$3 \times 3/1$	$64 \times 32 \times 16$
Residual 8	$3 \times 3/2$	$128 \times 16 \times 8$
Residual 9	$3 \times 3/1$	$128 \times 16 \times 8$
Dense 10		128
Batch and ℓ_2 normalization		128

Figura 24. Descriptor de apariencia

En la imagen anterior se muestra la arquitectura de la red para obtener el vector de características de apariencia, la capa densa 10 es la que se corresponde con este vector de características de dimensiones 128x1.

De modo que la métrica final utilizada será:

$$D = \lambda * Dk + (1 - \lambda) * Da$$

Donde *Dk* es la distancia de Mahalanobis, *Da* es la distancia del coseno entre los vectores de características de apariencia y λ es un factor de ponderación.

3.3 Implementación

La solución propuesta se desarrollará en el lenguaje Python haciendo uso de Tensorflow. Tensorflow una biblioteca de código abierto para aprendizaje automático desarrollada por Google. Se utilizará también Keras, que es una famosa biblioteca de código abierto para el desarrollo de redes neuronales.

Para el desarrollo de la aplicación necesitaremos descargar los repositorios correspondientes a YOLOv4 y DeepSORT cuyos repositorios podemos encontrar en la web de GitHub, los enlaces a los repositorios están disponibles en la bibliografía.

Crearemos un espacio de trabajo donde alojaremos los archivos necesarios para ambos proyectos, aquí crearemos el código que relacionará ambos al que llamaremos **people_tracker.py** cuya explicación se detalla a continuación.

```
1
2   #initial imports
3   import os
4   import tensorflow as tf
5   from tensorflow.compat.v1 import ConfigProto
6   from tensorflow.compat.v1 import InteractiveSession
7   import time
8
9   #import image processing libraries
10  import cv2
11  import numpy as np
12  import matplotlib.pyplot as plt
13  from PIL import Image
14  import cv2
15  import numpy as np
16  import matplotlib.pyplot as plt
17
18  # deep sort imports
19  from deep_sort import preprocessing, nn_matching
20  from deep_sort.detection import Detection
21  from deep_sort.tracker import Tracker
22  from tools import generate_detections as gdet
23
24  #yolo imports
25  import core.utils as utils
26  from core.yolov4 import filter_boxes
27  from tensorflow.python.saved_model import tag_constants
28  from core.config import cfg
29
30  #flags imports
31  from absl import app, flags, logging
32  from absl.flags import FLAGS
33
34
```

Figura 25. people_tracker.py fragmento 1

En primer lugar, importamos librerías y otros scripts que necesitaremos más adelante.

```
41
42  #defining flags that will be used to change execution parameters
43  flags.DEFINE_string('weights', './weights/yolov4-416','path to weights file')
44  flags.DEFINE_integer('size', 416, 'resize images to')
45  flags.DEFINE_boolean('tiny', False, 'yolo or yolo-tiny')
46  flags.DEFINE_string('model', 'yolov4', 'model saved with save_model.py file')
47  flags.DEFINE_string('video', None, 'input video path, webcam by default')
48  flags.DEFINE_string('output', None, 'path to output video')
49  flags.DEFINE_float('iou', 0.45, 'iou threshold')
50  flags.DEFINE_float('score', 0.50, 'score threshold')
51  flags.DEFINE_boolean('hide', False, 'hide video output')
52  flags.DEFINE_string('save_info', None, 'path to file where info will be saved')
53  flags.DEFINE_boolean('count', False, 'count of people on screen')
54
```

Figura 26. people_tracker.py fragmento 2

Definimos los *flags* que son las variables que podremos cambiar desde la ventana de comandos antes de ejecutar el código. Se especifica el tipo de dato de la variable y además entre paréntesis aparece la estructura *(nombre de la variable, valor por defecto, descripción)*.

Weights: corresponde al archivo .weights que contiene los pesos de la red ya entrenada. Se puede modificar, aunque por defecto especifico la ruta al archivo en mi espacio de trabajo.

Size: valor en píxeles al que se redimensionarán los fotogramas de salida, por defecto 416.

Tiny: variable para especificar si se utilizará YOLO o su versión reducida YOLO-Tiny.

Model: indicamos que el modelo corresponde con el modelo guardado con el script save_model.py disponible en el repositorio de YOLO mediante el cual adaptamos el modelo para usarse con tensorflow.

Video: indica la ruta del vídeo que queremos procesar como entrada, si no se hace uso de esta variable se usará por defecto la imagen en tiempo real de la webcam.

Output: permite especificar un directorio donde guardar el video procesado, si no se especifica no se guardará.

Iou: Valor umbral de intersección sobre unión. La intersección sobre unión se usa en la detección de objetos a fin de medir la superposición entre el cuadro delimitador previsto y el real, cuanto más se acerque la predicción al real mayor será la intersección y mayor valor de IoU.

Score: Puntuación umbral para las predicciones. No se mostrarán predicciones con puntuación menor a este umbral.

Hide: permite ocultar la ventana emergente que aparece por defecto durante la ejecución mostrando la salida de vídeo.

Save_info: usando este flag podremos especificar la ruta a un archivo de texto en el que se guardará información detallada de las detecciones, por defecto esta opción no está habilitada.

Count: habilitando este flag se mostrará un contador con el número de personas.

```python
def main(_argv):

    # Definition of parameters
    max_cosine_distance = 0.4
    nn_budget = None
    nms_max_overlap = 1.0

    # initialize deep sort
    model_filename = 'model_data/mars-small128.pb'
    encoder = gdet.create_box_encoder(model_filename, batch_size=1)
    # calculate cosine distance metric
    metric = nn_matching.NearestNeighborDistanceMetric("cosine", max_cosine_distance, nn_budget)
    # initialize tracker
    tracker = Tracker(metric)

    # load configuration for object detector
    config = ConfigProto()
    config.gpu_options.allow_growth = True
    session = InteractiveSession(config=config)
    STRIDES, ANCHORS, NUM_CLASS, XYSCALE = utils.load_config(FLAGS)
    input_size = FLAGS.size
    video_path = FLAGS.video

    #load tensorflow saved model
    saved_model_loaded = tf.saved_model.load(FLAGS.weights, tags=[tag_constants.SERVING])
    predict = saved_model_loaded.signatures['serving_default']
```

Figura 27. people_tracker.py fragmento 3

Definimos algunos parámetros de DeepSORT, inicializamos el Tracker y cargamos la configuración y el modelo de YOLOv4.

```python
    #open video input
    if video_path:
        in_video = cv2.VideoCapture(video_path)
    else:
        in_video = cv2.VideoCapture(0)

```

Figura 28. people_tracker.py fragmento 4

Preparamos la entrada de video, si así lo hemos establecido en flag se tomará el vídeo de la ruta indicada, en otro caso se activará la webcam.

```python
    #by default video is not saved
    out = None

    # prepare video to be saved if flag is set
    if FLAGS.output:
        width = int(in_video.get(cv2.CAP_PROP_FRAME_WIDTH))
        height = int(in_video.get(cv2.CAP_PROP_FRAME_HEIGHT))
        fps = int(in_video.get(cv2.CAP_PROP_FPS))
        codec = cv2.VideoWriter_fourcc(*'XVID')
        out = cv2.VideoWriter(FLAGS.output, codec, fps, (width, height))

```

Figura 29. people_tracker.py fragmento 5

Por defecto la salida de vídeo no se guardará, en cambio si el flag está activado preparamos el vídeo para guardarse en la ruta indicada.

```
#prepare information file if flag is set
if FLAGS.save_info:
    info_file = open(FLAGS.save_info, "w+")
```

Figura 30. people_tracker.py fragmento 6

Preparamos también el archivo donde se guardará la información detallada de los seguimientos si así se ha establecido. Si el archivo especificado en la ruta no existe se creará.

```
106     while True:
107         return_value, frame = in_video.read()
108         if return_value:
109             frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
110             image = Image.fromarray(frame)
111         else:
112             print('Something went wrong')
113             break
114 
115         frame_size = frame.shape[:2]
116         image_data = cv2.resize(frame, (input_size, input_size))
117         image_data = image_data / 255.
118         image_data = image_data[np.newaxis, ...].astype(np.float32)
119         t0 = time.time()
120 
```

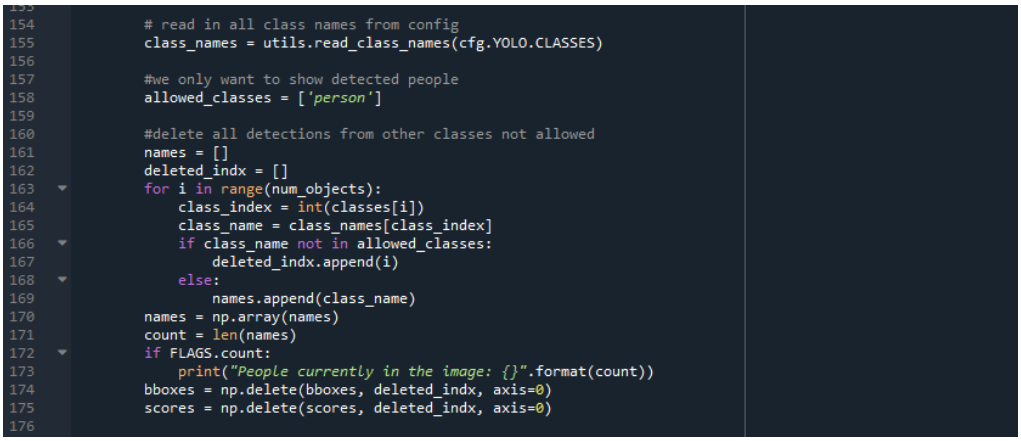
Figura 31. people_tracker.py fragmento 7

Tanto esta parte como los siguientes fragmentos se ejecutarán para cada fotograma mientras el vídeo continúe. Tomamos el fotograma, lo redimensionamos y guardamos el instante de tiempo inicial para posteriormente calcular los fotogramas por segundo.

```
121         #run detection
122         batch_data = tf.constant(image_data)
123         pred_bbox = predict(batch_data)
124         for key, value in pred_bbox.items():
125             boxes = value[:, :, 0:4]
126             pred_conf = value[:, :, 4:]
127 
128         boxes, scores, classes, valid_detections = tf.image.combined_non_max_suppression(
129             boxes=tf.reshape(boxes, (tf.shape(boxes)[0], -1, 1, 4)),
130             scores=tf.reshape(
131                 pred_conf, (tf.shape(pred_conf)[0], -1, tf.shape(pred_conf)[-1])),
132             max_output_size_per_class=50,
133             max_total_size=50,
134             iou_threshold=FLAGS.iou,
135             score_threshold=FLAGS.score
136         )
137 
138         # convert data to numpy arrays and delete non used elements
139         num_objects = valid_detections.numpy()[0]
140         bboxes = boxes.numpy()[0]
141         bboxes = bboxes[0:int(num_objects)]
142         scores = scores.numpy()[0]
143         scores = scores[0:int(num_objects)]
144         classes = classes.numpy()[0]
145         classes = classes[0:int(num_objects)]
146 
```

Figura 32. people_tracker.py fragmento 8

Se realizan las nuevas detecciones y se eliminan las detecciones no válidas.



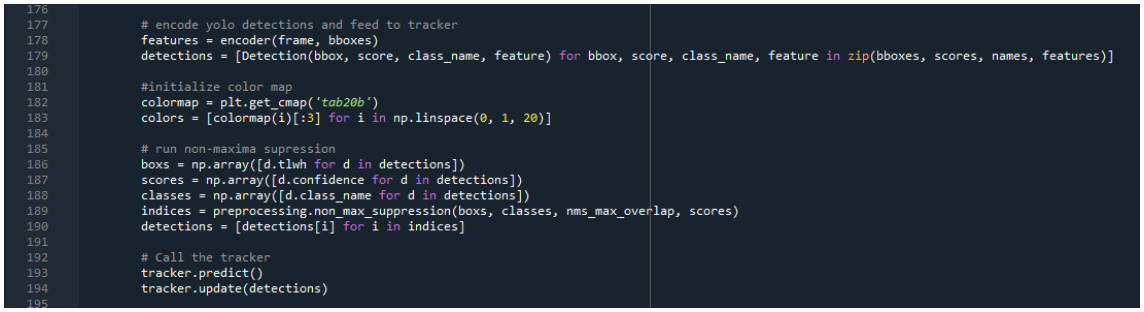
```
        # read in all class names from config
        class_names = utils.read_class_names(cfg.YOLO.CLASSES)

        #we only want to show detected people
        allowed_classes = ['person']

        #delete all detections from other classes not allowed
        names = []
        deleted_indx = []
        for i in range(num_objects):
            class_index = int(classes[i])
            class_name = class_names[class_index]
            if class_name not in allowed_classes:
                deleted_indx.append(i)
            else:
                names.append(class_name)
        names = np.array(names)
        count = len(names)
        if FLAGS.count:
            print("People currently in the image: {}".format(count))
        bboxes = np.delete(bboxes, deleted_indx, axis=0)
        scores = np.delete(scores, deleted_indx, axis=0)
```

Figura 33. people_tracker.py fragmento 9

El detector está entrenado para reconocer gran variedad de clases, pero solamente queremos hacer el seguimiento en personas. Así que, recorremos las detecciones buscando la clase a la que corresponden, y eliminamos todas las que no corresponden a la clase *person.* Aquí se cuenta el número de personas que aparecen actualmente en la imagen y se imprimen en la consola de comandos.



```
        # encode yolo detections and feed to tracker
        features = encoder(frame, bboxes)
        detections = [Detection(bbox, score, class_name, feature) for bbox, score, class_name, feature in zip(bboxes, scores, names, features)]

        #initialize color map
        colormap = plt.get_cmap('tab20b')
        colors = [colormap(i)[:3] for i in np.linspace(0, 1, 20)]

        # run non-maxima supression
        boxs = np.array([d.tlwh for d in detections])
        scores = np.array([d.confidence for d in detections])
        classes = np.array([d.class_name for d in detections])
        indices = preprocessing.non_max_suppression(boxs, classes, nms_max_overlap, scores)
        detections = [detections[i] for i in indices]

        # Call the tracker
        tracker.predict()
        tracker.update(detections)
```

Figura 34. people_tracker.py fragmento 10

Codificamos las detecciones y se las pasamos al tracker para que actualice las predicciones.

```
        # update tracks
        for track in tracker.tracks:
            if not track.is_confirmed() or track.time_since_update > 1:
                continue
            bbox = track.to_tlbr()
            class_name = track.get_class()

        # draw bbox on screen
            color = colors[int(track.track_id) % len(colors)]
            color = [i * 255 for i in color]
            cv2.rectangle(frame, (int(bbox[0]), int(bbox[1])), (int(bbox[2]), int(bbox[3])), color, 2)
            cv2.rectangle(frame, (int(bbox[0]), int(bbox[1]-30)), (int(bbox[0])+(len(class_name)+len(str(track.track_id)))*17, int(bbox[1])), color, -1)
            cv2.putText(frame, class_name + "-" + str(track.track_id),(int(bbox[0]), int(bbox[1]-10)),0, 0.75, (255,255,255),2)

        # if enable save_info flag then save the details in the specified file
            if FLAGS.save_info:
                info_file.write("Person id: {}, Position (Xmin, Ymin, Xmax, Ymax): {}, Timestamp: {}".format(str(track.track_id), (int(bbox[0]), int(bbox[1]), int(bbox[2]), int(bbox[3])),
                                                                                                            time.strftime("%a %b %d %H:%M:%S %Y")) + os.linesep)
```

Figura 35. people_tracker.py fragmento 11

Para cada una de las predicciones del tracker se dibuja el cuadro delimitador junto con la clase y el identificador, si se ha especificado guardar información detallada mediante el flag *save_info* se guardará en el archivo correspondiente una línea para cada persona que muestra su identificador, posición y marca temporal en formato fecha y hora, he elegido este tipo de marca temporal porque es más coherente para detecciones en tiempo real.

```
214
215         #get frames per second
216         fps = 1.0 / (time.time() - t0)
217         print("FPS: "+ str(round(fps,2)))
218
219         result = np.asarray(frame)
220         result = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
221
222         if not FLAGS.hide:
223             cv2.imshow("People Tracker Output", result)
224
225         # if output flag is set, save video file
226         if FLAGS.output:
227             out.write(result)
228         if cv2.waitKey(1) & 0xFF == ord('q'): break
229     cv2.destroyAllWindows()
230     if FLAGS.save_info:
231         info_file.close()
```

Figura 36. people_tracker.py fragmento 12

Por último, se obtienen los fotogramas por segundo haciendo uso del instante de tiempo inicial *t0* guardado cuando comenzó a procesarse el fotograma y el instante de tiempo actual en que ya ha sido procesado. Guardamos la imagen con las detecciones en la variable *result* que será mostrada en una ventana emergente con el título *People Tracker Output* si el flag *hide* no ha sido activado, también se guardará en el vídeo de salida si se ha especificado en el flag *output.*

Además, la ejecución del vídeo podrá detenerse pulsando la tecla *q*. Cuando la ejecución finalice o sea detenida desaparecerá la ventana emergente y se cerrará el archivo de información si estaba en uso.

3.4 Ejecución y resultados obtenidos

Para la ejecución del archivo utilizaremos una consola de comandos de Anaconda, donde utilizaremos dos entornos diferentes, uno preparado para aceleración con GPU y otro sin ella para mostrar la diferencia.

Las pruebas se han realizado sobre varios clips de vídeo captados por una cámara situada en Temple Bar, Dublín.

Figura 38. Fotograma clip de prueba 1

Figura 37. Fotograma clip de prueba 2

Una vez abierta la ventana de comandos accedemos al directorio de nuestro proyecto y activamos el entorno para GPU, una vez hecho esto ejecutamos la siguiente línea.

```
(yolov4-gpu) C:\PeopleTracker>python people_tracker.py --model yolov4 --video ./dublin_video1.mp4 --save_info ./info1.txt --output ./out_video1.avi --count True
```

Como podemos ver ejecuta el archivo people_tracker.py utilizando distintos parámetros como son el modelo, la ruta del vídeo de entrada, la ruta al archivo donde debe guardarse la información detallada y la ruta donde debe guardarse el archivo de salida, solicitamos además que cuente el número de personas.

Mientras está en ejecución podemos ver que la consola de comandos nos muestra un valor de unos 7-8 FPS. Además, se ha abierto la ventana emergente que muestra la salida de vídeo. Como podemos ver el valor del contador de personas mostrado se corresponde con el valor de personas detectadas en la imagen.

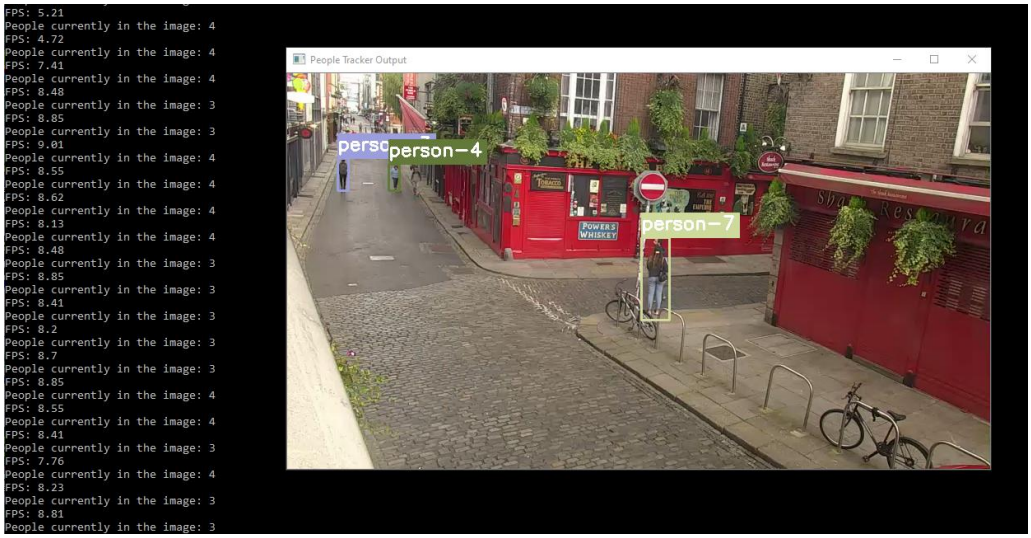


Figura 39. Ejemplo de ejecución con GPU 1

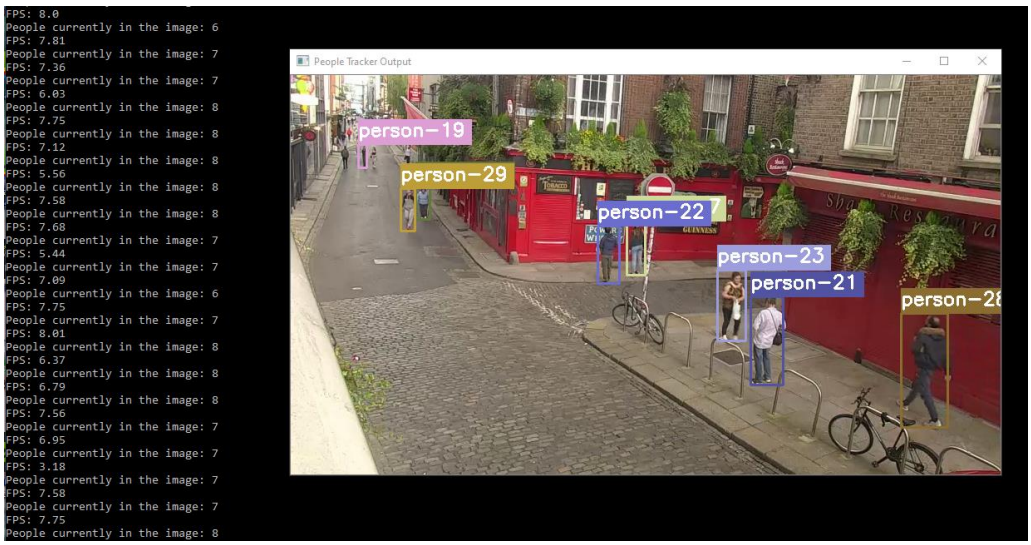


Figura 40. Ejemplo de ejecución con GPU 2

En esta imagen se muestra un ejemplo de ejecución en entorno CPU, podemos ver que cuando no está usando una tarjeta gráfica la velocidad con la que es capaz de procesar los fotogramas reduce significantemente no llegando a veces a un fotograma por segundo, lo cual sería impensable para un sistema de detección-seguimiento a tiempo real.

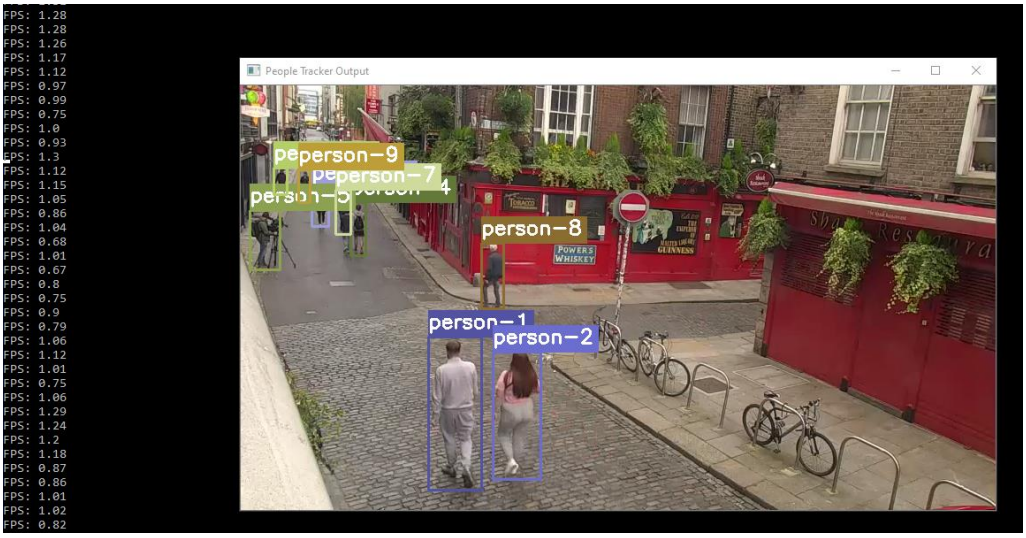


Figura 41. Ejemplo de ejecución CPU

En cuanto a resultados de detección y seguimiento se observa que el sistema tiene en ocasiones dificultades para detectar a las personas si ya se han alejado mucho. El algoritmo de seguimiento comete algunos errores al reidentificar a una persona que ha salido del encuadre y vuelve a aparecer y en ocasiones cambia los identificadores cuando dos personas se cruzan.

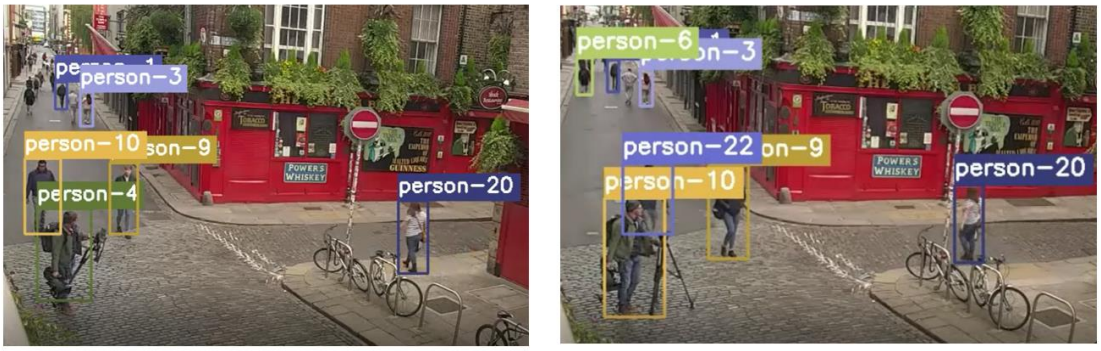


Figura 42. Secuencia errónea

En la figura anterior podemos ver a la izquierda una persona que sostiene un trípode que inicialmente tenía el identificador 4, en el momento en que la persona con identificador 10 pasa por detrás de él, sus identificadores cambian, el algoritmo ha asignado a la persona del trípode el identificador de la persona que pasó por detrás, y a éste otro un identificador nuevo como si nunca hubiera aparecido antes.

Estos son, en cambio, un par de ejemplos de secuencias donde el algoritmo funciona correctamente.

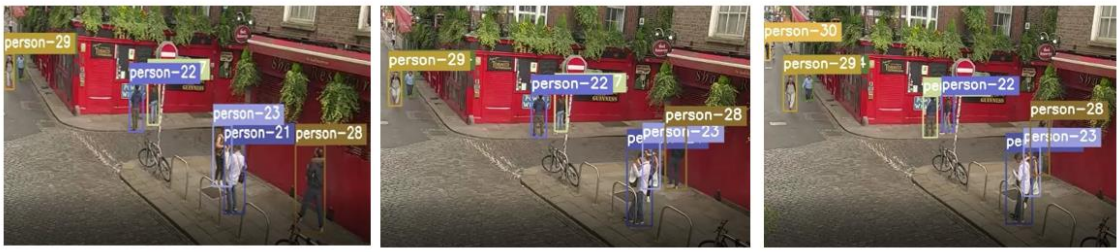


Figura 43. Secuencia correcta 1

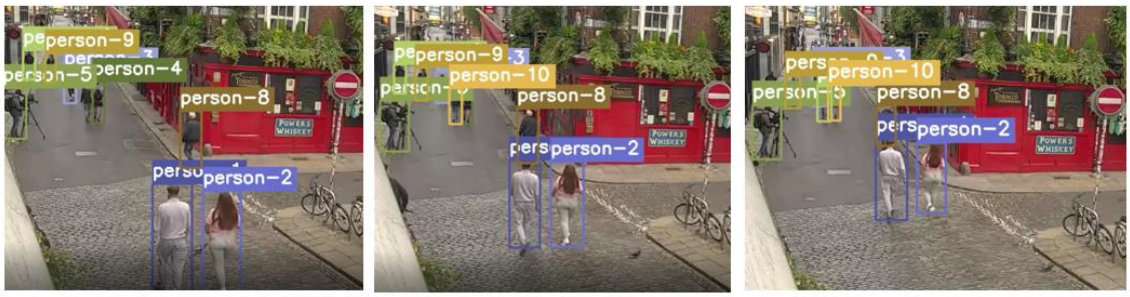


Figura 44. Secuencia correcta 2

La siguiente figura muestra una imagen de ejemplo del archivo obtenido activando el flag *save_info* con la información registrada tras la ejecución.

```
Person id: 1, Position (Xmin, Ymin, Xmax, Ymax): (287, 258, 343, 373), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 2, Position (Xmin, Ymin, Xmax, Ymax): (207, 241, 260, 379), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 3, Position (Xmin, Ymin, Xmax, Ymax): (167, 169, 197, 231), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 4, Position (Xmin, Ymin, Xmax, Ymax): (174, 121, 189, 156), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 5, Position (Xmin, Ymin, Xmax, Ymax): (94, 90, 108, 117), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 1, Position (Xmin, Ymin, Xmax, Ymax): (285, 255, 339, 369), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 2, Position (Xmin, Ymin, Xmax, Ymax): (207, 241, 259, 378), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 3, Position (Xmin, Ymin, Xmax, Ymax): (164, 167, 196, 233), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 4, Position (Xmin, Ymin, Xmax, Ymax): (174, 121, 188, 156), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 5, Position (Xmin, Ymin, Xmax, Ymax): (95, 91, 108, 117), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 1, Position (Xmin, Ymin, Xmax, Ymax): (282, 253, 337, 368), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 2, Position (Xmin, Ymin, Xmax, Ymax): (206, 241, 258, 378), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 3, Position (Xmin, Ymin, Xmax, Ymax): (163, 165, 195, 232), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 4, Position (Xmin, Ymin, Xmax, Ymax): (174, 122, 188, 156), Timestamp: Thu Sep 03 17:42:22 2020
Person id: 5, Position (Xmin, Ymin, Xmax, Ymax): (94, 91, 108, 118), Timestamp: Thu Sep 03 17:42:22 2020
```

Figura 45. Información registrada.

Capítulo 4. Caso práctico: Detección de emociones en rostro

En este capítulo detallaremos el proceso seguido para el desarrollo de un modelo capaz de clasificar imágenes en función de dos expresiones básicas en su rostro, expresión neutra o feliz. Además, se ha querido hacer referencia al problema del sesgo algorítmico, que está causando polémicas debido a negligencias de algunos sistemas de AI.

4.1 Elección de las capas de la red

¿Cuántas capas debe tener? ¿Cómo son estas capas? ¿Cuántas neuronas debe tener cada capa? Estas son algunas de las preguntas más frecuentes a la hora de desarrollar una red neuronal, la respuesta es que no existe una solución exacta, el diseño de la arquitectura de una red neuronal es un proceso complejo y depende del propósito que tengamos para nuestra red.

Podemos tomar distintos enfoques para "descubrir" qué arquitectura funcionaría bien para nuestro problema, como, por ejemplo:

- Experimentación: Este es el método a seguir cuando el problema que afrontamos es muy novedoso y no hay información anterior que ayude a resolver la incógnita. Se puede hacer mediante pruebas sistemáticas, usando heurísticos que faciliten la tarea, etc.

- Buscar arquitecturas ya usadas para problemas similares: Es frecuente que el problema que estemos intentando resolver ya haya sido resuelto por alguien anteriormente, o que sea similar. La idea en estos casos es adaptar a nuestras necesidades arquitecturas de redes que ya sabemos que funcionan bien para problemas similares.

La clasificación de imágenes es un campo de la inteligencia artificial bastante avanzado en el que las redes neuronales convolucionales son las que mejores resultados han proporcionado. Hay equipos de investigación totalmente dedicados a desarrollar arquitecturas de aprendizaje profundo para CNN. Ya vimos en el capítulo 2 las capas básicas que podemos encontrar en una CNN, veamos ahora como se distribuyen estas capas en algunas de las arquitecturas más conocidas como son VGGNet, ResNet, Inception o Xception.

VGGNet [30]

Fue una de las primeras arquitecturas en aparecer, fue presentada por Simonyan y Zisserman en 2014. Se trata de una arquitectura simple que usa bloques compuestos por capas convolucionales con filtros o kernels de tamaño 3x3. Además, entre las capas de convolución se intercalan bloques de agrupación (Max Pooling) para reducir a la mitad el tamaño de los mapas de activación obtenidos. Finalmente, se utiliza un bloque de clasificación, que consta de dos capas densas de 4096 neuronas cada una, y la última capa, que es la capa de salida, de 1000 neuronas.

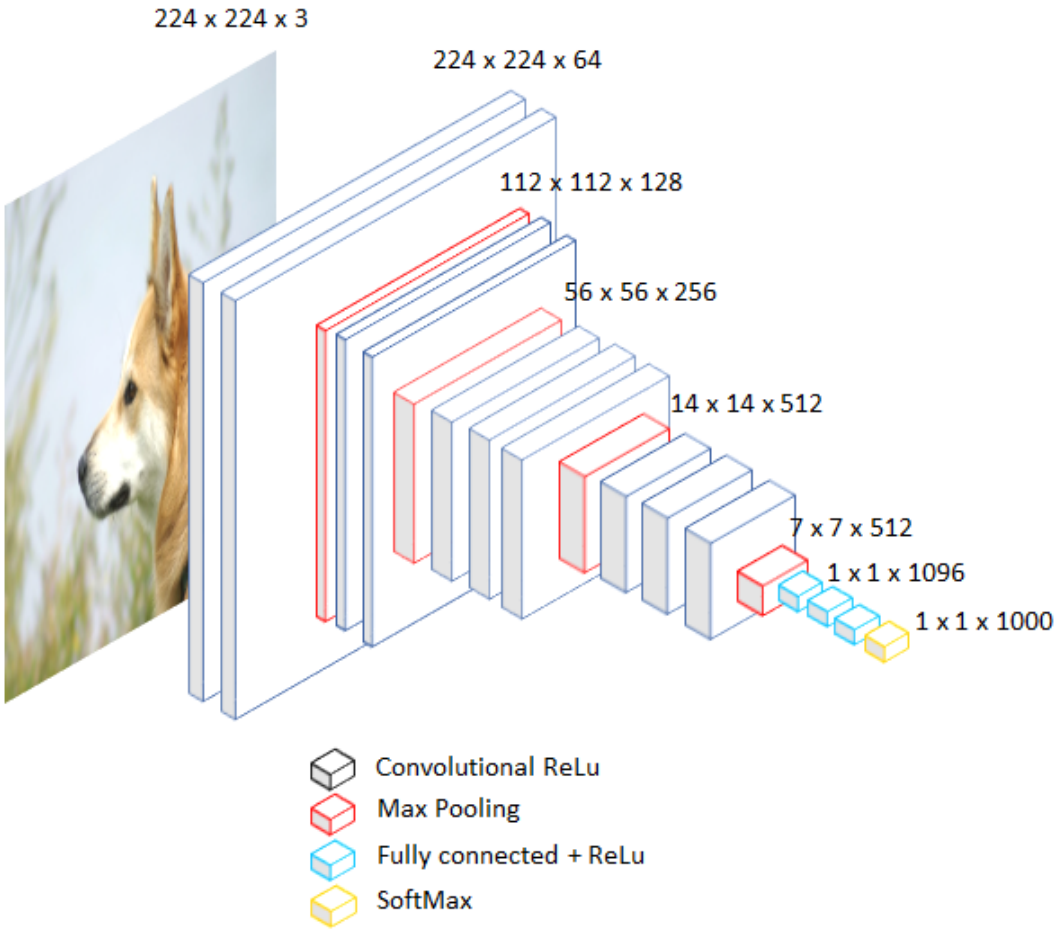


Figura 46. Arquitectura de VGGNet

Existen dos redes VGGNet diferentes, VGG16 y VGG19, que corresponden respectivamente con el número total de capas convolucionales y densas que tiene cada una.

Algunos de sus inconvenientes son el elevado tiempo de entrenamiento que requiere y el número de parámetros.

ResNet [31]

Se descubrió que las redes neuronales clásicas dejaban de funcionar bien a medida que la profundidad de la red supera cierto umbral. En la siguiente imagen podemos ver unos gráficos que aportó el propio autor de ResNet donde se registra el error de entrenamiento y test del conjunto de datos CIFAR-10 para una CNN clásica con 20 y 56 capas, y podemos apreciar como el error aumenta con la profundidad de la red.

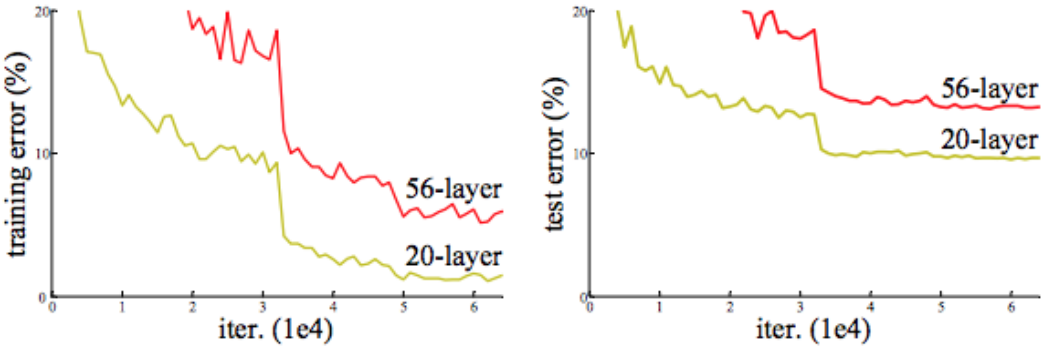


Figura 47. Gráfico de error en red profunda

La arquitectura ResNet aparece en 2015 solucionando el problema de la profundidad de la red mediante la introducción de un nuevo concepto: el bloque de aprendizaje residual.

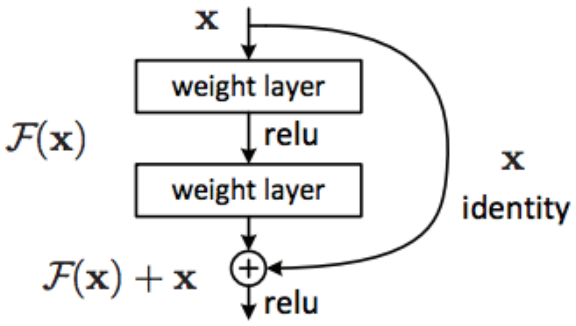


Figura 48. Bloque de aprendizaje residual

En las redes neuronales tradicionales, la salida de cada capa conecta con la siguiente capa. En una red con bloques residuales, la salida de cada capa no solo va a la siguiente capa sino también a siguientes capas que se encuentran unos saltos de distancia. La idea es omitir el entrenamiento de algunas capas para así evitar la degradación, esto se lleva a cabo mediante el uso de una función de identidad.

Inception [32]

Parte de la idea de que el área que ocupa un objeto de importancia en una imagen puede variar mucho. Por ejemplo, si el objeto está muy cerca el área será grande, por el contrario, si se encuentra alejado el área que ocupa en la imagen será menor. Esto tiene importancia a la hora de elegir el tamaño del filtro o kernel.

Esta arquitectura fue introducida en 2014 y utiliza bloques de convolución con filtros de diferentes tamaños que después se concatenan para extraer características a distintas escalas. Esto hace que la red se haga más amplia pero no más profunda.

Para disminuir costes computacionales los autores limitan el número de canales de entrada agregando una convolución adicional 1x1 antes de las convoluciones 3x3 y 5x5.

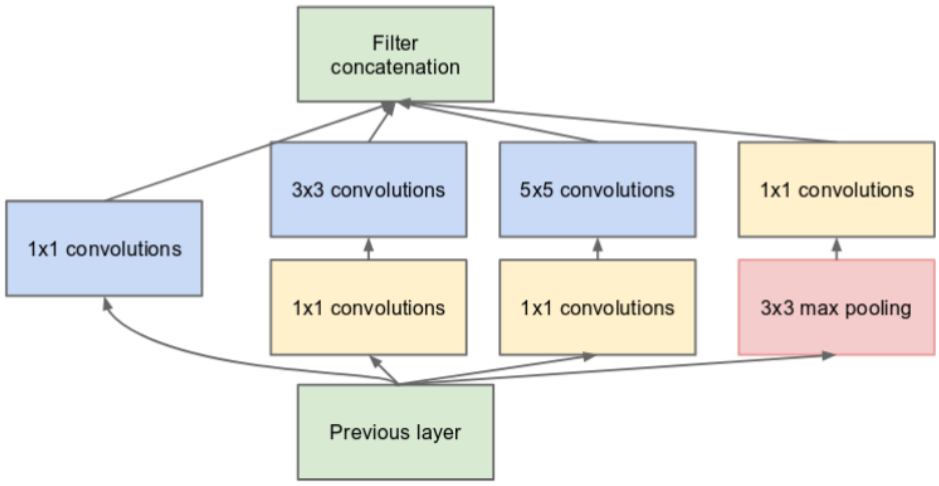


Figura 49. Filtros Inception

Tras la primera versión de Inception surgen Inception v2 e Inception v3, que intentan evitar los cuellos de botella representacionales (que se producen cuando las dimensiones de entrada a una capa reducen drásticamente ya que se pierde información) y tener cálculos más eficientes mediante el uso de métodos de factorización.

Surgen también después Inception v4 e Inception-Resnet, que intentan incluir la idea de los bloques residuales de ResNet.

Xception [33]

Esta arquitectura fue propuesta por François Chollet (el creador de Keras) y parte de la idea de Inception donde la convolución 1 × 1 se realiza antes de cualquier convolución espacial n × n.

La convolución original separable en profundidad se realizaba de la siguiente forma:

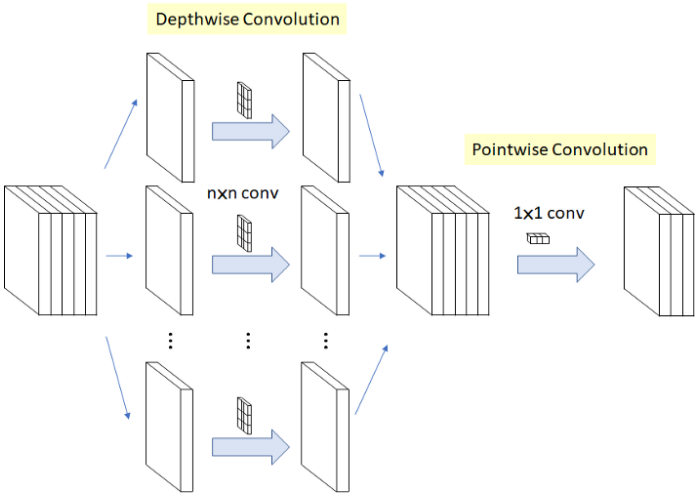


Figura 50. Convolución separable en profundidad

Se trata de una convolución en profundidad seguida de una convolución puntual, esto se corresponde respectivamente con una convolución espacial nxn en canal y una convolución 1x1 para redimensionar.

Lo que propone Xception a raíz de lo aprendido de Inception es modificar esta convolución, creando así una convolución separable en profundidad modificada, que funciona de la siguiente forma:

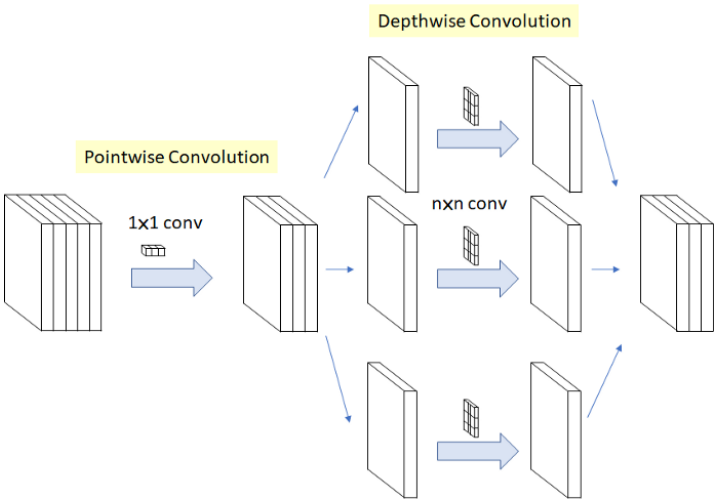


Figura 51. Convolución separable en profundidad modificada

En este caso se realiza primero la convolución puntual 1x1 y después la convolución nxn.

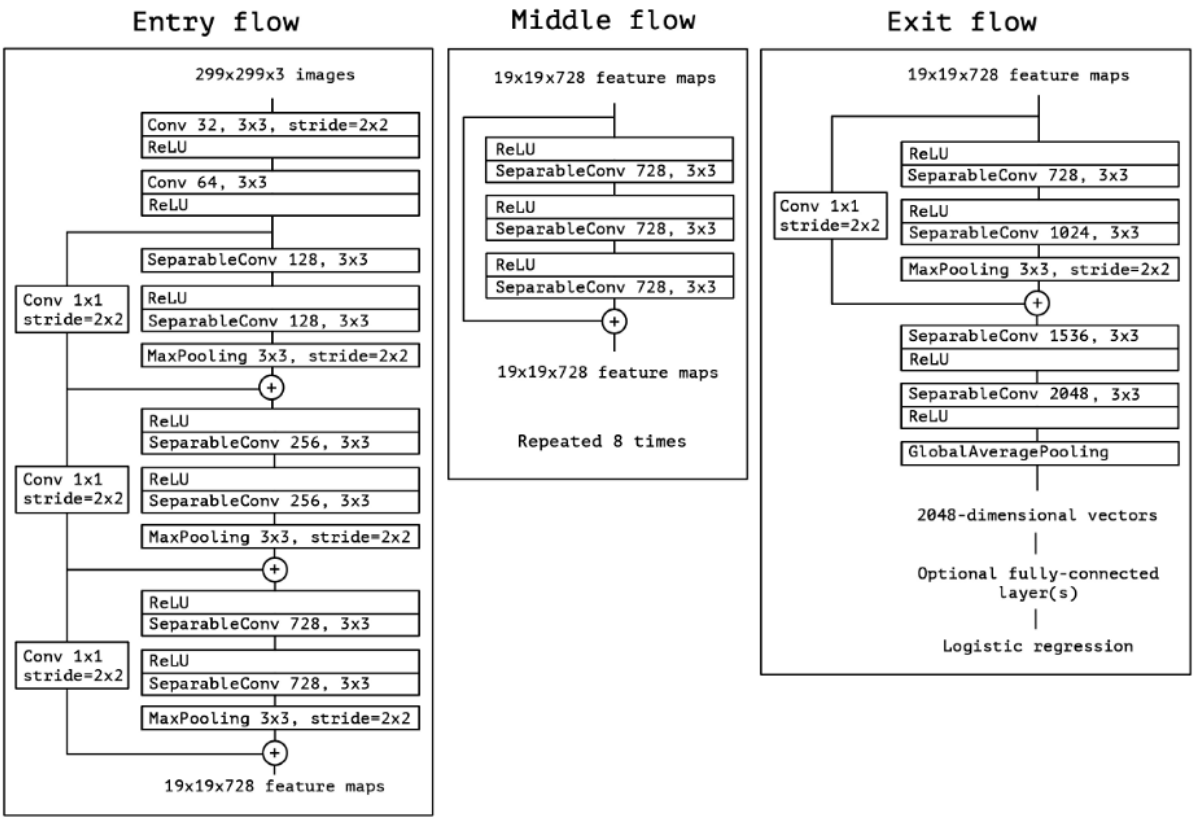


Figura 52. Arquitectura Xception

En la figura anterior vemos la arquitectura en capas de Xception, como podemos ver, SeparableConv es la convolución separable en profundidad modificada. También se pueden ver en la figura que aparecen bloques residuales heredados de ResNet. Esto ayuda a incrementar la precisión de los modelos entrenados con esta arquitectura, como podemos ver en el siguiente gráfico.

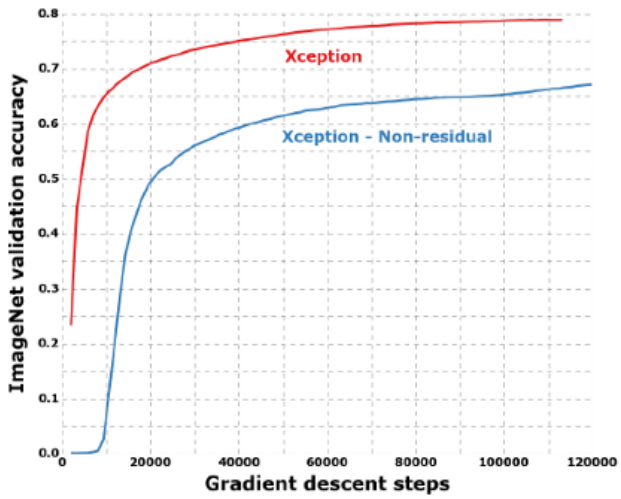


Figura 53. Comparación Xception con/sin bloques residuales

Para decidir qué tipo de arquitectura se adapta más a nuestro problema es necesario también tener en cuenta como es el conjunto de datos que utilizaremos para el entrenamiento.

Las imágenes serán procesadas antes para obtener únicamente el rostro de la persona centrado en una imagen de 48x48 píxeles, esto significa que los datos de interés en la imagen ocupan siempre áreas similares, por lo que no tiene interés el uso de convoluciones con filtros variados como proponía Inception.

Por tanto, el tipo de arquitectura que se adapta más a nuestras necesidades es la arquitectura de Xception, que como ya comentamos anteriormente incorpora también los bloques residuales de ResNet.

La arquitectura de nuestra red finalmente está basada en las características de Xception. En el primer módulo se ha modificado levemente el número de capas y también el tamaño de los filtros en las capas de convolución. En el siguiente esquema se aprecian las capas donde se indican de izquierda a derecha los filtros, el tamaño de kernel y el stride utilizado, que indica el valor de desplazamiento del filtro.

Aunque no se especifica en los esquemas, tras cada convolución se aplica normalización por lotes. Es un método utilizado en redes neuronales para conseguir mayor rapidez y estabilidad mediante la normalización de la capa de entrada volviendo a centrar y escalar.

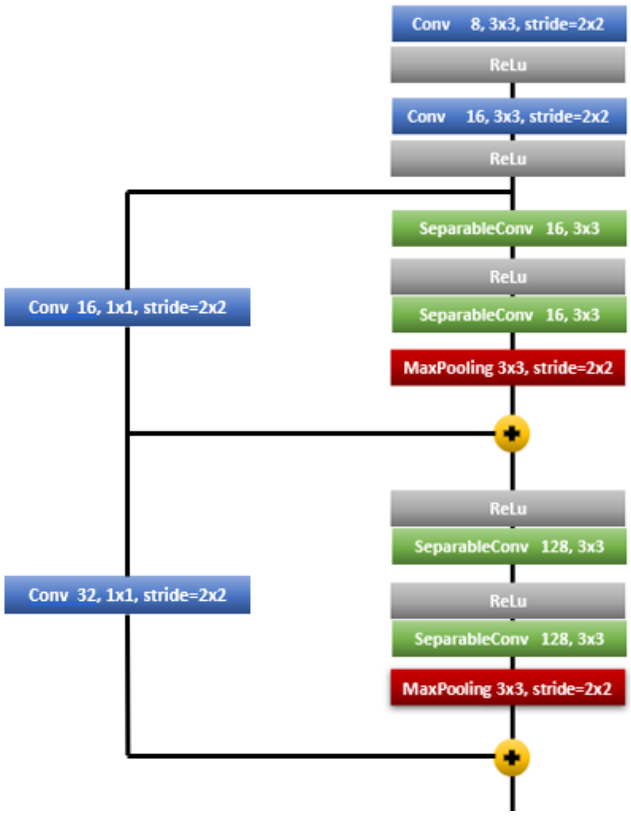


Figura 54. Módulo 1 de la arquitectura de la red

Se ha conservado la estructura del módulo intermedio cambiando también el número de filtros. Además, este conjunto de capas en la arquitectura Xception se repetían sucesivamente ocho veces, en este caso, lo haremos solamente cuatro veces.

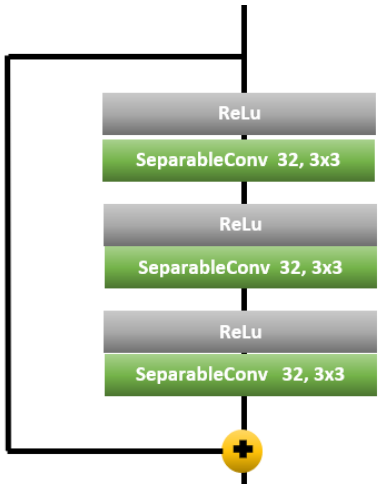


Figura 55. Módulo 2 de la arquitectura de la red

En cuanto al módulo final se conserva la arquitectura cambiando el tamaño de los filtros y se añade una capa densa o capa totalmente conectada con activación SoftMax al final de la estructura para tener el mismo número de salidas que de clases, en este caso dos.

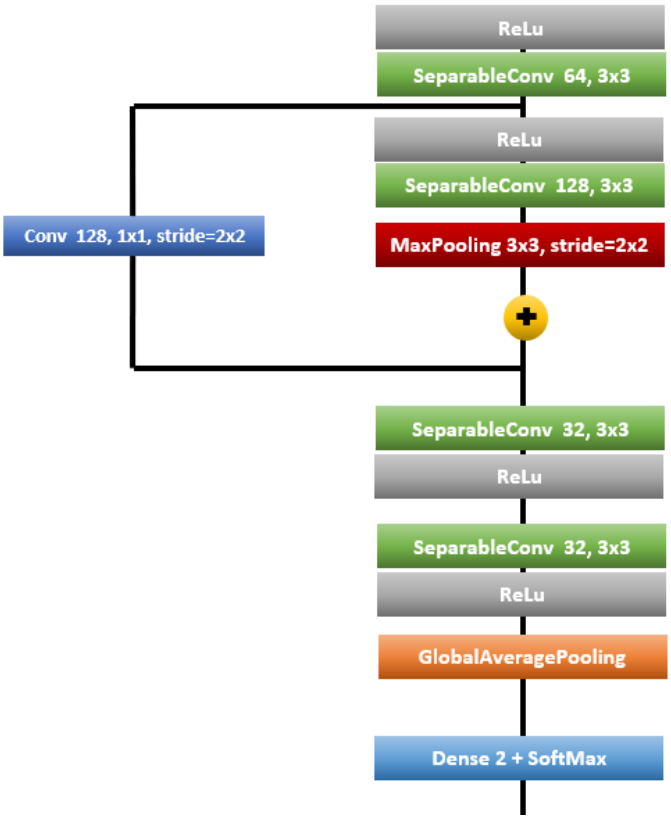


Figura 56. Módulo 3 de la arquitectura de la red

4.2 Conjunto de datos

Como ya se mencionó anteriormente existe un problema denominado sesgo algorítmico. Este problema ocurre cuando las personas que generan el conjunto de datos y entrenan los modelos inducen en ellos involuntariamente sus propios prejuicios.

Este problema ha desencadenado situaciones injustas de racismo en la inteligencia artificial, entre otras:

- Detenciones erróneas debido a algoritmos policiales de reconocimiento facial cuya precisión es mucho menor en personas de etnia no caucásica [41].
- Discriminación en el trato a pacientes por un software de toma de decisiones utilizado en hospitales de EE.UU. que daba preferencia a personas blancas antes que a negras [40].
- En 2015, Google tuvo que disculparse después de que su aplicación de reconocimiento de imágenes inicialmente etiquetara a los afroamericanos como "gorilas" [42].

Este problema es en gran parte ocasionado por los conjuntos de datos de entrenamiento. La variedad en el físico de la raza humana es muy amplia y no podemos generalizar el concepto visual de persona a la imagen de unas pocas personas. Para conseguir un algoritmo justo se necesitan enormes conjuntos de datos muy variados en función de etnias, edades, género, y demás características para tener en cuenta la gran diversidad.

El objetivo del conjunto de datos no será tratar este problema del sesgo ya que no es fácil conseguir conjuntos de datos tan grandes y variados y además el coste computacional de procesarlos sería enorme. En su lugar se ha intentado contribuir a este problema de forma simbólica y haciendo también un pequeño gesto al movimiento Black Lives Matter, de modo que el conjunto de datos reunido se centra en rostros de personas de piel negra.

En total se han recogido 11.929 imágenes, las cuales han sido preprocesadas para detectar el cuadrado que delimita el rostro, recortarlo, redimensionar la imagen a 48x48 píxeles y eliminar los canales de color obteniendo una imagen en blanco y negro. Para hacer esto se ha elaborado el siguiente script haciendo uso de las bibliotecas OpenCV y face_recognition.

```
import sys
import os
import dlib
from skimage import io
import cv2
import face_recognition


for  listdir in os.listdir("D:/Database"):
    for listdir2 in os.listdir("D:/Database/"+listdir):
        i=0;
        for fileName in os.listdir("D:/Database/"+listdir+"/"+listdir2):
            i=i+1
            face_image = cv2.imread("D:/Database/"+listdir+"/"+listdir2+"/"+fileName)
            face_locations= face_recognition.face_locations(face_image)
            top, right, bottom, left = face_locations[0]
            face_image = face_image[top:bottom, left:right]
            face_image = cv2.resize(face_image, (48,48))
            face_image = cv2.cvtColor(face_image, cv2.COLOR_BGR2GRAY)
            cv2.imwrite("C:/DatasetFinal/Database/"+listdir+"/"+listdir2+"/"+str(i)+".jpg", face_image)

```

Figura 57. Script pre-procesamiento.

Las imágenes ya procesadas se han separado en un conjunto de datos de entrenamiento de 10.000 imágenes de las cuales la mitad tienen expresión feliz y la otra mitad expresión neutra, se utiliza la misma cantidad de imágenes de las dos expresiones a detectar para evitar que el algoritmo genere con mayor probabilidad una de estas dos expresiones. Las 1.929 imágenes restantes se reservarán como datos de prueba.

En las siguientes figuras se muestra un ejemplo de las imágenes finales del dataset, en la primera figura las imágenes pertenecen a la categoría “happy” ya que su expresión es feliz, las imágenes de la segunda figura pertenecen a la categoría “neutral” porque su expresión es neutra.

Figura 58. Ejemplos de imágenes de entrenamiento, clase "happy"

Figura 59. Ejemplo de imágenes de entrenamiento, clase "neutral"

4.3 Entrenamiento del modelo

Para realizar el entrenamiento del modelo hemos utilizado la herramienta Google Colab. Colab es un servicio cloud, basado en los Notebooks de Jupyter, que permite el uso gratuito de las GPUs y TPUs de Google, con bibliotecas como: Scikit-learn, PyTorch, TensorFlow, Keras y OpenCV. Todo ello con bajo Python 2.7 y 3.6

Todo el código elaborado para el entrenamiento y prueba del modelo se ha elaborado en un cuaderno de Jupyter que iremos ejecutando haciendo uso de Colab, en la configuración de cuaderno hemos habilitado la opción de aceleración por GPU, esta opción permite usar GPUs proporcionadas por Google, de modo que es una buena opción para cuando no se dispone de ella para estos fines ya que el tiempo de procesado reduce significantemente.

De aquí en adelante se detalla el contenido del archivo FaceEmotionPredictor.ipynb

▾ **EMOTION DETECTOR**

First of all, import all the libraries

```
1 import keras
2 from keras import layers
3 from keras.layers import Conv2D, MaxPooling2D, AveragePooling2D, Input
4 from keras.layers import SeparableConv2D,GlobalAveragePooling2D
5 from keras.layers import Activation, Dense, BatchNormalization
6 from keras.models import Sequential, Model, load_model
7 from keras.callbacks import EarlyStopping,ReduceLROnPlateau,ModelCheckpoint
8 from keras.preprocessing.image import ImageDataGenerator
9 from keras.utils.np_utils import to_categorical
10 from keras import backend as K
11
12 import matplotlib
13 import matplotlib.pyplot as plt
14 import numpy as np
15
16 from google.colab import drive
17
18 from sklearn.model_selection import train_test_split
```

Figura 60. FaceEmotionPredictor.ipynb fragmento 1

Primero de todo importamos las bibliotecas necesarias, usaremos keras para definir las capas de la arquitectura de la red, algunas funciones de scikit learn, matplotlib y numpy para las imágenes y Google Drive porque es donde está alojado el dataset.

```
detected_emotions = ['Neutral', 'Happy']
emotions_values = {'Neutral': 0, 'Happy': 1}
batch_size = 256
nb_epoch   = 100
validation_split = 0.2
```

Figura 61. FaceEmotionPredictor.ipynb fragmento 2

A continuación, definimos algunos parámetros que necesitaremos más adelante:

Detected_emotions: se corresponde con los dos títulos de las clases que queremos predecir.

Emotions_values: valor que se asigna a cada clase.

Batch_size: es el tamaño del lote de imágenes que la red procesará por cada fase de entrenamiento.

Nb_epoch: es el número de fases de entrenamiento que se realizarán.

Validation_split: Porcentaje de los datos de entrenamiento que serán usados para validación.

Mount directories from drive

```
drive.mount('/content/drive')
```

Define the path to train and test data in the drive directories mounted

```
[ ]  data_train_path = '/content/drive/My Drive/dataset/train'
     data_test_path = '/content/drive/My Drive/dataset/test'
```

Figura 62. FaceEmotionPredictor.ipynb fragmento 3

Montamos el directorio de drive y establecemos las rutas a los datos de entrenamiento y prueba.

Now we will collect the images and set the labels

```
import os
train_images=[]
labels=[]
test_images=[]
```

```
[ ]  for filename in os.listdir(data_train_path+'/happy'):
       image = plt.imread(data_train_path+'/happy/'+filename)
       train_images.append(image)
       labels.append(0)
```

```
[ ]  for filename2 in os.listdir(data_train_path+'/neutral'):
       image2 = plt.imread(data_train_path+'/neutral/'+filename2)
       train_images.append(image2)
       labels.append(1)
```

Figura 63. FaceEmotionPredictor.ipynb fragmento 4

Vamos guardando las imágenes en arrays y estableciendo la etiqueta correspondiente, que será cero para la clase happy y uno para la clase neutral.

Convert the created arrays of images and labels to numpy array

```
[ ]  1 Y_train=np.array(labels)
     2 X_train=np.array(train_images, dtype=np.uint8)
```

Here we se the classes we have

```
[ ]  1 classes=np.unique(Y_train)
     2 print(classes)
```

```
[0 1]
```

Figura 64. FaceEmotionPredictor.ipynb fragmento 5

Convertimos los arrays de imágenes a numpy array para poder trabajar con ellos y comprobamos que las clases que se han establecido son 0 y 1.

Use a sckit learn function to split the train data into train and "test" data, this test data is actually VALIDATION data

```
[ ]  1 train_X,test_X,train_Y,test_Y = train_test_split(X_train,Y_train,test_size=0.2)
```

Normalize data before training

```
[ ]  1 train_X = train_X.astype('float32')
     2 test_X = test_X.astype('float32')
     3 train_X = train_X / 255.
     4 test_X = test_X / 255.
     5 train_Y_OH = to_categorical(train_Y)
     6 test_Y_OH = to_categorical(test_Y)
```

Figura 65. FaceEmotionPredictor.ipynb fragmento 6

Utilizamos una función de scikit learn para dividir el conjunto de datos de entrenamiento en datos de entrenamiento y validación, después, normalizamos los datos antes de pasarlos a la red para el entrenamiento.

Set the input shape used, we will use it as parameter for the CNN layers

```
1 if K.image_data_format() == 'channels_first':
2     train_X = train_X.reshape(train_X.shape[0], 1, 48, 48)
3     input_shape = (1, 48, 48)
4 else:
5     train_X = train_X.reshape(train_X.shape[0], 48, 48, 1)
6     input_shape = (48, 48, 1)
```

Figura 66. FaceEmotionPredictor.ipynb fragmento 7

Fijamos el tamaño de los datos de entrada en la variable *input_shape* que utilizaremos posteriormente como parámetro para crear la arquitectura de la CNN.

```
def architecture(input_shape, n_classes):

  input= Input(shape=input_shape)
  x = Conv2D(8, (3, 3), strides=(2, 2),use_bias=False)(input)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  x = Conv2D(16, (3, 3), strides=(2, 2), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  residual_block = Activation('relu')(x)

  x = SeparableConv2D(16, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  x = SeparableConv2D(16, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = MaxPooling2D(pool_size=3, strides=2, padding='same')(x)
  residual_block= Conv2D(16, (1, 1), strides=(2, 2),use_bias=False)(residual_block)
  residual_block = BatchNormalization()(residual_block)

  x = layers.add([x, residual_block])

  x = Activation('relu')(x)
  x = SeparableConv2D(32, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  x = SeparableConv2D(32, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = MaxPooling2D(pool_size=3, strides=2, padding='same')(x)
  residual_block= Conv2D(32, (1, 1), strides=(2, 2),use_bias=False)(residual_block)
  residual_block = BatchNormalization()(residual_block)

  x = layers.add([x, residual_block])

  for _ in range(4):
    x = Activation('relu')(x)
    x = SeparableConv2D(32, (3, 3), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = SeparableConv2D(32, (3, 3), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = SeparableConv2D(32, (3, 3), padding='same', use_bias=False)(x)
    x = BatchNormalization()(x)
    residual_block = layers.add([x, residual_block])

  x = Activation('relu')(residual_block)
  x = SeparableConv2D(64, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  x = SeparableConv2D(128, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = MaxPooling2D(pool_size=3, strides=2, padding='same')(x)

  residual_block= Conv2D(128, (1, 1), strides=(2, 2),use_bias=False)(residual_block)
  residual_block = BatchNormalization()(residual_block)

  x = layers.add([x, residual_block])


  x = SeparableConv2D(192, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)
  x = SeparableConv2D(256, (3, 3), padding='same', use_bias=False)(x)
  x = BatchNormalization()(x)
  x = Activation('relu')(x)

  x = GlobalAveragePooling2D()(x)
  output = Dense(units=n_classes, activation='softmax')(x)

  model = Model(input, output)
  return model
```

Figura 67. FaceEmotionPredictor.ipynb fragmento 8

Aquí podemos ver la arquitectura en capas de la red que ya se describió anteriormente, *input_shape* se utiliza para crear la capa de entrada mientras que la capa densa final toma el número de clases (dos), que será el tamaño de salida.

Create the model, check the summary information and compile

```
[ ]  1 model = architecture(input_shape,2)
     2 model.summary()
     3 model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
```

Figura 68. FaceEmotionPredictor.ipynb fragmento 9

Ahora utilizamos la función que creamos para definir la arquitectura, comprobamos el resumen generado y compilamos el modelo.

```
Model: "functional_43"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_40 (InputLayer)        [(None, 48, 48, 1)]       0
_________________________________________________________________________________________________
conv2d_184 (Conv2D)          (None, 23, 23, 8)         72          input_40[0][0]
_________________________________________________________________________________________________
batch_normalization_603 (BatchN (None, 23, 23, 8)      32          conv2d_184[0][0]
_________________________________________________________________________________________________
activation_455 (Activation)  (None, 23, 23, 8)         0           batch_normalization_603[0][0]
_________________________________________________________________________________________________
conv2d_185 (Conv2D)          (None, 12, 12, 16)        1152        activation_455[0][0]
_________________________________________________________________________________________________
batch_normalization_604 (BatchN (None, 12, 12, 16)     64          conv2d_185[0][0]
_________________________________________________________________________________________________
separable_conv2d_452 (Separable (None, 12, 12, 16)     400         batch_normalization_604[0][0]
_________________________________________________________________________________________________
batch_normalization_605 (BatchN (None, 12, 12, 16)     64          separable_conv2d_452[0][0]
_________________________________________________________________________________________________
activation_457 (Activation)  (None, 12, 12, 16)        0           batch_normalization_605[0][0]
_________________________________________________________________________________________________
separable_conv2d_453 (Separable (None, 12, 12, 16)     400         activation_457[0][0]
```

Figura 69. FaceEmotionPredictor.ipynb fragmento 10

Esta es una pequeña parte del resumen generado que muestra las primeras capas, el resumen completo se incluye en los anexos.

We use will use ImageDataGenerator later for data augmentation

```
[ ]  1 data_generator = ImageDataGenerator(
     2                         featurewise_center=False,
     3                         featurewise_std_normalization=False,
     4                         rotation_range=10,
     5                         width_shift_range=0.1,
     6                         height_shift_range=0.1,
     7                         zoom_range=.1,
     8                         horizontal_flip=True)
```

Figura 70. FaceEmotionPredictor.ipynb fragmento 11

Esto se utilizará para el aumento de datos sobre los datos de entrenamiento, esto consiste en aplicar transformaciones simples a las imágenes para aumentar el conjunto de datos.

```
filepath='/content/drive/My Drive/dataset/weights/'+str(batch_size)+'-{epoch}-{val_accuracy}.hdf5'
checkpoint = ModelCheckpoint(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max')

callbacks_list = [checkpoint]

print ('Start training....')

history = model.fit_generator(data_generator.flow(train_X, train_Y_OH, batch_size=batch_size),
                    steps_per_epoch=len(train_X) / batch_size,
                    epochs= nb_epoch,
                    verbose=1,
                    callbacks = callbacks_list,
                    validation_data= (test_X, test_Y_OH))
```

Figura 71. FaceEmotionPredictor.ipynb fragmento 12

Antes de empezar el entrenamiento se establece la ruta donde se guardará el modelo y se crea un *checkpoint*, esto hará que se guarden los datos cada vez que la precisión incrementa.

Para entrenar el modelo se ejecuta la función *fit_generator* de keras que está preparada para utilizar el generador de datos definido anteriormente y de los datos resultantes tomar un bloque de tamaño *batch_size* para el entrenamiento. Se harán tantos pasos de entrenamiento como se hayan definido en la variable *batch_size.*

El conjunto de validación no se utiliza para entrenar la red, sino que se usa para medir la precisión del modelo sobre estos datos, lo que se muestra como *val_accuracy* y que es la medida que tomamos para decidir si nuestro modelo ha mejorado o no.

```
Epoch 97/100
32/31 [==============================] - ETA: 0s - loss: 0.2054 - accuracy: 0.9118
Epoch 00097: val_accuracy improved from 0.91700 to 0.92500, saving model to /content/drive/My Drive/dataset/weights/256-97-0.925000011920929.hdf5
32/31 [==============================] - 3s 86ms/step - loss: 0.2054 - accuracy: 0.9118 - val_loss: 0.1953 - val_accuracy: 0.9250
Epoch 98/100
32/31 [==============================] - ETA: 0s - loss: 0.2013 - accuracy: 0.9130
Epoch 00098: val_accuracy did not improve from 0.92500
32/31 [==============================] - 3s 87ms/step - loss: 0.2013 - accuracy: 0.9130 - val_loss: 0.2052 - val_accuracy: 0.9180
Epoch 99/100
32/31 [==============================] - ETA: 0s - loss: 0.2010 - accuracy: 0.9137
Epoch 00099: val_accuracy did not improve from 0.92500
32/31 [==============================] - 3s 87ms/step - loss: 0.2010 - accuracy: 0.9137 - val_loss: 0.2022 - val_accuracy: 0.9150
Epoch 100/100
32/31 [==============================] - ETA: 0s - loss: 0.1938 - accuracy: 0.9172
Epoch 00100: val_accuracy did not improve from 0.92500
32/31 [==============================] - 3s 87ms/step - loss: 0.1938 - accuracy: 0.9172 - val_loss: 0.2161 - val_accuracy: 0.9175
```

Figura 72. Salida del entrenamiento

Una vez finalizado el entrenamiento podemos ver que la salida muestra que la precisión obtenida sobre los datos de validación (*val_accuracy* en el paso 97) es de un 92’5% .

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

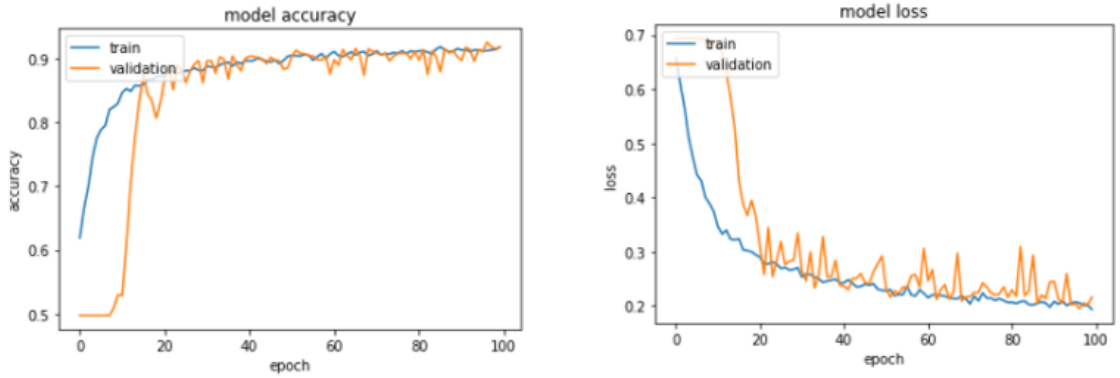


Figura 73. Gráficos de precisión y pérdidas del modelo

Se ha representado la variación de las medidas de precisión y pérdidas obtenidas durante el entrenamiento en dos gráficos que comparan también la precisión y pérdidas evaluadas en los conjuntos de entrenamiento y validación.

4.4 Pruebas y resultados

Para las pruebas se utiliza el modelo entrenado en el apartado anterior y el conjunto de datos de prueba reservados que ya habíamos procesado previamente, aquí se muestra el código de prueba y los resultados obtenidos.

```
[34]  import matplotlib.image as mpimg
      from matplotlib.pyplot import imshow
      import cv2
      import numpy as np
      !pip install face_recognition
      import face_recognition
```

```
%matplotlib inline
testim = mpimg.imread('/content/drive/My Drive/dataset/test/neutral/5032.jpg')
imshow(testim, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7f81424882e8>
```

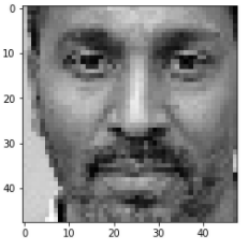


Figura 74. Código de prueba fragmento 1

Primero importamos las bibliotecas necesarias y cargamos una imagen y la mostramos.

```
[73]  1 face_image = np.asarray(testim)
      2 face_image = np.reshape(face_image, [1, face_image.shape[0], face_image.shape[1],1])
```

```
[74]  1 face_image = face_image.astype('float32')
      2 face_image = face_image / 255.
      3 predicted_class = model.predict(face_image)
      4 print(predicted_class)

      [[0.00461372 0.9953863 ]]
```

Figura 75. Código de prueba fragmento 2

Convertimos la imagen a numpy array, redimensionamos y normalizamos. Después hacemos la predicción con el método *predict*. El resultado son dos valores entre 0 y 1 que indican la correspondencia a cada clase, lo que se traduce en porcentajes, de manera que la predicción final es 0.461372% HAPPY frente a un 99.5863% NEUTRAL. Podemos comprobar en la imagen mostrada que la expresión era neutral y por tanto la predicción es acertada.

Aquí tenemos algunos otros ejemplos de predicciones en las que podemos ver que las predicciones se ajustan bastante bien a la expresión que las personas percibiríamos.

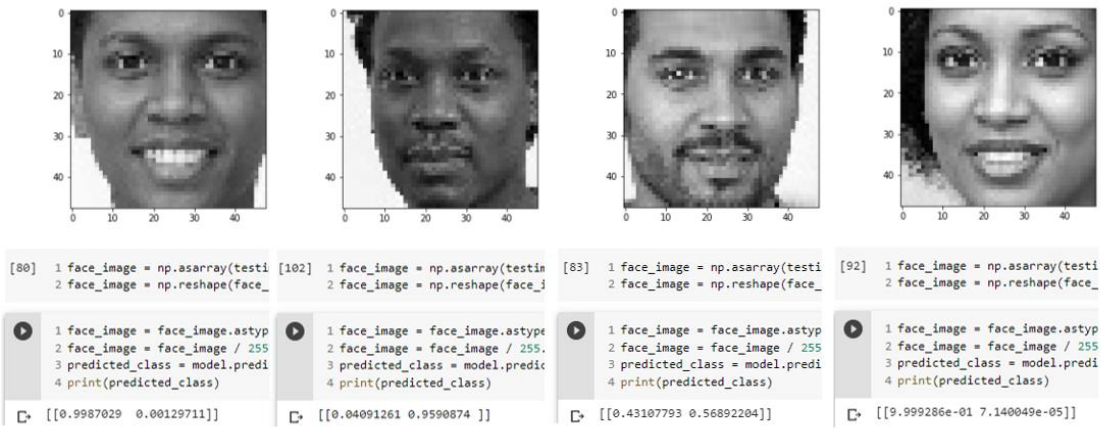


Figura 76. Predicciones realizadas con el modelo.

En la primera imagen se predice con un 99.87% de probabilidad que la expresión es feliz, mientras que en la segunda imagen vemos un 95.90% de probabilidad que la expresión es neutra.

En un caso más complicado de predecir como podría ser la tercera imagen en la que no hay una sonrisa muy clara vemos que el modelo toma predicciones bastante acertadas, es capaz de predecir que la expresión se encuentra casi en un punto medio entre neutra y feliz dando como resultado unas probabilidades de 43.10% feliz frente a 56.89% neutra.

En la última imagen se aprecia que hay ocasiones en las que las predicciones pueden llegar a probabilidades muy elevadas como en este caso 99.99% de probabilidad de que la expresión sea feliz.

Se ha desarrollado también un código que permite dada una imagen dibujar en ella el recuadro que delimita el rostro con la etiqueta de la clase correspondiente.

```
import cv2
import keras
from keras.models import load_model
import face_recognition
from face_recognition import face_locations
import matplotlib
import numpy as np
from PIL import Image

model = load_model("C:/database/model.hdf5")
image = cv2.imread("C:/database/test_image.jpg")

face_image = face_recognition.face_locations(image)
top, right, bottom, left = face_image[0]
face_image = image[top:bottom, left:right]
face_image = cv2.resize(face_image, (48,48))
face_image = cv2.cvtColor(face_image, cv2.COLOR_BGR2GRAY)
face_image = np.asarray(face_image)
face_image = np.reshape(face_image, [1, face_image.shape[0], face_image.shape[1],1])
face_image = face_image.astype('float32')
face_image = face_image / 255.
predicted_class = np.argmax(model.predict(face_image))
if (predicted_class==0):
    emotion="HAPPY"
    color = (0,0,255)
else:
    emotion = "NEUTRAL"
    color = (255,0,0)
cv2.rectangle(image, (left, top), (right, bottom), color, 2)
cv2.rectangle(image, (left, bottom - 35), (right, bottom), color, cv2.FILLED)
font = cv2.FONT_HERSHEY_DUPLEX
cv2.putText(image, emotion, (left + 6, bottom - 6), font, 1.0, (255, 255, 255), 1)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
img = Image.fromarray(image)
img.show()
```

Figura 77. Dibujar recuadro con emoción predicha DetectEmotion.py

El proceso es prácticamente el mismo, abrimos la imagen y detectamos el rostro que recortamos, redimensionamos, pasamos a blanco y negro y normalizamos para que pueda ser utilizado por el modelo.

Generamos la predicción, pero esta vez utilizamos *np.argmax()* sobre la predicción, esto nos devolverá el valor correspondiente a la clase que tiene un mayor porcentaje, en función de si el valor devuelto es 0 o 1 sabremos cual es la etiqueta que debe aparecer.

Sobre la imagen original dibujamos un rectángulo donde se encuentra el rostro, ubicamos también un rectángulo más pequeño relleno sobre el que pondremos el texto que indica la expresión predicha.

Estas son un par de imágenes donde se muestra el resultado de la ejecución del código anterior.

Figura 78. Resultados DetectEmotion.py

Como podemos ver se dibuja correctamente el recuadro que encuadra el rostro mostrando para la clase predicha la etiqueta correspondiente en un color diferente para que sea más visual.

Capítulo 5. Conclusiones

Tras el desarrollo y pruebas de los dos casos prácticos de este proyecto se obtienen las siguientes conclusiones:

- En lo referente a la trazabilidad de objetos, hemos visto que podemos utilizar diferentes técnicas o modelos en función del uso que deseemos darle. La aplicación de seguimiento ha resultado no ser muy precisa debido a la combinación de modelos que priman la velocidad ante la precisión, a pesar de ello nos permite tener una buena visión general del espacio que se está analizando. Esto podría ser útil, por ejemplo, para analizar el tráfico en carreteras, para controlar el aforo en lugares concurridos, etc. Se concluye también que, modelos ya existentes, podrían llegar a aplicarse en situaciones diferentes de las situaciones para las que han sido entrenados mediante su combinación con técnicas matemáticas. Además, los principales modelos de detección de objetos permiten ser reentrenados para conjuntos de entrenamiento propios, lo que nos permite trabajar con clases diferentes a las aprendidas por el modelo.

- En cuanto la detección de expresión en rostro, obtuvimos buenos resultados debido a la robustez de la arquitectura de red y la cantidad de datos de entrenamiento proporcionados para cada clase. La ampliación de este modelo o de la base de datos recopilada podrían usarse para modelos de detección de mayor cantidad de expresiones, disminución del sesgo en modelos de reconocimiento facial, etc. Se concluye además que, ante un problema complejo, la combinación de varios modelos especializados en tareas más concretas permite realizar de manera más metódica y precisa cada una de estas tareas, obteniendo mejores resultados en el conjunto final.

Este proyecto tomaba como base el aprendizaje supervisado, que es solamente una parte de las que componen la inteligencia artificial. Hemos comprobado que el aprendizaje supervisado nos abre un gran mundo de posibilidades, siempre y cuando, dispongamos de los datos de entrenamiento necesarios. A veces obtener grandes cantidades de datos para una tarea muy concreta podría resultar complicado, pero en estos casos, todavía disponemos de opciones de entrenamiento diferentes dentro de la inteligencia artificial. Esto nos lleva a que las aplicaciones y problemas que podrían resolverse con el uso de inteligencia artificial son innumerables y es un campo que seguirá en constante desarrollo en los años próximos.

Cada vez más, las necesidades computacionales se ven reducidas mediante el uso de herramientas proporcionadas por grandes empresas, como Google Colab, que se ha utilizado en este proyecto como muestra de ello. Esto permite que el desarrollo de aplicaciones que utilizan inteligencia artificial no se limite únicamente a quien disponga de grandes equipos, sino que cualquier persona con un ordenador y conexión a internet puede explorar y hacer uso de esta innovadora tecnología.

Bibliografía

[1] L. Vladimirov. 'Training Custom Object Detector - TensorFlow Object Detection API tutorial documentation'. 2020. https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#creating-tensorflow-records

[2] K. Patel. 'Custom Object Detection using TensorFlow from Scratch'. 2019. https://towardsdatascience.com/custom-object-detection-using-tensorflow-from-scratch-e61da2e10087

[3] J. Brownlee. 'A Gentle Introduction to Object Recognition With Deep Learning'. 2019. https://machinelearningmastery.com/object-recognition-with-deep-learning/

[4] A. Geitgey. 'Machine Learning is Fun! Part 4: Modern Face Recognition with Deep Learning'. 2016. https://medium.com/@ageitgey/machine-learning-is-fun-80ea3ec3c471

[5] C. García Moreno. '¿Qué es el Deep Learning y para qué sirve?'. 2016. https://www.indracompany.com/es/blogneo/deep-learning-sirve

[6] 'What is Deep Learning?' https://deepai.org/machine-learning-glossary-and-terms/deep-learning

[7] Redacción APD. '¿Qué es Machine Learning y cómo funciona?'. 2019. https://www.apd.es/que-es-machine-learning/#:~:text=Machine%20Learning%20o%20Aprendizaje%20autom%C3%A1tico,de%20datos%20en%20su%20sistema.

[8] D. Santos, L. Dallos, P. A. Gaona-García. 'Algoritmos de rastreo de movimiento utilizando técnicas de inteligencia artificial y machine learning'. 2020. https://scielo.conicyt.cl/scielo.php?pid=S0718-07642020000300023&script=sci_arttext

[9] R. Gandhi. 'R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms'. 2018. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[10] M. Hollemans. 'Real-time object detection with YOLO'. 2017. https://machinethink.net/blog/object-detection-with-yolo/

[11] A. Amidi, S. Amidi. 'Convolutional Neural Networks cheatsheet'. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

[12] 'Rich feature hierarchies for accurate object detection and semantic segmentation'. 2014. https://arxiv.org/abs/1311.2524

[13] 'Fast-RCNN'. 2015. https://arxiv.org/abs/1504.08083

[14] 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks'. 2016. https://arxiv.org/abs/1506.01497

[15] 'Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition'. 2015. https://arxiv.org/abs/1406.4729

[16] 'You Only Look Once: Unified, Real-Time Object Detection'. 2016. https://arxiv.org/abs/1506.02640

[17] 'SSD: Single Shot MultiBox Detector'. 2016. https://arxiv.org/abs/1512.02325

[18] '¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador'. 2018. https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

[19] 'Object Tracking in Deep Learning'. https://missinglink.ai/guides/computer-vision/object-tracking-deep-learning/

[20] 'Simple Online and Realtime Tracking'. 2017. https://arxiv.org/abs/1602.00763

[21] 'Simple Online and Realtime Tracking with a Deep Association Metric'. 2017. https://arxiv.org/abs/1703.07402

[22] ‘Multi-Object Tracking with Siamese Track-RCNN’. 2020. https://arxiv.org/abs/2004.07786
[23] ‘Tracking without bells and whistles’. 2019. https://arxiv.org/abs/1903.05625
[24] S. R. Maiya. ‘DeepSORT: Deep Learning to Track Custom Objects in a Video’. 2019. https://nanonets.com/blog/object-tracking-deepsort/
[25] D. Tomaszuk. ‘GUI_Blob_Tracker’. 2019. https://github.com/dghy/GUI_Blob_Tracker
[26] N. Wojke. ‘Deep SORT’ 2019. https://github.com/nwojke/deep_sort
[27] R. Kanjee. ‘DeepSORT — Deep Learning applied to Object Tracking’. 2020. https://medium.com/@riteshkanjee/deepsort-deep-learning-applied-to-object-tracking-924f59f99104
[28] R. Prabhu. ‘Understanding of Convolutional Neural Network (CNN) — Deep Learning’. 2018. https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148
[29] I. Ramos García. ‘Facial emotion recognition using Deep Learning techniques and Google Colab’. 2020. https://medium.com/swlh/facial-emotion-recognition-using-deep-learning-techniques-and-google-colab-4098798845bb
[30] ‘Very Deep Convolutional Networks for Large-Scale Image Recognition’. 2015. https://arxiv.org/abs/1409.1556
[31] ‘Deep Residual Learning for Image Recognition’. 2015. https://arxiv.org/abs/1512.03385
[32] ‘Going Deeper with Convolutions’. 2014. https://arxiv.org/abs/1409.4842
[33] ‘Xception: Deep Learning with Depthwise Separable Convolutions’. 2017. https://arxiv.org/abs/1610.02357
[34] V. Roman. ‘Most Popular Convolutional Neural Networks Architectures’. 2020. https://towardsdatascience.com/convolutional-neural-networks-most-common-architectures-6a2b5d22479d
[35] S. Sahoo. ‘Residual blocks — Building blocks of ResNet’. 2018. https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec#:~:text=In%20fact%2C%20if%20you%20look,as%20identity%20shortcut%20connections%20too.&text=It%20has%20also%20been%20observed,rather%20than%20only%20the%20input.
[36] B. Raj. ‘A Simple Guide to the Versions of the Inception Network’. 2018. https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202
[37] S. Tsang. ‘Review: Xception — With Depthwise Separable Convolution, Better Than Inception-v3 (Image Classification)’. 2018. https://towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568
[38] J. Hui. ‘Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3)’. 2018. https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359
[39] T. Simonite. ‘The Best Algorithms Struggle to Recognize Black Faces Equally’. 2019. https://www.wired.com/story/best-algorithms-struggle-recognize-black-faces-equally/
[40] T. Simonite. ‘A Health Care Algorithm Offered Less Care to Black Patients’. 2019. https://www.wired.com/story/how-algorithm-favored-whites-over-blacks-health-care/
[41] W. Douglas. ‘Predictive policing algorithms are racist. They need to be dismantled’. 2020. https://www.technologyreview.com/2020/07/17/1005396/predictive-policing-algorithms-racist-dismantled-machine-learning-bias-criminal-justice/
[42] A. Hern ‘Google's solution to accidental algorithmic racism: ban gorillas’. 2018. https://www.theguardian.com/technology/2018/jan/12/google-racism-ban-gorilla-black-people