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

Detección de armas en vídeos mediante técnicas de Deep Learning

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Christian Gutiérrez Lancho

Director: Mikel Galar Idoate

Pamplona, 10/6/2019

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen

Este TFG se centra en el estudio de técnicas de *Deep Learning* con el objetivo de crear un sistema detector de objetos capaz de detectar armas de forma automática en imágenes y vídeos. Para resolver este problema el cual pertenece al campo de la inteligencia artificial, se realizará un estudio centrado en las redes neuronales las cuales son la base para la creación de un sistema detector de objetos. El principal algoritmo que se estudiará y optimizará será Yolo, *"You only looks once",* además de otros en menor profundidad como el SSD. Con estos algoritmos un ordenador será capaz de generalizar la idea de arma, para que pueda reconocerla detectándolas de manera automática.

Palabras clave

-Deep learning

-Aprendizaje

-Redes neuronales

-Detección de Objetos

-Inteligencia Artificial

Índice

Contenido

Índice....3
1.Introducción....4
2.Preliminares....5
2.1. Introducción a Machine Learning....5
2.2.Introducción a redes neuronales y a *Deep Learning*....7
2.3.Convolutional Neuronal Network....9
2.4.Entrenamiento de redes neuronales....13
2.5 *Frameworks* o Herramentas utilizadas....16
3.Deep Learning para detección de Objetos....18
3.1 Detección de Objetos....18
3.2 Yolo (v2)....19
3.2.1 Introducción a Yolo....19
3.2.2 Arquitectura y Loss function....22
3.2.3 Beneficios y desventajas del algoritmo Yolo....24
3.2.4 Yolo Vs Yolo v2....25
3.3. SSD....29
3.3.1 Introducción....29
3.3.2 Arquitectura y Predicción de las BB....30
3.3.3 *Loss function*....32
3.3.4 Hard negative mining, data augmentation y resultado del SSD....33
3.4. Yolo(v2) Vs SSD....34
4.Detección de armas en imágenes y videos....36
4.1 BBDD....36
4.2 Optimización de Yolo v2....37
4.2.1 Primer acercamiento en la optimización de Yolo v2....38
4.2.2. Pruebas para la optimización de Yolo v2 y Data Augmentation....39
4.3 Optimización de SSD....45
4.3.1 Pruebas de optimización para SSD....45
4.4 Comparativa Yolo v2 Vs SSD (300-512)....50
5.Conclusiones y líneas futuras....56
Glosario Figuras....57
Bibliografía....60

1.Introducción

Hoy en día la inteligencia artificial, a pesar de ser un concepto que algunas personas lo asocian a un futuro lejano, está más presente en nuestras vidas que nunca. Desde anuncios personalizados en las páginas que visitamos, hasta coches capaces de circular sin la necesidad de un conductor. La Inteligencia Artificial ya está presente en nuestra vida diaria pero aún queda mucho por mejorar y explorar en este campo debido a las infinitas posibilidades que este ofrece.

Pero ¿Qué es la Inteligencia Artificial? Resumiéndolo de manera simple, es la inteligencia ejercida por máquinas. Las máquinas, en este caso los ordenadores, son increíblemente capaces a la hora de realizar cálculos, pueden hacer miles de operaciones en tiempos realmente cortos. En este aspecto son mucho más rápidas que los seres humanos, sin embargo, a la hora de generalizar o aprender un concepto nuevo, los seres humanos hasta ahora hemos sido superiores. Si una persona no ha visto un gato nunca, con ver unos cuantos, es capaz de generalizar la idea de gato, y aunque vea gatos de distintos colores, a distintas distancias o en distintas posiciones sigue siendo capaz de reconocer que lo que está viendo es un gato. Sin embargo, este razonamiento no es aplicable a un ordenador. Ahí es donde entramos en juego nosotros, los ordenadores tienen la potencia, pero nosotros debemos ser quien desarrolle métodos para que los ordenadores sean capaces de aprender.

Para ello se han desarrollado métodos y técnicas que, a partir de una gran cantidad de datos, el ordenador pueda obtener ciertas características claves que le permitan desarrollar su tarea, ya sea reconocer un objeto, clasificar una imagen u obtener ciertas conclusiones a partir de unos datos. Estos algoritmos que permiten a una máquina "aprender" se engloban dentro del *Machine Learning*, que trata de dotar a las máquinas de la capacidad de aprender. Uno de los subcampos del *Machine Learning* que más atención ha recibido recientemente, es el *Deep Learning,* que consiste en dotar de esa capacidad de aprendizaje simulando el funcionamiento del cerebro humano. Ambos conceptos se explican en el apartado 2.2.

Dentro de la inteligencia artificial hay un problema recurrente que se presenta en proyectos como el coche autónomo, que es la detección de objetos. El problema consiste en, a partir de una imagen, que la máquina sea capaz de detectar una serie de objetos con la mayor precisión posible. Este problema es en lo que se ha centrado este trabajo fin de grado.

Objetivos y Motivación

El objetivo de este trabajo fin de grado es el de desarrollar un sistema de detección de objetos para un objeto en particular, en este caso armas de fuego, centrándonos en pistolas. Es indiscutible que un tema de preocupación general hoy en día es la seguridad. Cuanto más avanza la tecnología más se nos facilita la vida, pero al mismo tiempo tenemos más medios para ejercer acciones mal intencionadas hacia los demás. Sin embargo, la inteligencia artificial puede ayudar en este aspecto, previendo o detectando rápidamente acciones que violen la seguridad de otras personas.

Un sistema de detección de armas automático en vídeos, podría ser muy beneficioso para la seguridad, ya que aplicándolo a cámaras de seguridad en aeropuertos, escuelas o en la vía pública permitiría detectar este tipo de armas de manera veloz, de forma más eficiente que un humano y así permitiría actuar de la forma más rápida posibles ante estas situaciones.

Este es el objetivo general, pero para alcanzarlo primero hemos de establecer ciertos objetivos particulares para llegar a dicha meta.

- Primero realizar un **estudio sobre el *Deep Learning***, enfocado en la detección de objetos, para familiarizarnos y entender los algoritmos que se utilizan para que la máquina sea capaz de aprender. Entender las bases desde un punto de vista teórico y práctico.
- Algo indispensable en este tipo de problemas son los datos que se van a utilizar para que el ordenador sea capaz de aprender, por tanto, hay que buscar en nuestro caso, una **base de datos con fotos de armas** y con sus respectivos documentos que etiquetan la posición de cada una de las armas que aparecen en las fotos**.**
- Una vez que tenemos ciertos conocimientos sobre los algoritmos que permiten detectar objetos y los datos necesarios para ejercer nuestra tarea, queda la **aplicación de *Deep Learning* para la detección de armas con YOLO y SSD**, dos algoritmos de detección de objetos.

Estos fueron los pasos, en rasgos generales, que se llevaron a cabo para alcanzar el objetivo propuesto y en los siguientes puntos se profundizará más en cada uno de ellos.

2.Preliminares

2.1. Introducción a Machine Learning

El *Machine Learning* es un subcampo dentro de la Inteligencia Artificial (IA) que se centra en el aprendizaje de máquinas. Aprender en este contexto se refiere a identificar patrones complejos en millones de datos. El que aprende en realidad es el algoritmo que se ejecuta en dicha máquina el cual se encarga de tratar los datos y de obtener información de estos.

Gracias a la información extraída el sistema aprende de forma autónoma a tomar las decisiones. De forma práctica esto se traduce en una función en la que a partir de una **entrada** se obtiene una **salida**, con lo que el problema radica en construir esta función matemática de forma automática, a la que se le suele denominar modelo. La diferencia entre la IA y *Machine Learning*, por lo tanto, radica en que un programa muy inteligente que tiene un comportamiento similar al humano podría considerarse como IA, pero a menos que sus parámetros se aprendan automáticamente a partir de los datos, no es *Machine Learning*.

Con forme avanza el tiempo la cantidad de información que manejamos cada vez es mayor y debido a esta gran cantidad de datos el *Machine Learning* tiene un rol esencial, ya que puede producir modelos de manera rápida y automática que puedan analizar datos más grandes y complejos y producir resultados más rápidos y precisos – incluso en una escala muy grande.

Y debido a lo que ofrece, no es de extrañar que el *Machine Learning* esté presente en el gobierno, servicios financieros, atención a la salud, marketing, transporte, búsqueda de nuevas energías etc. Es decir, en cualquier campo en el cual se posea mucha información y extraer conclusiones de estos datos sea esencial para desempeñar cierta tarea.

Por poner unos ejemplos el *Machine Learning* ha sido un pilar fundamental para la recomendación personalizada de anuncios, el coche autónomo, saber lo que usuarios de twitter piensan sobre cierto producto o para la detección de fraudes.

Dentro del *Machine Learning* podemos diferenciar tres grandes tipos:

- **Aprendizaje Supervisado**: El Aprendizaje Supervisado se caracteriza porque se entrena proporcionándole muchos ejemplos y a partir de esto, pueda generalizar y aplicar las ideas a prendidas de los ejemplos proporcionados, que están etiquetados, para generalizar sobre ejemplos nuevos, no etiquetados. Dentro de esta rama podemos diferenciar dos tipos de problemas, los de **clasificación** y los de **regresión**. Un ejemplo de problema de clasificación sería tener imágenes de perros y gatos y querer que nuestro sistema aprenda a clasificar correctamente las imágenes. Para ello se le proporcionaría al sistema muchas imágenes de perros y gatos etiquetadas, indicando si la imagen es de perro o gato, y una vez entrenado se le pasarían imágenes nuevas sin marcar, para ver como de bien realiza la clasificación.

  En cuanto al problema de regresión este consistiría en predecir un valor real en vez de una clase o valor discreto como en clasificación. Un ejemplo sería a partir de cierta información sobre un piso, como el número de habitaciones y metros cuadrados, determinar el precio final. Para ello, se le pasaría al sistema información de pisos, indicando para cada uno de ellos todas las características mencionadas anteriormente, y su precio (la etiqueta). Una vez entrenado el sistema con dicha información, para comprobar su eficacia le pasaríamos pisos con los metros cuadrados y el número de habitaciones, pero sin el precio final, para que el sistema lo predijera.

  El problema que nos atañe en este trabajo es el de detectar armas, y si lo enfocamos de forma más general, sería el de detectar objetos. La detección de objetos combina ambos problemas del Aprendizaje Supervisado, por un lado, clasificamos el objeto detectado en una clase, ya que podemos intentar detectar varios objetos en una imagen, lo que es un problema de clasificación, y por otro queremos devolver un "caja", al fin y al cabo, dos puntos en la imagen, que se ajusten a la posición y escala del objeto.

- **Aprendizaje no Supervisado**: En el aprendizaje no supervisado los ejemplos que el sistema utiliza para aprender no están etiquetados previamente. El Aprendizaje no Supervisado tiene como objetivo aprender y abstraer los patrones de información contenidos en los datos aportados al sistema. Un problema típico de aprendizaje no supervisado es el *clustering*, donde se trata de crear grupos de datos similares de manera automática.

- **Aprendizaje Por refuerzo:** Aquí el sistema aprende a través de la experiencia. Si el sistema toma una mala decisión se le "castiga" pero en caso contrario se le "premia", es una técnica de prueba y error. Esta técnica es muy prometedora ya que no requiere muchos datos, los cuales muchas veces son difíciles de conseguir.

En el siguiente punto cambiaremos del *Machine Leraning* al *Deep Learning* y hablaremos sobre las estructuras sobre las cuales se apoya el *Deep Learning*, las redes neuronales.

2.2.Introducción a redes neuronales y a *Deep Learning*

El *Deep Learning* es una subárea del *Machine Learning*, es decir, existen técnicas de *Machine Learning* que no tiene por qué ser de *Deep Learning*. El *Deep Learning* lleva a cabo el proceso de *Machine Learning* usando una **red neuronal artificial**. Una red neuronal artificial al fin y al cabo es una asociación de neuronas distribuidas por capas, en la que las capas se comunican entre sí. En la capa inicial la red aprende algo simple y luego envía esta información al siguiente nivel. El siguiente nivel toma esta información sencilla, la combina, compone una información algo o un poco más compleja, y se lo pasa al tercer nivel, y así sucesivamente. Un esquema general de una red neuronal artificial estaría formado por neuronas cada una de ellas en una capa que pasa información a las neuronas de la capa siguiente como se puede observar en la Figura 1.

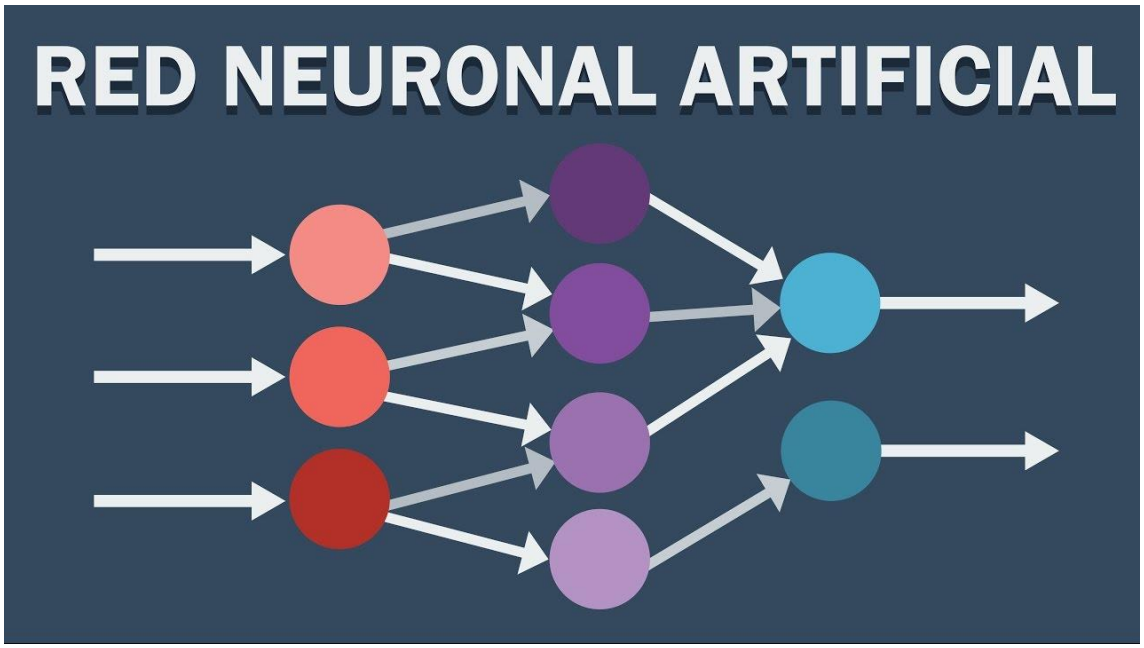


Figura 1, esquema general de una red neuronal , Fuente [1]

Hemos mencionado que las redes neuronales están formadas por neuronas, pero ¿Qué es una neurona? Las neuronas, en una red neuronal artificial, intentan modelar el comportamiento de una neurona natural y es la unidad esencial de las redes neuronales. La función que realiza una neurona es recibir un conjunto de entradas y devolver una salida. La neurona realiza una suma ponderada de las entradas, se le aplica una función de activación y devuelve una salida. Vamos a apoyarnos en la Figura 2 para analizar los distintos parámetros que intervienen.

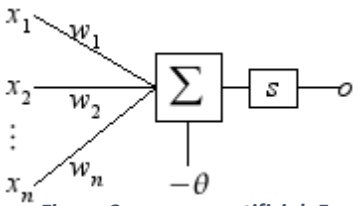


Figura 2, neurona artificial, Fuente [2]

La salida de la neurona se expresa como $s(red)$ donde s es la función de activación y red sería la suma ponderada de las entradas que tendría la siguiente forma en este ejemplo:

$$red = w_1 x_1 + \cdots + w_n x_n - \theta$$

Wi es el peso entre la i-ésima entrada y la neurona, theta es el valor de umbral, s la función de activación. Al final una neurona se podría modelar en un ordenador como una vector de pesos.

Una neurona simple solo puede trabajar con problemas linealmente separables, por tanto, para afrontar problemas más complejos creamos una red multicapa, una red con varias capas, agrupando varias neuronas por capa. En una red neuronal hay dos acciones indispensables para que esta sea capaz de aprender, que son la propagación hacia delante, *forward propagation*, y la propagación hacia atrás, *backward propagation*. Con la propagación hacia adelante proporcionamos un ejemplo a la red y esta lo propaga a lo largo de la red, desde el principio hasta el final devolviendo una salida. Una vez realizado el *forward propagation*, propagamos el error que hemos cometido, que se propaga gracias a la derivada de la función de coste, desde el final hacia el principio. Modificamos los pesos de las diferentes neuronas de cada capa y obtenemos así, en teoría, una mejora a la hora de predecir un resultado respecto al estado anterior.

Para explicar el funcionamiento de una red neuronal artificial vamos a partir de un ejemplo. Supongamos que queremos crear un programa que sea capaz de identificar coches en fotografías. Queremos, por tanto, que el ordenador sea capaz de generalizar la idea de que es un coche y para ello le pasamos muchas fotos de coches y aprendemos una red neuronal artificial.

El nivel inicial de una red de *Deep Learning* podría utilizar las diferencias entre las zonas claras y oscuras de una imagen para saber dónde están los bordes de la imagen. El nivel inicial pasa esta información al segundo nivel, que combina los bordes construyendo formas simples, como una línea diagonal o un ángulo recto. El tercer nivel combina las formas simples y obtiene objetos más complejos cómo óvalos o rectángulos. El siguiente nivel podría combinar los óvalos y rectángulos, formando las ruedas, puertas, el morro del vehículo etc. El proceso continúa hasta que se alcanza el nivel superior en la jerarquía, en el cual la red aprende a identificar coches. En la figura 3 se muestra este mismo proceso.

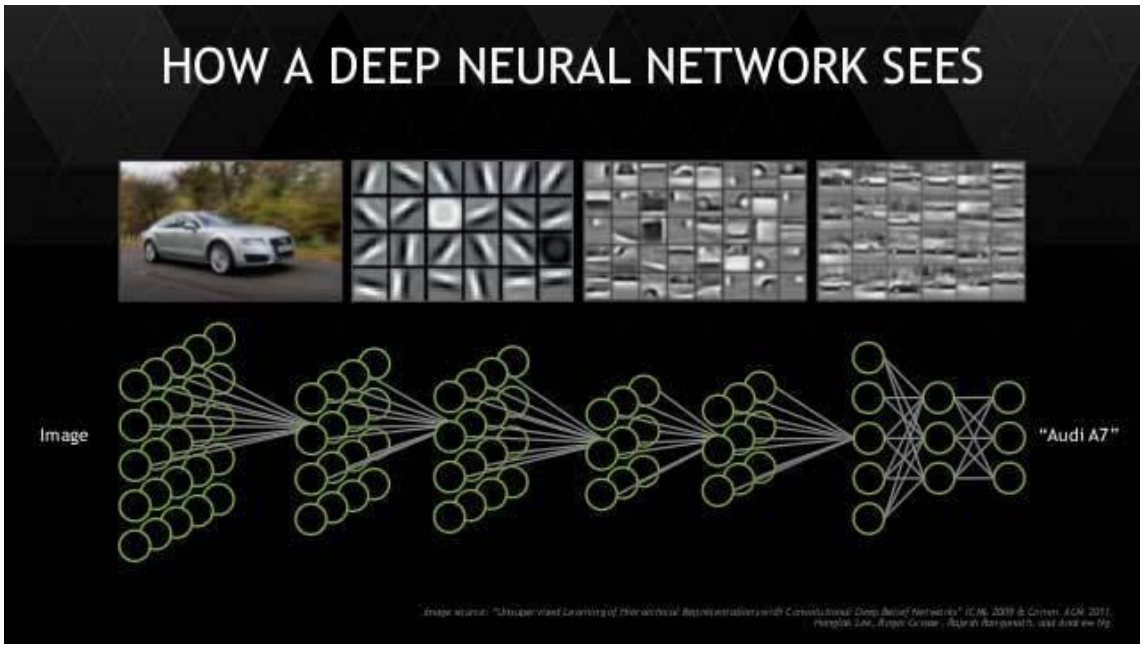


Figura 3, lo que aprende una red neuronal a lo largo de sus capas, Fuente [3]

El *Deep Learning* ha llamado mucho la atención principalmente por dos factores, por su potencial utilidad en distintos tipos de aplicaciones en el "mundo real" y debido a que obtiene tasas de éxito elevadas con entrenamiento supervisado. El *Deep Learning* se está utilizando para resolver problemas muy variados, unos pocos ejemplos serían:

- Reconocimiento de voz
- Identificar clientes potenciales
- Clasificación de imágenes y videos
- Localización de caras y emociones faciales
- Mejor comprensión de enfermedades, mutaciones de enfermedades y terapias genéticas.
- Orientación de anuncios y predicción de las preferencias de los clientes.

2.3.Convolutional Neuronal Network

La inteligencia artificial nos ha permitido avanzar considerablemente en muchos campos y uno de ellos es la visión artificial. La idea es que las máquinas vean el mundo como lo hacen los seres humanos, lo perciban de una manera similar e incluso utilicen este conocimiento para una multitud de tareas como el reconocimiento de imagen y video, análisis y clasificación de imágenes, recreación de medios, sistemas de recomendación, procesamiento del lenguaje natural, etc. Estos avances en la visión artificial no podrían haber sido posibles sin las redes neuronales convolucionales.

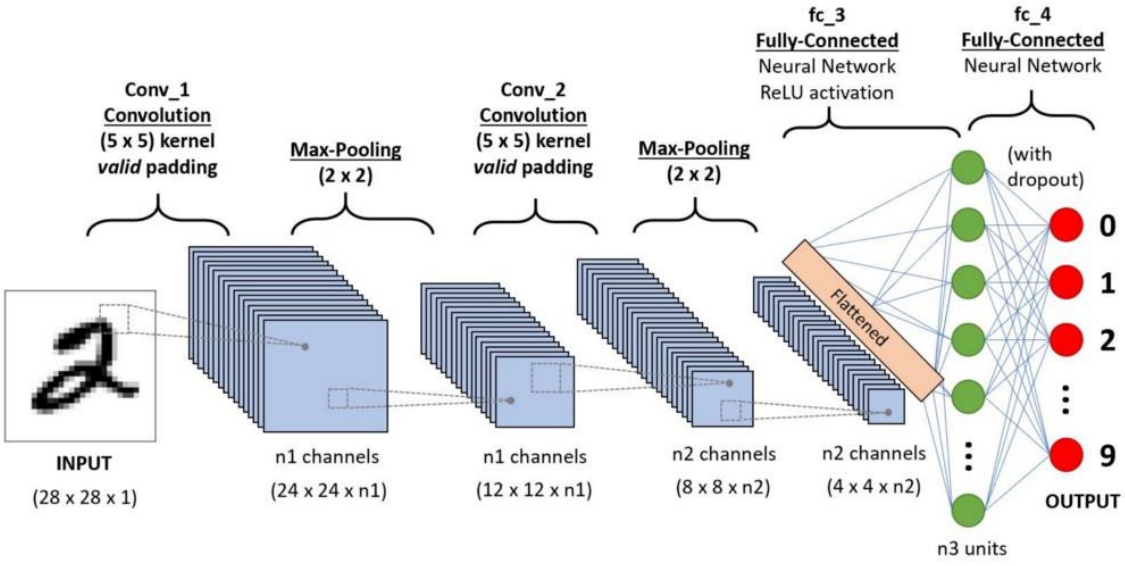


Figura 4, ejemplo de una CNN, Fuente [4]

Las redes convolucionales, a las cuales nos referiremos desde ahora como CNN, tienen muchas ventajas en comparación a una red neuronal convencional. En primer lugar, son capaces de captar características importantes de una imagen o vídeo a distintos niveles al igual que hace el cerebro humano. Otra de las ventajas y el principal motivo por el cual han adquirido peso dentro

del *Deep Learning* es la increíble reducción de parámetros que consigue en comparación a una red neuronal normal, gracias a que todas las zonas de la imagen comparten pesos. Vamos a utilizar un ejemplo para comprender la eficiencia de las CNN. Supongamos que una capa de nuestra CNN tiene 10 filtros 5x5. El número de parámetros que tendríamos sería 10*5*5, más los 10 bias harían un total de 260 parámetros. Ahora cojamos una red neuronal normal con 250 neuronas, aquí el número de parámetros por neurona depende del tamaño de la imagen, que si definimos K como el producto de la altura de la imagen por la anchura tendíamos 250*K parámetros, más los bias, que en este ejemplo si las imágenes fueran de 100*100 sería un total de 250*10000+1=2500001. Como se puede observar la diferencia entre una NN, red neuronal, y una CNN respecto al número de parámetros es considerable y la cantidad de memoria que se requiere para almacenar los parámetros en las CNN es mucho más reducida. El preprocesamiento requerido para las CNN es mucho más bajo en comparación con otros algoritmos, además mientras que otros métodos requieren más intervención humana las CNN con suficiente entrenamiento son capaces de aprender ciertas características de las imágenes.

Las CNN son extremadamente útiles ya que son capaces de captar las dependencias espaciales en una imagen a través de la aplicación de sucesivos filtros. La arquitectura realiza un buen entrenamiento respectos a los datos al mismo tiempo que reduce el número de parámetros involucrados mediante filtros. En la Figura 4 se puede ver un ejemplo de la arquitectura de una CNN.

Una imagen en color es una matriz formada por tres matrices, cada una representa la intensidad de un pixel para un color en concreto (rojo, verde o azul) , como se puede apreciar en la Figura 5. Por tanto, imaginemos la cantidad de información que podemos llegar a tener en una imagen en 8K (7680x4320).

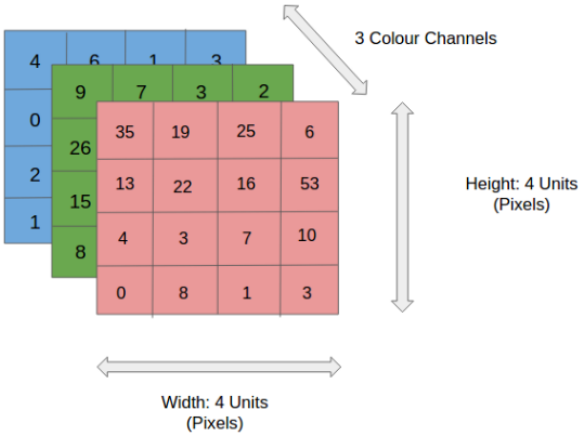


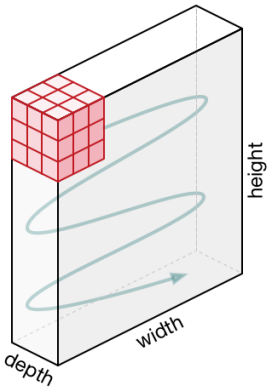
Figura 5, representación de una imagen en color, Fuente [5]

El papel que juegan las CNN es, reducir la forma de la imagen mediante las capas de *pooling* para que tengamos menos información que procesar, requiriendo así menos espacio y tiempo para realizar la tarea, mientras que las capas convolucionales se encargan de extraer características fundamentales de la imagen que son indispensables para obtener una buena predicción.

Así conseguimos una arquitectura que no es solo buena haciendo predicciones, sino que es capaz de trabajar con una cantidad masiva de datos en un tiempo aceptable.

En las Figuras 6 y 7 se puede observar el modo de aplicación de un filtro. El filtro es una matriz más pequeña que la matriz sobre la cual se va a aplicar, el número de pixeles que se desplaza en horizontal el filtro viene condicionado por el *stride*, que en el ejemplo de la Figura 7 es de 1. El filtro se aplica comenzando por la esquina superior izquierda, desplazándolo de izquierda a derecha hasta alcanzar la esquina inferior derecha. Si el filtro tiene un *stride* mayor que uno, y no se aplica un *padding* a la imagen sobre la cual se va a aplicar el filtro, esto provoca que la matriz resultante tenga una dimensión menor a la original.

Figura 6, ejemplo de aplicación de un filtro en una imagen en color, Fuente [6]



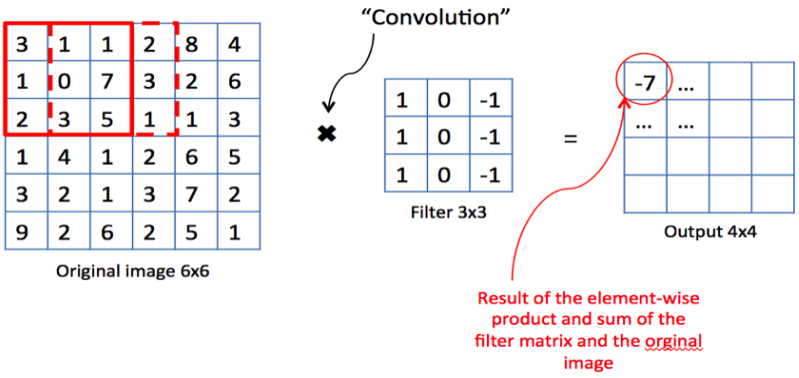


Figura 7, ejemplo práctico de cómo aplicar un filtro, Fuente [7]

El objetivo de estas operaciones convolucionales es el de extraer características de alto nivel como bordes, por ejemplo. No tiene porque solo haber una capa convolucional, normalmente, la primera capa convolucional se encarga de capturar las características de bajo nivel como lo bordes, la orientación del gradiente etc. Con más capas convolucionales la CNN aprende características de más alto nivel, formas más concretas, adquiriendo un conocimiento de las imágenes parecido al nuestro.

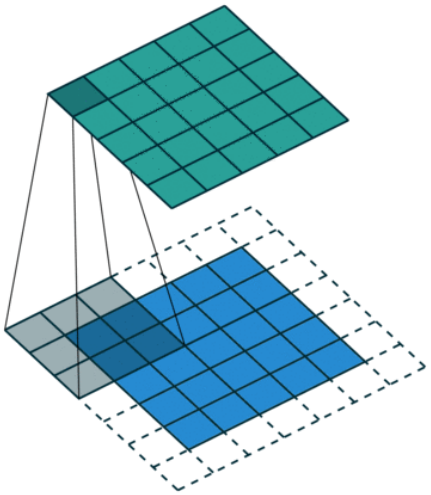
A la hora de realizar las operaciones convolucionales hay dos tipos de resultados, que la dimensión del resultado sea igual que la dimensión de la entrada, o que esta se haya reducido. En el primer caso se añaden filas y columnas a la imagen original para que a la hora de aplicar las operaciones convolucionales el resultado no tenga una dimensión menor, como se puede apreciar en la Figura 8, y en el segundo caso no se modifica la entrada y por tanto el resultado es menor que el original, respecto a las dimensiones.

Dentro de las CNN hay otro tipo de capas conocidas como las *Pooling layers*, cuya función es reducir el tamaño de la entrada, las cuales son muy útiles para obtener características independientes de la posición o la rotación, además de para reducir el tamaño de la imagen para las capas posteriores. Principalmente, se diferencian dos tipos de *Pooling layers*:

- *Max Pooling*, que devuelven el valor máximo de la porción de la imagen que cubre el filtro.
- *Average Pooling*, que devuelven la media de los valores que están dentro de la porción de la imagen que cubre el filtro.

Podemos observar un ejemplo de cada una en la Figura 9.

Figura 8, ejemplo de una imagen a la que se le añade un padding para que después de aplicar el filtro se mantenga la dimensión original, Fuente [8]



Dependiendo de las imágenes, el número de capas se puede aumentar para captar detalles de bajo nivel mejor a costa de potencia computacional.

Una vez que se ha reducido el tamaño de la imagen manteniendo hasta cierto punto las características fundamentales de la imagen original, hay que pasar esta información a una red neuronal convencional para la tarea de clasificación. Para ello convertimos nuestro resultado en un vector columna y se lo entregamos a la última capa, la *fully connected layer*, clasificando así el resultado. Se puede observar un esquema general en la Figura 10.

Figura 9, ejemplos de aplicación de pooling, Fuente [9]

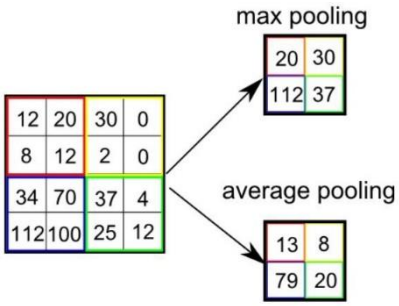
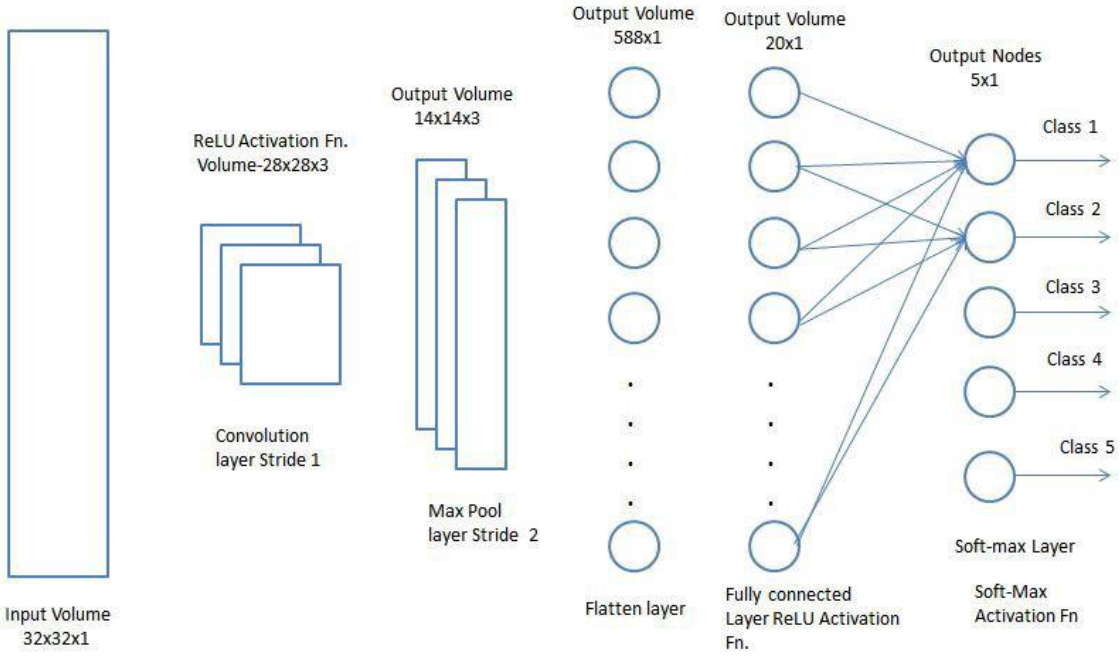


Figura 10, ejemplo de las distintas partes de una red convolucional, Fuente [10]



2.4.Entrenamiento de redes neuronales

Para conseguir el objetivo de ser capaces de detectar armas en videos necesitamos una cantidad grande de datos, fotos de armas etiquetadas, y una red neuronal capaz de usar esa información para aprender la forma que tiene un arma.

Para ello debemos entrenar la red con los datos y deberemos realizar varias pruebas ya que hay muchos parámetros que debemos tomar en cuenta a la hora de entrenar una red neuronal artificial. Mencionaré algunos de los parámetros más importante y métodos necesarios para el entrenamiento de la red:

1. ***Gradient Descent***: El problema que tenemos entre manos es el de crear un sistema capaz de localizar armas en fotos y vídeos, pero si lo simplificamos, tenemos un problema de minimización. Tenemos una función de coste, que depende del algoritmo y las analizaremos en mayor profundidad en siguientes apartados, que nos dice cómo lo estamos haciendo, y lo que queremos es minimizarla. Es decir, nuestro objetivo es **minimizar nuestra función de coste**, lo que significa que queremos encontrar aquellos valores mediante los cuales la función de coste alcanza su mínimo. Para ellos nos apoyamos en el descenso por gradiente. Debemos de modificar los parámetros para ir acercándonos al mínimo y para ello nos apoyamos en las matemáticas, la derivada de una función dJ/dw (donde J representa la función de coste y w los pesos) nos indica la "pendiente" de la función, lo que nos señala la dirección en la que debemos ir y como debemos modificar los valores para alcanzar el mínimo (ejemplo en la Figura 11).

   Una mediada a tener en cuenta a la hora de aplicar el descenso por gradiente es normalizar los parámetros de las diferentes características, para que estos se muevan en el mismo rango de valores y así que el descenso por gradiente sea más rápido en alcanzar el mínimo deseado. Esto se puede observar claramente en la Figura 13, en la cual hay dos gráficas, cada una con dos parámetros, en la primera cada uno tienen rangos diferentes y en la segunda no. Gracias a la normalización, el trayecto hacia el mínimo es más directo en el segundo caso.

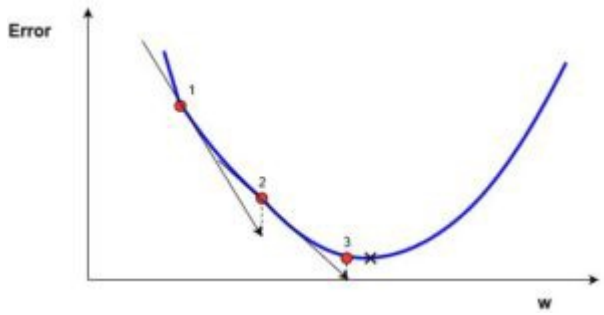


Figura 11, representación gráfica del descenso por gradiente, Fuente[11]

2. ***Batch size y Epoch*** : El *batch size* define el número de muestras que vamos a propagar hacia delante y hacia atrás en la red y una *Epoch*, época en español, es el número de veces que tomamos todos los datos de entrenamiento. Supongamos que tenemos 1000 imágenes y elegimos un *batch size* de 100, cogeremos 100 imágenes de las 1000

disponibles y entrenaremos toda la red aprendiendo y modificando los pesos, después cogeremos otras 100 imágenes y realizaremos el mismo proceso, cuando hayamos realizado esto 10 veces habremos recorrido todas las imágenes y por tanto llevaremos una época. Puede surgir la pregunta de por qué se divide el conjunto de entrenamiento en pequeños conjuntos en vez de entrenar la red neuronal con todas las muestras a la vez. Esto se debe a que en primera instancia es muy probable que no podamos almacenar en memoria todo el conjunto de entrenamiento, es decir por una limitación de hardware. En segundo lugar, llevaría una gran cantidad de tiempo actualizar el modelo al estar usando todos los ejemplos. Debido a esto se usan *mini-batches* para que los modelos se actualicen rápidamente, no se requiera cantidades de memoria elevadas y se siga manteniendo así modelos precisos y eficientes.

3. ***Learning rate***: Gracias a la derivada de la función de coste respecto a los pesos sabemos en qué dirección se encuentra el mínimo y por ende como modificar los parámetros, pero ¿Cuánto modificamos los parámetros?, aquí entra en juego el *Learning rate*. El *Learning* rate es un hiperparametro y no hay un valor que valga para todos los modelos, sino que hay que afinarlo para cada caso en concreto, aunque sí hay ciertos valores que limitan la búsqueda del valor óptimo de este. Por defecto se le suele dar el valor 0.001, cuanto mayor sea el *Learning rate* mayor serán los "saltos" que demos hacia el mínimo, pero hay que tener cuidado ya que, si el *Learning rate* es muy elevado, los saltos serán demasiado grandes y en vez de converger acabaremos divergiendo, como se puede observar en la Figura 12.

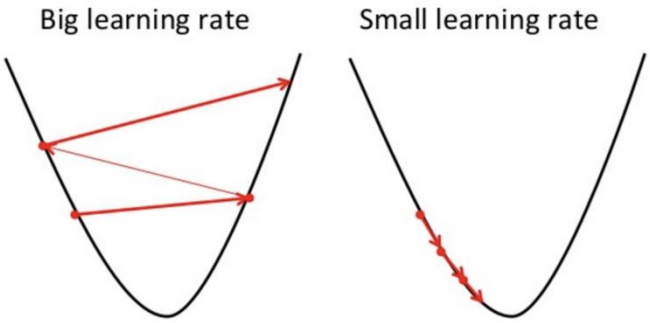


Figura 12, comparación entre diferentes valores de learning rate, Fuente [12]

4. ***Batch Normalitation***: El *batch normalization* no es un parámetro de la red como tal sino un método que consiste en normalizar la salida de una capa para reducir el rango entre el cual oscilan los valores de los pesos, restándole a cada salida la media del *batch* y dividiéndolo entre la desviación. Normalmente se normaliza la entrada de los datos a la red ya que si los valores de dos características tienen rangos muy distintos provocará que el aprendizaje sea más lento (como se ha mostrado en el punto anterior apoyándonos en la Figura número 12). La idea detrás del *Batch normalization* es, ya que normalizamos la entrada, ¿Por qué no normalizamos también las salidas de las capas de la red neuronal?

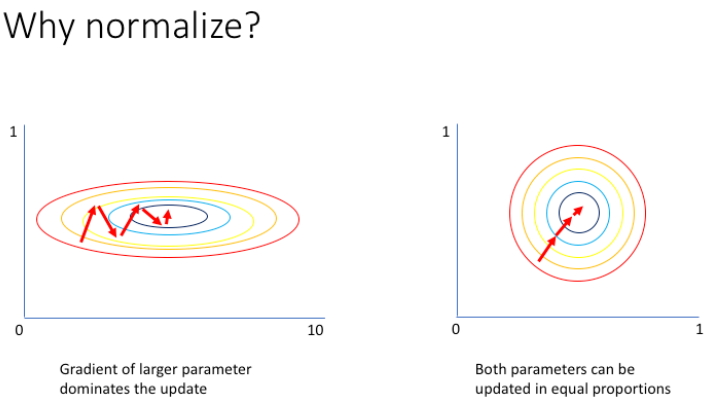


Figura 13, comparación entre dos modelos, uno sin normalizar y el otro si, Fuente [13]

El *Batch normalization* reduce el rango en el que los pesos varían, evitando que los pesos alcancen valores muy altos o bajos, y esto puede ser beneficioso, ya que permite a cada capa aprender de manera más independiente de las demás y así generalizar mejor. Gracias a que el rango de valores en el cual varían los pesos está controlado, nos permite usar *Learning rates* más grandes y reduce ligeramente el sobreentrenamiento.

5. **SGD y Adam:** El SGD, *Stochastic Gradient Descent*, consiste en aplicar el descenso por gradiente pero de 1 en 1 , es decir seleccionamos 1 ejemplo, aplicamos el forward y el *backward propagation*, actualizamos los parámetros y así con los ejemplos que nos queden. Es como aplicar el *mini-bacth gradient descent*, pero con *batch size* igual a 1. Una vez que tenemos la idea de lo que es el SGD, Adam es un algoritmo de optimización que se diferencia del SGD ya que este mantiene un *Learning rate* constante, que no cambia a lo largo del entrenamiento y para todos los parámetros igual. Sin embargo, Adam, no tiene un *Learning rate* fijo y se pude entender como una mezcla de SGD con *momentum y RMSprop*.

    a. El descenso por gradiente con momentum tiene en cuenta los gradientes anteriores para suavizar la actualización y alcanzar de forma más directa el mínimo deseado. Para ello vamos a apoyarnos en la Figura 14, empezamos en el punto A y después de una iteración de descenso por gradiente acabamos en el punto B, después de otra iteración nos moveríamos al punto C y así sucesivamente hasta el mínimo local. Como se pude observar al principio el desplazamiento vertical es elevado mientras que es desplazamiento horizontal es escaso, y en caso de aumentar el *Learning rate* este desplazamiento vertical se incrementaría. Sin embargo, si utilizamos una media exponencial ponderada de los valores dW y db, las oscilaciones en el eje vertical tienden a cero, lo que nos permite ir de forma más directa al punto deseado realizando menos iteraciones, como se puede observar en la Figura 15.

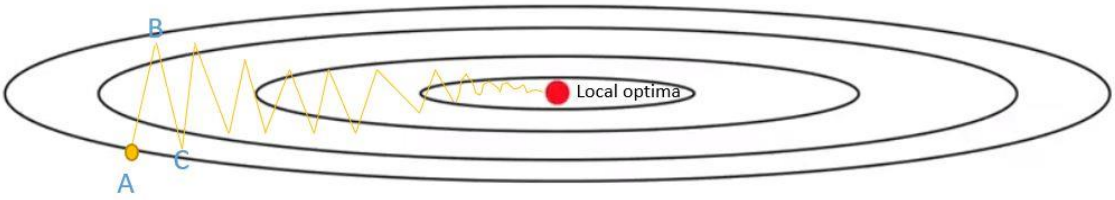


Figura 14, gradiente descent sin momentum, Fuente [14]

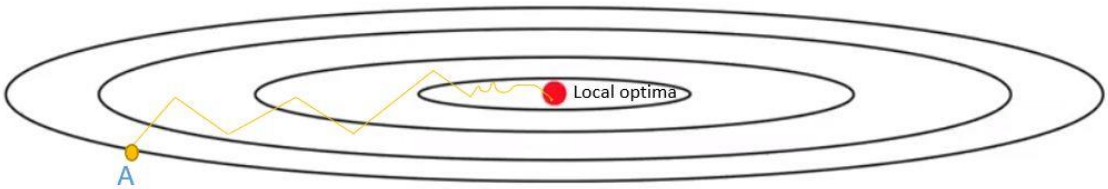


Figura 15, gradient descent con momentum, Fuente [15]

b. La idea que reside tras el algoritmo de optimización *RMSprop* es, que en vez de tener un *Learning rate* fijo tenemos un vector con un *Learning rate* por cada parámetro que va siendo actualizado basándonos en los valores obtenido en iteraciones anteriores reduciendo la oscilación como en el SGD con *momentum*.

Adam hace uso de estas 2 ideas dando muy buenos resultados, convirtiéndose en un algoritmo por defecto para utilizar en problemas de Deep *Learning*.

2.5 *Frameworks* o Herramentas utilizadas

Para programar un sistema de detección de armas se ha hecho uso de varias herramientas las cuales se exponen a continuación especificando brevemente en qué consisten y por qué se han elegido.

- **Python:** El lenguaje de programación mediante el cual se ha realizado el sistema, es Python. Python destaca por su facilidad de uso, legibilidad, portabilidad y simplicidad. Además, este lenguaje de programación suele ser el predilecto para realizar temas de inteligencia artificial debido a varios factores:
    - Una gran comunidad
    - Una mayor cantidad de bibliotecas si lo comparamos con otros lenguajes, hay una gran cantidad de recursos *open-source* para trabajar con temas de IA.
    - Los prototipos se pueden programar más rápido, al ser python un lenguaje dinámico y con un tipado débil, esto permite desarrollar a gran velocidad.
    - Muchos de los cursos de educación online, tutoriales o artículos para programar algoritmos dentro del campo de la IA están en Python.
- **Librerias de Python:** Se han utilizado muchas librerías de Python para desarrollar este proyecto, las más destacadas son:

- **Numpy:** Biblioteca de funciones matemáticas que facilita el trabajo con vectores y matrices.
- **Opencv:** Es una biblioteca libre de visión artificial la cual facilita enormemente el trabajo con imágenes y vídeos.
- **Keras:** Una de las librerías más potentes a la hora de desarrollar modelos de *Deep Learning* es keras, que es capaz de ejecutarse sobre *TensorFlow*. El objetivo de esta librería es el de facilitar el trabajo a la hora de manejar redes neuronales y ha sido esencial en este proyecto.

- **Coursera:** El *Deep Learning* ha cogido mucha fuerza en estos últimos años y es un campo relativamente "joven", sin embargo, ha evolucionado mucho y hay muchos conceptos e ideas que es necesario interiorizar antes de ponerse a programar un modelo de *Deep Learning.* Para obtener esa base teórica que permitiera tener un entendimiento de las bases, los algoritmos y las ideas detrás de estos, se ha utilizado la plataforma Coursera, la cual es una plataforma de educación virtual nacida en octubre de 2011 y desarrollada por académicos de la Universidad de Stanford con el fin de brindar oferta de educación masiva a la población. Dentro de Coursera hay una infinidad de cursos, pero aquel que tiene un interés relevante para este proyecto es el de *Deep Learning* (https://www.deeplearning.ai/). En este curso desde un punto de vista teórico y también práctico, se tratan temas como las redes neuronales, las CNN, acercamiento a problemas como detección de objetos etc. Una manera sencilla de adentrarse en el *Deep Learning* que permite adquirir una base para enfrentarse a algoritmos más complejos dentro de este campo.

- **Google Colab:** Entrenar un sistema suele ser costoso desde un punto de vista temporal y de espacio en memoria. Como se ha mencionado anteriormente las CNN nos permiten usar menos parámetros comparado con el número de parámetros que necesitaríamos en una NN, lo que provoca un ahorro en memoria, pero, aun así, la cantidad de memoria requerida suele ser elevada. Como entrenar un sistema puede llevar mucho tiempo, estos modelos se suelen ejecutar desde la GPU la cual es mucho más veloz que la CPU, pero la memoria que suele tener asociada es bastante limitada. Debido a esto un ordenador de sobremesa común es probable que no tenga los suficientes GB de memoria asociada a su tarjeta gráfica para entrenar ciertos modelos de *Deep Learning*. Aquí es donde Google Colab juega un papel muy importante. Google Colab es un servicio en la nube en el cual se puede ejecutar notebooks en Python, otros de los motivos por los cuales se eligió este lenguaje de programación, y que te proporciona una GPU con hasta 12GB de memoria de manera totalmente gratuita. Esto te permite desarrollar tu modelo en esta plataforma, la cual tiene una gran cantidad de librerías de Python instalada, y ejecutar tu código desde una GPU permitiéndote entrenar modelos a gran velocidad sin que se desborde la memoria. Los datos necesarios para entrenar el modelo se deben subir a Drive, un almacenamiento en la nube de hasta 15GB que Google te proporciona de manera gratuita. Una alternativa ideal ya que te permite trabajar con librerías de Python muy potentes sin perder tiempo en instalarlas, es totalmente gratis con una potencia de computación considerable y lo único necesario es una cuenta de Gmail y conexión a internet. Sin embargo, este entorno de ejecución tiene algunas pegas, cada 12h el entorno se reinicia y si pierdes la conexión puede afectar a la ejecución del programa.

- **Anaconda:** Anaconda es una distribución libre y abierta de los lenguajes Python y R, utilizada en ciencia de datos, y aprendizaje automático (*Machine Learning*). Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos. Está orientado a simplificar el despliegue y administración de los paquetes de software. Se ha utilizado para realizar pequeñas pruebas e iba a ser el principal entorno de desarrollo hasta que las limitaciones de memoria obligaron a trasladar el proyecto a Google Colab.

3.Deep Learning para detección de Objetos

3.1 Detección de Objetos

El problema de detección de objetos es uno de los grandes problemas dentro del *Deep Leraning*, ya que, a diferencia del problema de clasificación, no nos vale con saber si en una imagen hay un gato o un perro, por ejemplo, sino que también debemos localizarlo dentro de la foto. Para localizar el objeto que queremos dentro de una foto, nuestro algoritmo de detección de objetos devuelve una *Bounding box*, una caja, en la cual, dentro de ella está el elemento que buscamos. Al fin y al cabo, lo que nuestro algoritmo devuelve es o el punto central de la *Bounding box* junto a la anchura y altura de la caja, o el punto de la esquina superior izquierda y el de la esquina inferior derecha, junto a la clase a la que pertenece. Es necesario un parámetro que indique la clase a la cual pertenece la detección ya que podemos detectar varios tipos de objetos por

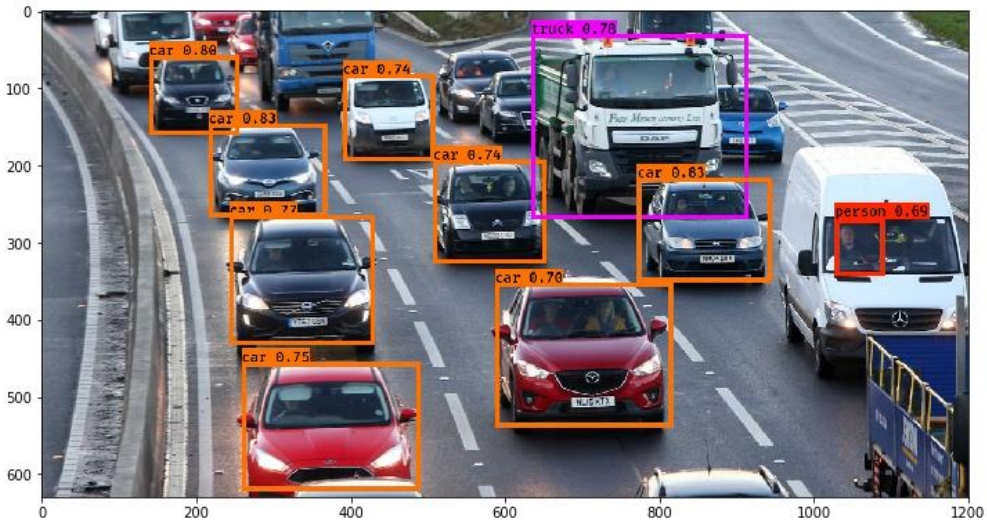


Figura 16, ejemplo de detección de objetos, Fuente [16]

imagen. Además, hay un parámetro que indica la "precisión" de la detección como se puede observar en la Figura 16.

Para que nuestro sistema sea capaz de detectar un objeto en concreto en un vídeo o imagen, en nuestro caso pistolas, es necesario proporcionarle muchas fotos en las que aparezca dicho objeto. Sin embargo, esto no es suficiente, ya que los ejemplo deben estar etiquetados, y a diferencia de un problema de clasificación que valdría con indicar si en la foto aparece o no una pistola, sería necesario indicar cuantas armas aparecen y cuál es la posición de cada una dentro de la imagen. Por tanto, se debe tener una base de datos con varias imágenes en las que

aparezca el susodicho objeto y a su vez por cada imagen un documento con la información esencial para la localización del objeto.

Pero, una vez entrenado nuestro sistema necesitamos saber que tal lo hace para ver si ha generalizado correctamente la idea de pistola, por tanto, necesitamos además de un conjunto de entrenamiento, el cual será utilizado por la CNN para entrenar, un conjunto de Test, etiquetado, para comprobar la eficiencia de nuestro sistema.

Una vez que sabemos cuáles son los datos necesarios y cuál es nuestra salida, explicaremos los dos algoritmos de detección de objetos que se han estudiado e implementado en este trabajo para conseguir un sistema de detección de pistolas.

3.2 Yolo (v2)

El algoritmo Yolo, cuyo nombre son las siglas de "*You only looks once*", es un algoritmo de detección de objetos. Es un algoritmo que proporciona buenos resultados en comparación con otros algoritmos con el mismo fin. Ahora se va a explicar su funcionamiento paso a paso ya que este ha sido uno de los algoritmos que se ha implementado para la detección de pistolas.

3.2.1 Introducción a Yolo

Figura 17, foto de ejemplo, Fuente [17]

En primer lugar, tomemos una imagen como ejemplo para facilitar la explicación, en este caso partimos de la imagen de la Figura 17. Yolo divide la imagen en SxS celdas. Cada celda se encarga de predecir un objeto. Por ejemplo, como se puede ver en la Figura 18, en la que se ha dividido la imagen de la Figura 17 con un valor S=7, dividiendo la imagen en 7x7 celdas. La celda amarilla intenta predecir la posición de la persona, ya que el centro de este objeto recae sobre esa celda.

Cada celda predice un número fijo de *Bounding Box* (BB), en este ejemplo la celda amarilla predice dos BB para localizar a la persona, como se puede observar en la figura 19 aunque solo una de las dos sería la predicción con la que nos quedaríamos.

Por cada celda:

- Se predicen un número B de BB donde cada una tiene un valor asignado conocido como *box confidence score*, el factor de confianza de la caja.
- Detecta un solo objeto independientemente del número de BB.

- Yolo puede intentar localizar varios objetos por imagen, por tanto, por cada celda calcula C probabilidades, donde C es el número de clases. Cada probabilidad refleja la probabilidad de que habiendo un objeto, dicho objeto pertenezca a una clase.

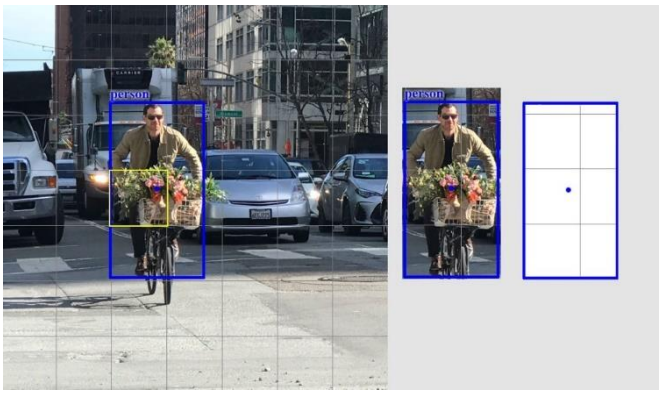


Figura 18, aplicación de Yolo sobre la foto ejemplo, Fuente [18]

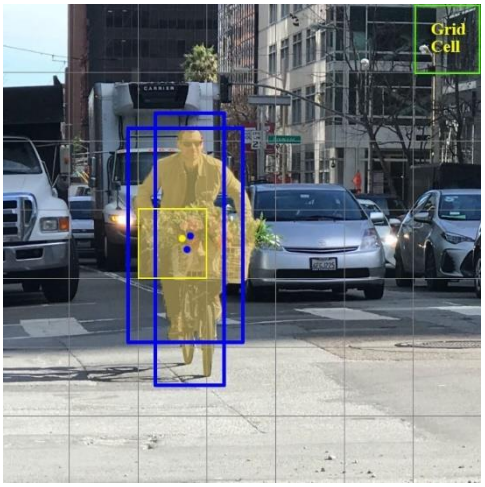


Figura 19, BB de la celda amarilla, Fuente [19]

Cada BB está formada por cinco valores $(x, y, w, h, factor\ de\ confianza)$. Cuatro de esos valores son necesarios para representar la posición de la BB en la imagen, x e y reflejan el centro de la caja mientras que h y w son la altura y la anchura. El quinto valor es el factor de confianza de la caja. El factor de confianza refleja como de probable es que la caja contenga a un objeto y cómo de precisa es esa predicción. Se normalizan los valores que reflejan la posición, la anchura y la altura de la BB respecto a la anchura y altura de la imagen y las coordenadas del centro de la BB respecto al tamaño de la celda, quedando todos los valores comprendidos entre 0 y 1. Cada celda tiene una *conditional class probability*, probabilidad de clase condicional, que refleja la probabilidad de que el objeto pertenezca a una determinada clase. Por tanto, el tamaño que tiene una predicción de este algoritmo sería:

$$(S, S, B * 5 + C)$$

Donde S es el tamaño de las celdas, B es el número de BB por celda, C el número de clases a predecir y el valor 5 viene de los 5 valores que hay asociados a cada BB.Yolo usa una CNN para reducir la dimensión espacial hasta obtener una matriz de dimensiones S*S*1024. Después Yolo aplica la regresión mediante dos capas *fully-connected*, obteniendo así S*S*B predicciones,

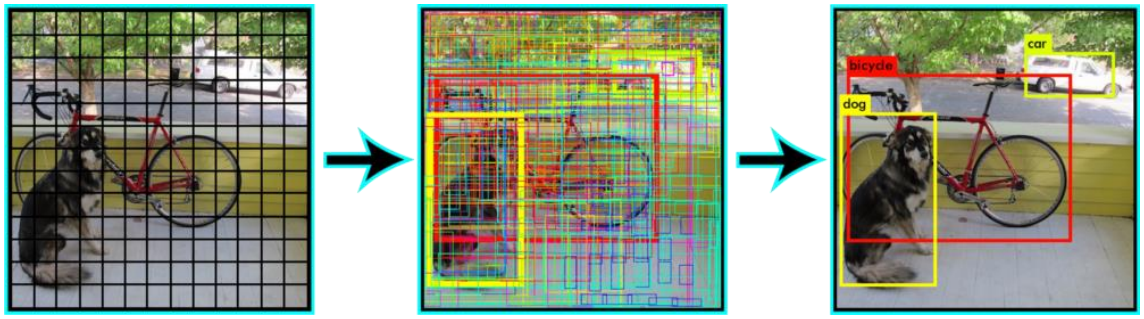


Figura 20, aplicación de Yolo sobre una imagen paso a paso, Figura [20]

donde B es el número de BB que se predice por celda (como se puede observar en la foto del medio de la Figura 20). Para obtener la predicción final aplicamos lo que se conoce como *Non-*

maximal suppression. Que no es más que ordenar las predicciones por su factor de confianza, de mayor a menor, y por cada predicción miramos si hay alguna predicción que tiene la misma clase y una IoU > 0.5, lo cual indicaría con una alta probabilidad que ambas predicciones detectan el mismo objeto. En caso de que la haya, ignoramos esa predicción, la que tiene menos confianza la cual se explica cómo calcularla a continuación, y repetimos el proceso con todas. Esto es lo que se aplica entre la imagen del centro y la de la derecha en la Figura 20.Otra medida que calculamos es la *class confidence* score, el factor de confianza de la clase, que se define como:

$$class\ confidence\ score = box\ confidence\ score * conditional\ class\ probability$$

Esta medida nos indica tanto la confianza de la clasificación, qué objeto es el que hemos detectado, y la confianza de la localización del objeto. Las fórmulas que se usan para calcular el el factor de confianza de la caja, la probabilidad de clase condicional y el factor de confianza de la clase se pueden ver en las Figura 21. Donde *ground Truth* es la posición donde está el objeto

$$\begin{aligned}
\text{box confidence score} &\equiv P_r(object) \cdot IoU \\
\text{conditional class probability} &\equiv P_r(class_i|object) \\
\text{class confidence score} &\equiv P_r(class_i) \cdot IoU \\
&= \text{box confidence score} \times \text{conditional class probability}
\end{aligned}$$

where

$P_r(object)$ is the probability the box contains an object.
IoU is the IoU (intersection over union) between the predicted box and the ground truth.
$P_r(class_i|object)$ is the probability the object belongs to $class_i$ given an object is presence.
$P_r(class_i)$ is the probability the object belongs to $class_i$

Figura 21, fórmulas para el calcular el class confidence score, Fuente [21]

en la imagen, el valor real. Estas fórmulas se utilizan a la hora de entrenar el sistema ya que conocemos la BB real y por tanto podemos calcular la IoU de la BB predicha sobre la original. Sin embargo, a la hora de aplicar este algoritmo sobre el conjunto de test, el cual no está etiquetado, la *class confidence score* se calcula como la mayor probabilidad de todas las clases por el valor que devuelve la red. Donde IoU es una medida conocida como la intersección sobre la unión, que consiste básicamente en dividir el área de la intersección de dos figuras sobre el área de la unión. En la figura 22 se puede ver un gráfico y ejemplo muy clarificador sobre esta medida.

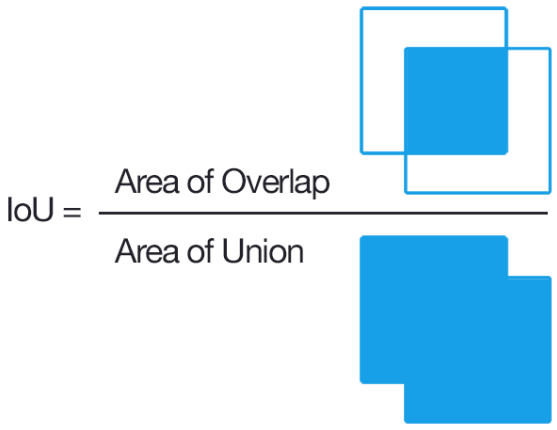


Figura 22, IoU, Fuente [22]

3.2.2 Arquitectura y Loss function

Respecto a la arquitectura de la red neuronal que utiliza Yolo está formada por 24 capas convolucionales seguida por dos capas *fully connected*. Algunas de las capas convolucionales son de 1x1 para reducir la profundidad. Se puede ver toda la estructura de la red neuronal de forma detallada en la Figura 23.

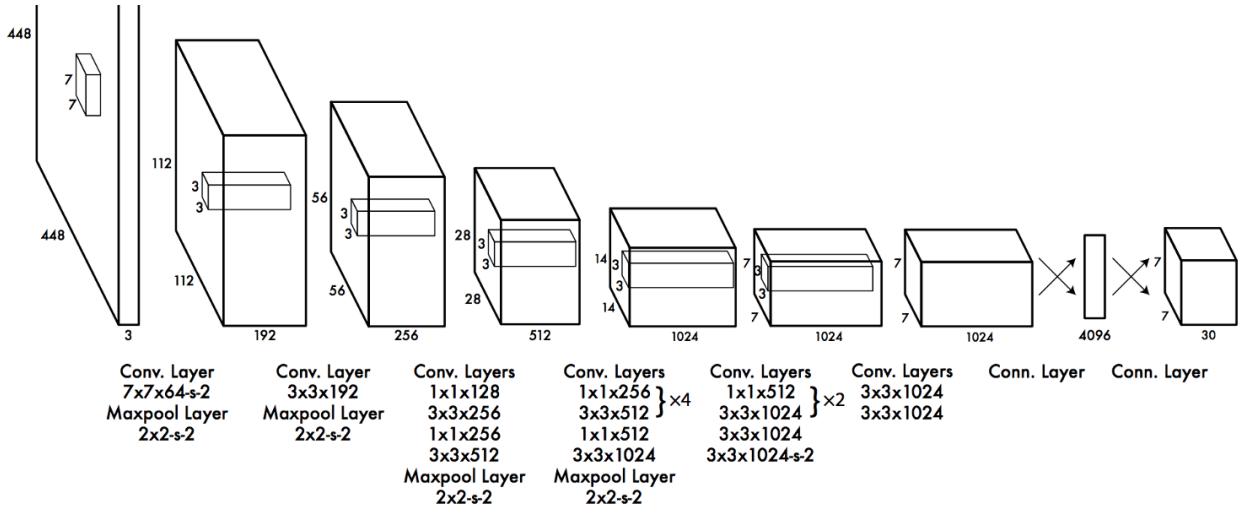


Figura 23, arquitectura Yolo, Fuente [23]

La arquitectura se pensó para utilizar el dataset Pascal VOC y los autores del algoritmo usaron los siguientes valores:

$$S = 7, B = 2, C = 20$$

Esto explica porque la salida tiene una dimensión de (7,7,30), donde 30 se obtiene como $(B * 5 + C)$. Lo cual tiene sentido, ya que al principio la imagen se "divide" en celdas, 7 a lo ancho y 7 a lo largo, lo que hace un total de 49 celdas. La salida nos da por cada celda 30 valores los cuales son 2 BB y la probabilidad de que el objeto pertenezca a cada una de las clases, las cuales son 20 en el Pascal VOC dataset.

Yolo predice múltiples BB por cada celda, a la hora de calcular el error en los true positives solo queremos que una de las BB sea responsable del objeto, por ello durante el entrenamiento seleccionamos aquella que posee las IoU más alta con la BB real. La función de coste se puede dividir en tres apartados según lo que se esté intentando medir en cada caso:

- ***Classification Loss*** **:** Mide el error al clasificar un objeto en una clase. Se calcula mediante la fórmula que se muestra en la Figura 24.

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

where

$\mathbb{1}_i^{obj} = 1$ if an object appears in cell i, otherwise 0.

$\hat{p}_i(c)$ denotes the conditional class probability for class c in cell i.

Figura 24, classification los, Fuente [24]

- ***Localization Loss*:** Mide el error cometido en la posición y tamaño de la BB predicha respecto a la original, solo tenemos en cuenta la BB con IoU máxima que representa la BB predicha como se ha mencionado anteriormente. Se calcula mediante la fórmula que se muestra en la Figura 25.

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$
$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

where

$\mathbb{1}_{ij}^{obj} = 1$ if the j th boundary box in cell i is responsible for detecting the object, otherwise 0.

λ_{coord} increase the weight for the loss in the boundary box coordinates.

Figura 25, localization los, Fuente [25]

No queremos ponderar de la misma manera el error en BB grandes que el error en BB pequeñas, ya que equivocarse 2 pixeles en la predicción de una BB grande es más permisible que equivocarse 2 pixeles en la predicción de una BB pequeña. Por ello usamos la raíz cuadrada de la altura y la anchura, en vez de usar directamente la anchura y la altura. Además, para poner más énfasis en la precisión de las BB multiplicamos el error por λ_{coord}, que por defecto tiene el valor de 5.

- **Confidence loss:** Mide el error cometido al decir que una BB contiene un objeto, siempre y cuando un objeto sea detectado en la BB. Se calcula mediante la fórmula de la Figura 26 , en caso de que la BB no detecte ningún objeto el *confidence loss* se calcularía mediante la fórmula de la Figura 27.

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

where

$\hat{C}_i$ is the box confidence score of the box j in cell i.

$\mathbb{1}_{ij}^{obj} = 1$ if the j th boundary box in cell i is responsible for detecting the object, otherwise 0.

Figura 26, confidence los cuando la BB detecta objeto, Fuente [26]

$$\lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2$$

where

$\mathbb{1}_{ij}^{noobj}$ is the complement of $\mathbb{1}_{ij}^{obj}$.

$\hat{C}_i$ is the box confidence score of the box j in cell i.

λ_{noobj} weights down the loss when detecting background.

Figura 27, confidence los cunado la BB no detecta objeto, Fuente [27]

La mayoría de las cajas no contienen objetos, esto causa un problema de desequilibrio de clases ya que entrenamos el modelo para que detecte más a menudo el fondo que los objetos que nos interesan. Para remediar esto rebajamos el peso de esta función multiplicándola por λ_{noobj}, por defecto 0.5.

Por tanto, la función de coste general se obtendría uniendo las tres funciones mencionadas anteriormente. La función que calcula el error a la hora de clasificar el objeto en una clase, la función que calcula el error en la localización del objeto y la función que calcula el error de que en la BB haya un objeto. En la Figura 28 se puede observar la función de coste final.

$$\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2
\end{aligned}$$

Figura 28, función de coste general del algoritmo Yolo, Fuente [28]

Yolo puede predecir diferentes BB para el mismo objeto, para evitar esto se aplica la *non-maximal suppresion* para eliminar aquellas BB con una confianza menor.

3.2.3 Beneficios y desventajas del algoritmo Yolo

En cuanto a las ventajas que posee este algoritmo destacan:

- Rápido, obtiene buenos resultados el tiempo real.

- Las predicciones las realiza una sola red neuronal, se puede entrenar de principio a fin para mejorar la precisión.
- Yolo es más general, supera a otros métodos cuando hay que generalizar partiendo de imágenes reales a otras áreas como el arte.
- Aquellos métodos basados en regiones están limitados a la región seleccionada, sin embargo, Yolo accede a toda la imagen para realizar predicciones.
- Yolo detecta un objeto por celda, esto fuerza a la diversidad espacial a la hora de realizar predicciones.

Sin embargo, como una celda detecta solo un objeto, esto limita lo cerca que los objetos pueden llegar a estar, ya que si la proximidad es elevada Yolo no será capaz de detectar todos los objetos.

3.2.4 Yolo Vs Yolo v2

Uno de los grandes competidores del algoritmo Yolo es el algoritmo SSD el cual obtiene mayor precisión en tiempo real. Yolo v2 es la segunda versión de Yolo, cuyo objetivo es aumentar la precisión y mejorar la velocidad del algoritmo. Antes de hablar sobre las mejoras que añade esta segunda versión se explicará la medida utilizada para medir como de bueno es nuestro modelo. La medida que se suele utilizar con los algoritmos de detección de objetos suele ser la mean *Average Precison* (mAP). El cálculo de la mAP se basa en la precisión, que señala cómo de preciso es nuestro modelo, y el *recall* que indica cómo de bueno es nuestro modelo encontrado todos los positivos. Las fórmulas para calcular el *recall* y la precisión se encuentran en la Figura 29.

$$Precision = \frac{TP}{TP + FP} \qquad TP = \text{True positive}$$

$$TN = \text{True negative}$$

$$Recall = \frac{TP}{TP + FN} \qquad FP = \text{False positive}$$

$$FN = \text{False negative}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Figura 29, formulas del recall, precisión y F1 score, Fuente [29]

Una vez que sabemos qué es y cómo se calcula el *recall*, la precisión y la IoU (la cual se ha explicado en el punto 3.2.1) podemos calcular la AP. Para ello dividimos el intervalo entre 0 y 1 en intervalos más pequeño, de 0.1 en 0.1 por ejemplo. Después, ordenamos todas las predicciones de nuestro modelo de forma descendente por el *confidence score*. Por cada ejemplo que recibimos indicamos si la predicción es correcta o no, una predicción es correcta si la IoU con la BB original es mayor a 0.5, calculamos la precisión y el *recall* hasta ese ejemplo y así sucesivamente con todos, se puede ver un ejemplo de este cálculo en la tabla de la Figura 30. Una vez hecho esto por cada valor que tengamos en nuestro intervalo entre 0 y 1, buscamos aquel valor máximo en la precisión que tenga un valor igual o mayor en el *recall* respecto al valor seleccionado del intervalo [0-1], se puede observar cómo realizar este proceso de manera gráfica en la Figura 31, la cual está relacionada con la Figura 30. Una vez obtenido este valor por

cada intervalo, realizamos la media y esto sería el AP, ya que solo tenemos una clase. En caso de tener varias, se calcularía el AP de cada clase y se haría la media, obteniendo así el mAP.

Rank	Correct?	Precision	Recall
1	True	1.0	0.2
2	True	1.0	0.4
3	False	0.67	0.4
4	False	0.5	0.4
5	False	0.4	0.4
6	True	0.5	0.6
7	True	0.57	0.8
8	False	0.5	0.8
9	False	0.44	0.8
10	True	0.5	1.0

Figura 30, tabla con valores de ejemplo ordenamos de manera descendente por confidence score, Fuente [30]

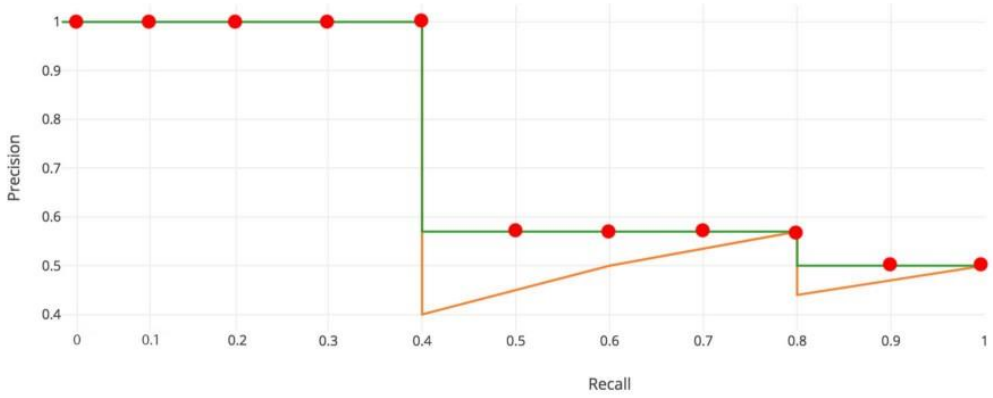


Figura 31, representación gráfica del calculo de la AP respecto a los valores de la Figura 30, Fuente [31]

Las principales mejoras que se añaden son:

- **Batch Normalization** en las capas convolucionales, esto elimina la necesidad de aplicar *dropout*, desactivar algunas neuronas de manera aleatoria con el objetivo de evitar el sobreentrenamiento. El sobreentrenamiento es ajustarse demasiado a los datos de entrada siendo incapaz de generalizar lo aprendido a otros conjuntos de entrada. Esto mejora la mAP en un 2%.
- **Clasificador de alta resolución:** El entrenamiento del Yolo está formado por 2 fases, primero entrenamos una red de clasificación como la VGG16. Después reemplazamos las capas *fully connected* por capas convolucionales y la reentrenamos para la detección de objetos. Yolo utiliza imágenes de 224x224 para el clasificador e imágenes 448x448 para la detección de objetos. Yolo v2 utiliza imágenes 224x224 para el clasificador, pero lo reentrena con imágenes 448*448 usando menos *epochs*, obteniendo una mejoro en el mAP del 4%.

- **Anchor boxes:** Una de las principales mejoras que presenta el Yolo v2 respecto al Yolo es el uso de anchor boxes. Yolo realiza predicciones arbitrarias para las BB. Las predicciones podían ser correctas para algunos objetos, pero no ajustarse del todo bien para otros. Aquí es donde los anchor box juegan un papel importante. La idea que hay detrás de estos es que hay objetos que tienen formas similares, así que si tenemos una base aproximada de la forma general de algunos objetos las predicciones serán mucho más precisas. Por ejemplo, los coches suelen tener BB más anchas que altas y las personas al revés. Por tanto, se obtiene un valor fijo de anchor boxes sobre el conjunto, mediante algún **algoritmo de *clustering*** como el K-means. Básicamente lo que hacemos es agrupar las BB en grupos similares, mediante la IoU, y obtener una representación general de cada grupo. Si el número de anchor boxes que generamos es 5, el número de predicciones sería de 5 elementos para cada BB (x, y, h, w, factor de confianza), y el vector de probabilidades de que ese objeto pertenezca a una clase, tomando el valor de 20 objetos como ejemplos anteriores, daría un total de 5*(20+5) =125 elementos por celda.

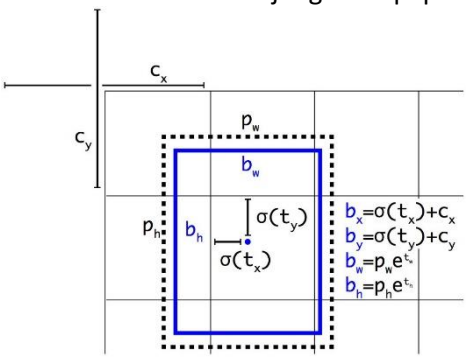


Figura 32, BB final respecto al anchor box, Fuente [32]

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$
$$Pr(\text{object}) * IOU(b, \text{object}) = \sigma(t_o)$$

where

t_x, t_y, t_w, t_h are predictions made by YOLO.

c_x, c_y is the top left corner of the grid cell of the anchor.

p_w, p_h are the width and height of the anchor.

c_x, c_y, p_w, p_h are normalized by the image width and height.

b_x, b_y, b_w, b_h are the predicted boundary box.

$\sigma(t_o)$ is the box confidence score.

Figura 33, fórmulas para calcular la BB a partir de los anchors, Fuente [33]

Esto obliga a realizar pequeños cambios en la arquitectura de la red como la eliminación de las capas *fully connected* responsables de la predicción de la BB, añadir alguna capa convolucional y modificar la dimensión de alguna capa. Ahora las **predicciones que hacemos se realizan en base a los anchors,** calculándose como se puede ver en la Figura 33. Además, en la Figura 32 se puede ver de forma gráfica como se realiza la predicción final de la BB, donde la caja azul es la BB predicha y la caja punteada el anchor.

- Las capas convolucionales reducen gradualmente la dimensión espacial, como la resolución decrece es más difícil detectar objetos pequeños. Para ello Yolo v2 cambia la capa 26*26*512 por una capa de 13*13*2048 y la concatena con la original de 13*13*1024 dando lugar a una capa de 13*13*3072 para realizar predicciones.
- Después de eliminar las capas *fully connected* Yolo v2 puede tomar fotos de distintos tamaños siempre y cuando estas sean múltiplos de 32. Durante el entrenamiento, cada 10 *batches*, Yolo v2 selecciona un valor aleatorio comprendido entre 320 a 608 múltiplo de 32 para entrenar el modelo y así forzar a la red a entrenar con tamaños y escalas de imagen diferentes.

Aplicando todas las mejoras mencionadas en este punto obtenemos una mejora considerable en la mAP, como se puede observar en la Figura 34.

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓
anchor boxes?				✓	✓			
new network?					✓	✓	✓	✓
dimension priors?						✓	✓	✓
location prediction?						✓	✓	✓
passthrough?							✓	✓
multi-scale?								✓
hi-res detector?								✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figura 34, comparativa entre el Yolo y el Yolo v2 al añadir las mejoras mencionadas, Fuente [34]

3.3. SSD

Gran parte del tiempo del proyecto se enfocó en el estudio y ajuste del algoritmo Yolo v2 sobre el dataset de armas. Después de varias pruebas y experimentos que se comentarán más adelante, se pensó que sería interesante probar otra arquitectura para ver si con el mismo conjunto de entrenamiento y de test, otro algoritmo de detección de objetos obtendría mejores resultados. Ese algoritmo es el SSD, cuyas siglas provienen de "*Single Shot MultiBox Detector*" (https://www.cs.unc.edu/~wliu/papers/ssd.pdf).

3.3.1 Introducción

Se ha mencionado en el anterior párrafo de donde vienen las siglas SSD, pero ¿De dónde viene ese nombre? La primera parte, "*Single Shot*", que se podría traducir como única pasada, significa que la tarea de localización y clasificación se realiza en una única pasada hacia delante en la red neuronal. "*Multibox*" ya que realiza múltiples predicciones formadas por la BB y el factor de confianza. Y la palabra "Detector" se debe a que la red es un detector de objetos que clasifica los objetos detectados. Este algoritmo se caracteriza por su velocidad de detección ya que está pensado para detectar imágenes en tiempo real.

El algoritmo SSD se divide en dos grandes partes:

1. Extracción de características, que para ello se apoyará en la arquitectura VGG-16 de la cual se habla en el siguiente punto.
2. Aplicar filtros convolucionales para la detección de objetos. Para realizar la predicción se utiliza una capa Conv4_3 (38*38*4), la cual "divide" la imagen en 38*38 recuadros y realiza 4 predicciones por celda. Cada predicción esta formada por la BB y por el número de clases que queremos detectar más uno, la cual es el fondo, en caso de que una BB no tenga objetos pertenecerá a la clase 0, que es el fondo. En la Figura 35 se puede observar el proceso que realiza la capa Conv4_3 pero en vez de ser de 38*38*4 en este ejemplo es de 8*8*4.

Figura 35, ejemplo de aplicación de la capa Conv4_3, Fuente [35]

Para detectar los objetos SSD utiliza pequeños filtros convolucionales. Después de extraer las características, SSD aplica filtros convolucionales 3*3 para realizar las predicciones. Cada filtro devuelve una predicción de tamaño igual a 4+número de clases a detectar+1. El 4 se debe al punto central de la BB formado por dos coordenadas, más la altura y la anchura. El más uno se debe a que representa el fondo de la imagen. En la Conv4_3 aplicamos 4*(21+4) filtros 3*3, obteniendo por cada casilla 4 BB con la probabilidad de que el objeto pertenezca a cada una de las clases.

$$(38 \times 38 \times 512) \xrightarrow{(4\times3\times3\times512\times(21+4))} (38 \times 38 \times 4 \times (21+4))$$

3.3.2 Arquitectura y Predicción de las BB

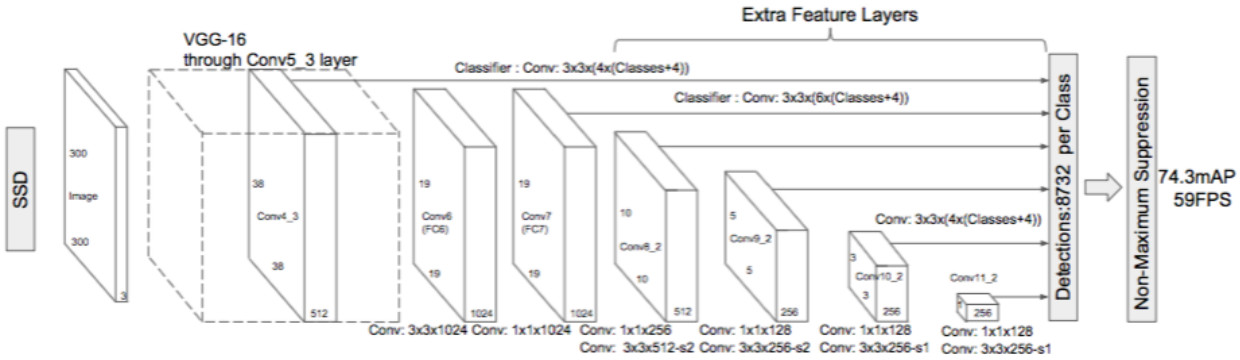


Figura 36, arquitectura SSD, Fuente [36]

Respecto a la arquitectura del SSD está se basa en la arquitectura del VGG-16. La VGG-16 es una red convolucional que ha sido entrenada con más de un millón de imágenes. Tiene 16 capa de profundidad y es capaz de clasificar imágenes en mil categorías de objetos distintas desde teclados hasta animales. La arquitectura del SSD se basa en la arquitectura del VGG-16 debido a la gran precisión que este último posee a la hora de clasificar imágenes. Aunque es un tema que se tratará más adelante, aquí se puede ver que SSD utiliza mapas de característica multi-escala

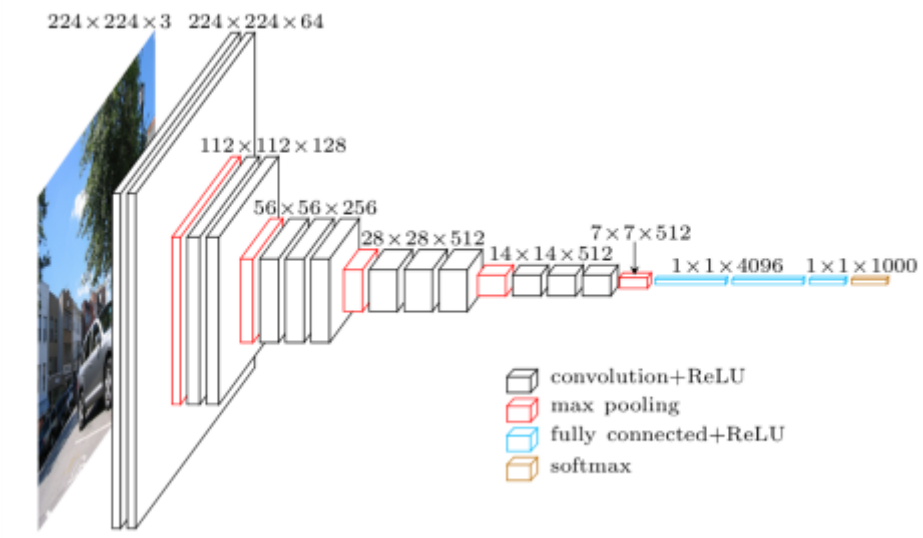


Figura 37, arquitectura VGG-16, Fuente [37]

para detectar objetos de distintos tamaños. Se puede ver en la imagen de la Figura 37 como se realizan predicciones sobre diferentes mapas de características: 38*38, 19*19, 10*10 etc.

Sin embargo, la arquitectura del SSD no es exactamente la misma que la del VGG-16, ya que sustituye la capas *fully-connected* por 6 capas convolucionales 5 de las cuales se encargan de detectar objetos. Tres de esas capas añadidas realizan 6 predicciones en vez de cuatro dando un total de 8732 predicciones para una imagen de entrada de 300*300. En la Figura 32 se puede observar la arquitectura del SSD y en la Figura 33 se puede ver la arquitectura del VGG-16.

Para predecir la BB podríamos partir de una predicción aleatoria e ir modificándola mediante el entrenamiento. El problema es que es probable que los distintos objetos que queremos detectar tengan formas distintas, por ejemplo, coches y personas. El modelo durante el entrenamiento es probable que pelee consigo mismo para ver qué forma, la de una persona o un coche, optimiza para cada predicción. Queremos que nuestras predicciones sean diversas como se puede apreciar en la Figura 38, no similares como se puede ver en la Figura 39.

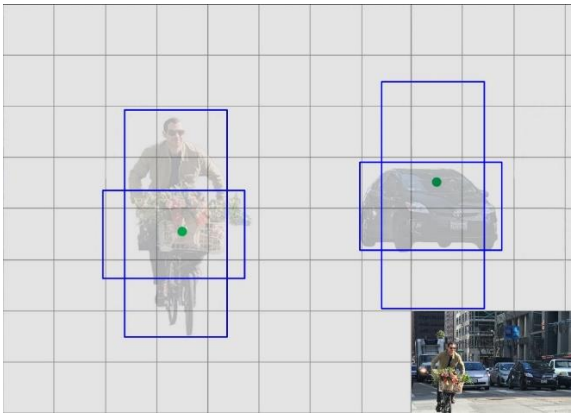


Figura 38, predicciones diversas, Fuente [38]

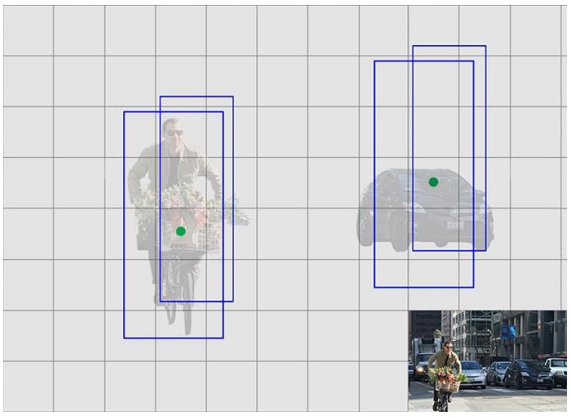


Figura 39, predicciones similares, Fuente [39]

Para que nuestras predicciones sean diversas se hace uso de anchor Boxes al igual que en algoritmo Yolo v2. En el algoritmo Yolo v2 se seleccionaban las 5 BB más representativas del conjunto mediante un algoritmo de *clustering*, como el k-means por ejemplo, y las predicciones realizadas partían de estas BB y las modificaban para ajustarse más al objeto. El algoritmo SSD realiza la misma idea solo que el método de selección de las BB representativas no se lleva a cabo mediante un algoritmo de *clustering*. Para seleccionar los anchor boxes SSD asigna un valor de escala a cada capa de extracción de características comenzando por la izquierda por la capa Conv4_3. A esta capa, que se encarga de detectar objetos a la escala más pequeña, se le asigna el valor más pequeño, que suele ser 0.2, y se va incrementando linealmente este valor de escala hasta alcanzar el valor de 0.9 en la capa de más a la derecha. Computando la escala con el aspecto ratio, valor que se le asigna a cada capa, calculamos la altura y la anchura del anchor box. La anchura se calcula como $w = escala * \sqrt{aspectratio}$ y la altura como $h = escala/\sqrt{aspectratoi}$. SSD suele seleccionar de 4 a 6 anchor box por defecto.

SSD define como predicciones positivas aquellas BB predichas que tienen una IoU>0.5 respecto a la BB real, éstas predicciones positivas son las que se tienen en cuenta a la hora de calcular la función de coste. SSD utiliza mapas de características multi-escala y anchor boxes para detectar objetos a diferentes escalas. Para mostrar esto nos vamos a apoyar en la Figura 40. El perro se

localiza mediante una BB en rojo en el mapa de características de la capa 4*4, sin embargo, ninguna BB encaja correctamente con la posición del perro en la 8*8. Con el gato pasa lo mismo, pero al revés, se detecta en la capa de alta resolución de 8*8, mediante las 2 BB azules, pero no en la capa 4*4.

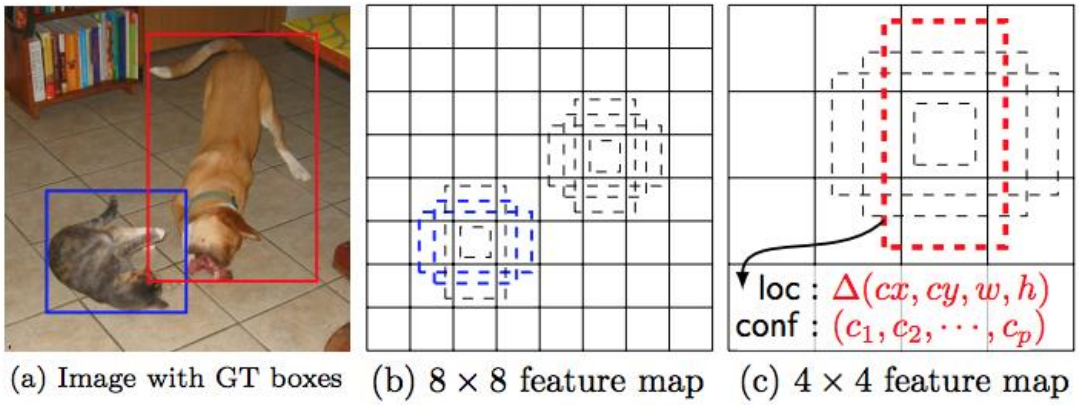


Figura 40, mapas de características a distintas escalas y BB por defecto, Fuente [40]

Las capas de alta resolución son las que se encargan de detectar objetos pequeños, en SSD esta capa sería la Conv4_3 la cual es de 38*38, una reducción considerable partiendo que la imagen de entrada es de 300*300. Debido a esto SSD suele dar resultados peores que otros detectores de objetos a la hora de detectar objetos pequeños.

3.3.3 *Loss function*

La función de coste se divide en 2 partes, la *localization* los que mide el error entre las BB predicha y la BB real, solo de las BB positivas, las negativas se ignoran. La *localization loss* se calcula mediante la fórmula de la Figura 41. En la cual se utiliza la función $smooth_{L1}$. Esta función trabaja de la siguiente de manera:

$$smooth_{L1}(x) = \begin{cases} 0.5x^2 & Si\ |x| < 1 \\ |x| - 0.5 & en\ otro\ caso \end{cases}$$

The localization loss between the predicted box l and the ground truth box g is defined as the smooth L1 loss with cx, cy as the offset to the default bounding box d of width w and height h.

$$L_{loc}(x, l, g) = \sum_{i \in Pos}^{N} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^{k} \text{smooth}_{\text{L1}}(l_i^m - \hat{g}_j^m)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^h}\right)$$

$$x_{ij}^p = \begin{cases} 1 & \text{if } IoU > 0.5 \text{ between default box } i \text{ and ground true box } j \text{ on class } p \\ 0 & \text{otherwise} \end{cases}$$

Figura 41, localization loss, Fuente [41]

Por otro lado, tenemos la *confidence loss*, que mide el error a la hora de predecir la clase a la cual pertenece el objeto. Cada predicción positiva se penaliza de acuerdo al factor de confianza

de la clase correspondiente. Para las predicciones negativas se penaliza la perdida respecto a la clase "0", la cual representa que ningún objeto se ha detectado. El cálculo de esta función se muestra en la Figura 42.

It is calculated as the softmax loss over multiple classes confidences c (class score).

$$L_{conf}(x,c) = -\sum_{i\in Pos}^{N} x_{ij}^{p} log(\hat{c}_i^p) - \sum_{i\in Neg} log(\hat{c}_i^0) \quad \text{where} \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}$$

where N is the number of matched default boxes.

Figura 42, confidence loss, Fuente [42]

La función de coste final se computa como:

$$L(x,c,l,g) = 1/N(L_{conf}(x,c) + \propto L_{loc}(x,l,g))$$

3.3.4 Hard negative mining, data augmentation y resultado del SSD

Nuestro objetivo con cualquier algoritmo de detección de objetos en este caso el SSD, es detectar ciertos objetos en una imagen, pero el número de predicciones obtenidas es mucho mayor al número de objetos que hay en la imagen. Hay más predicciones negativas que positivas lo que crea un desbalance en el entrenamiento ya que la clase 0, la que representa que no hay ningún objeto recibe muchas predicciones. Sin embargo, SSD necesita estas predicciones para saber que es una mala predicción. Para remediar esto, SSD en vez de utilizar todas las predicciones negativas, ordena las predicciones negativas por *confidence score* y selecciona aquellas que están en el top de tal forma que el ratio entre las negativas y positivas sea de 3:1, lo que da lugar a un entrenamientos más rápido y estable.

A la hora de desarrollar un modelo para detectar objetos la parte más importante son los datos. Desgraciadamente, muchas veces los datos son difíciles de conseguir ya que hay que invertir una gran cantidad de tiempo y dinero para obtenerlos y etiquetarlos. Aquí es donde el *Data Augmentation* juega un papel fundamental. El *data Augmentation* consiste en modificar ligeramente los ejemplos de entrenamiento, ya sea aplicando un pequeño zoom, cambiando el color de la imagen o volteándola en horizontal para tener una mayor variedad de datos sin añadir ejemplos nuevos, partiendo de los que se tiene. SSD es un algoritmo que se ve muy beneficiado al aplicar *data augmentation*. En la figura 43 se puede observar cómo se mejora la mAP sobre el mismo conjunto de entrenamiento al ir aplicando diferentes técnicas de data *augmentation*.

data augmentation	SSD300		
horizontal flip	✔	✔	✔
random crop & color distortion		✔	✔
random expansion			✔
VOC2007 test mAP	65.5	74.3	77.2

Figura 43, mejora del mAP en SSD300 al aplicar data augmentation, Fuente [43]

SSD al igual que Yolo aplica *non-maximum suppression* para eliminar las predicciones duplicadas. Se podría decir que hay dos tipos de SSD. El SSD300 que recibe imágenes de entrada de 300*300

y el SSD512 que recibe imágenes de entrada de 512*512. El primero se caracteriza porque al ser las imágenes menos "pesadas" es bastante veloz mientras que el segundo es más preciso, pero más lento. Una comparación de los resultados del SSD respecto a otros algoritmos de detección de objetos se puede ver en la Figura 44.

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	~ 1000 × 600
Fast YOLO	52.7	155	1	98	448 × 448
YOLO (VGG16)	66.4	21	1	98	448 × 448
SSD300	74.3	46	1	8732	300 × 300
SSD512	76.8	19	1	24564	512 × 512
SSD300	74.3	59	8	8732	300 × 300
SSD512	76.8	22	8	24564	512 × 512

Figura 44, comparación de resultados del SSD con otros detectores de objetos, Fuente [44]

3.4. Yolo(v2) Vs SSD

Ambos algoritmos son buenas opciones para generar un modelo que detecte uno o varios objetos, sin embargo, difieren en muchos aspectos. En este apartado se centra en mostrar las principales diferencias teóricas entre ambos métodos que se puede observar en la Tabla 1:

Parámetro para comparar	YOLO v2	SSD
Tamaño de la imagen de entrada	El tamaño de la imagen de entrada tiene que ser múltiplo de 32.	El tamaño que se suele escoger es de 300, denotado como SSD300. Este tamaño permite que la red pueda trabajar con una mayor cantidad de frames por segundo a la hora de realizar predicciones. Otro tamaño de entrada que se suele utilizar es de 512. El SSD 512 es más preciso, pero no puede procesar tantos frames por segundo como el SSD300.
Salida de la red	Yolo devuelve 4 parámetros que hacen referencia a la posición de la BB, un factor de confianza de la BB predicha y la probabilidad de que el objeto pertenezca a cada una de las clases de objeto que se quiere detectar.	SSD devuelve 4 parámetros para señalar la posición de la BB y un valor por cada clase más 1 , el cual hace referencia al fondo.
Método de selección de los anchor boxes	Mediante un algoritmo de *clustering*.	Lo realiza mediante un método propio.(punto 3.3.1)

Data augmentation	El *data Augmentation* es una técnica del cual Yolo se puede beneficiar pero no depende tanto de esta técnica como el SSD.	Es fundamental para el correcto funcionamiento del SSD.
Número de predicciones	El número de predicciones finales se limita al número de objetos que cree haber detectado.	Al tener una clase que representa el fondo, la cantidad de predicciones finales es muy elevada si lo comparamos con el número de predicciones sobre una misma imagen con otro algoritmo, como el Yolo.
Localization loss	Para calcular el error en la localización de la BB, Yolo solo utiliza la BB responsable de localizar el objeto. Es decir, aquella que tiene la IoU más alta con el valor real de la BB.	SSD clasifica cada BB como positiva o negativa, dependiendo de si la IoU sobre el valor real es mayor que 0.5. Solo tiene en cuenta las BB positivas para calcular el error en la localización.
multiescala	Yolo v2 entrena con imágenes de distintas escalas, siempre y cuando sean múltiplo de 32, pero no realiza predicciones a distintas escalas como SSD.	Realiza predicciones a distintas escalas.
Objeto/celda	Yolo solo detecta un objeto por celda.	SSD no tiene esta limitación.

Tabla 1, comparación teórica Yolo vs SSD

Estas son algunas de las diferencias teóricas principales entra ambos métodos de detección de objetos. En la tabla se ha intentado comparar aspectos más específicos y no elementos más generales como la arquitectura o la función de coste final las cuales son distintas en ambos métodos. A continuación, se comparará mediante otra tabla las ventajas y desventajas de ambos métodos:

	Yolo	SSD
Ventajas	- La velocidad a la cual puede aplicarse. (45 fps) - Una gran generalización, capaz de aprender del mundo real y realizar predicciones en arte, por ejemplo. - Una precisión bastante buena.	- El ssd300 es bastante veloz (60 fps) y en caso de que priorice más la precisión que la velocidad, el ssd512 proporciona muy buenos resultados.
Desventajas	Al detectar un objeto por celda si hay objetos muy próximos es probable que no sea capaz de detectarlos todos.	El ssd512 es algo lento (22 fps) y suele tener problemas a la hora de detectar objetos pequeños. El ssd300 a pesar de ser más rápido es menos preciso.

Tabla 2, ventajas y desventajas de yolo y SSD

4.Detección de armas en imágenes y videos

Un sistema detector de objetos puede cumplir infinitos roles, pero como la seguridad es un tema recurrente y de máxima importancia se ha querido enfocar este trabajo fin de grado en la detección de armas. Un sistema de detección de armas con una alta precisión que trabaje en tiempo real podría ser una opción factible para mantener la seguridad en determinadas zonas. A través de las cámaras de seguridad sería capaz de detectar estas armas y dar la alarma para que personas especializadas en estas situaciones puedan actuar. A continuación, se tratarán diferentes puntos, todos ellos centrados en la implementación de este sistema detector de armas.

4.1 BBDD

Se podría decir que un sistema detector de objetos tiene dos partes fundamentales, una de ellas es el algoritmo que permitirá al ordenador ser capaz de "aprender" y generalizar la idea de uno o varios objetos para ser capaz de detectarlos, y la otra parte igual o incluso más importante que la anterior son los datos. Para que el ordenador sea capaz de aprender a detectar armas necesita una gran cantidad de datos, en este caso imágenes en las cuales se muestren armas de distintos tamaños, en distintas posiciones, de diferentes colores, etc. Es decir, ejemplos variados para que nuestro sistema sea capaz de detectar armas en situaciones muy variadas y no en casos en concreto. Además, cada imagen debe de estar etiquetada. Nuestro sistema no es un sistema de clasificación por lo que no sería suficiente con indicar si en una imagen hay un arma o no. Nuestro objetivo es localizar el arma dentro de la imagen, por tanto, por cada imagen debemos especificar cuantas armas aparecen y en que posiciones se encuentra cada una.

Con esta información nos hacemos una idea de la forma que va a tener nuestra BBDD. Por un lado, tenemos las imágenes de armas y por otro una serie de documentos, uno por cada imagen, en este caso en formato xml, para indicar la posición de todas las ramas que aparecen en la imagen. La base de datos ha ido variando a lo largo del proyecto. Hay muchas opciones para mejorar la mAP de un detector de objetos, como el *data Augmentation* o modificar valores internos del algoritmo de detección, pero una de las opciones que suelen dar mejores resultados es la de aumentar el conjunto de datos. Por ello, a largo del desarrollo se han ido añadiendo más datos para obtener mejores resultados. A continuación, presentamos las diferentes versiones de la BBDD que hemos utilizado.

- **Versión 1:** La base de datos de la cual se disponía al principio del proyecto, la cual se obtuvo de la siguiente fuente https://sci2s.ugr.es/weapons-detection, estaba formada por 3000 imágenes de armas variadas (imágenes de solo armas, de personas con armas, en distintos ángulos y posiciones) cada una de ellas etiquetada mediante un archivo xml. Además, también se disponía de un conjunto de test de 608 imágenes en las cuales la mitad eran de armas y la otra mitad no, aunque este conjunto no estaba etiquetado, se utilizó para medir la precisión del clasificador . Es decir, para ver si en imágenes en las que hay armas detecta algo y si se abstiene en imágenes en las que no hay armas .También se obtuvieron armas de otra BBDD formada por 333 ejemplos

(https://github.com/SasankYadati/Guns-Dataset) en los que había exclusivamente gente con armas, todos ellos etiquetados. En resumen, la BBDD v1 estaba formada por 3333 ejemplos.

- **Versión 2:** En esta versión la BBDD era igual que la BBDD de la versión 1 pero con el añadido de mil imágenes generadas de manera artificial (esta técnica se explica en el apartado 4.3).
- **Versión 3:** En esta versión la BBDD es igual que la BBDD de la versión 1 pero se añadieron 490 objetos mediante object tracking (esta técnica se explicará en el apartado 4.3).

4.2 Optimización de Yolo v2

Como ya se ha mencionado en este documento para tener adquirir los conocimientos necesarios sobre las redes neuronales convolucionales y cómo funciona un sistema detector de objetos, realicé el curso de *Deep Learning* que ofrece la plataforma Coursera. En ella el algoritmo de detección de objetos que se enseña es Yolo. Debido a esto y a lo novedoso que es, la mayor cantidad de pruebas se han realizado con este algoritmo. Para la implementación de este algoritmo se partió de una implementación de Yolo v2 obtenida de la siguiente fuente (https://github.com/experiencor/keras-yolo2).

Se realizaron numerosos cambios en el código para que el algoritmo funcionara sobre el conjunto de datos del cual se disponía. También se tuvieron que hacer cambios en las etiquetas de los datos debido a que las anotaciones de la BBDD de 333 imágenes estaban en un formato diferente a las anotaciones de la BBDD de 3000 imágenes. El conjunto de datos se dividió de la manera en la que se muestra en la Tabla 3.

Tipo de conjunto	% de imágenes sobre el total	Nº de imágenes
Conjunto de entrenamiento	75% del total	2500 imágenes
Conjunto de Validación	15% del total	500 imágenes
Conjunto de Test	10% del total	333 imágenes

Tabla 3, división de los datos

A la hora de realizar las pruebas, se partió de los parámetros por defecto que se pueden observar en la Tabla 4.

Parámetro	Valor
Tamaño de las imágenes de entrada	416*416
Altura de las celdas	13*13
Número de anchors	5
Obj_threshold : IoU mínima de la BB predicha con la BB real.	0.4
Nms_threshold: IoU mínima entre distintas BB predichas para considerar que están prediciendo el mismo objeto	0.3
Data Augmentation	-Flip horizontal con una probabilidad de 0.5 -Zoom 1-1.1
BB por celda	5

optimizador	Se utilizó el Adam con los siguientes parámetros: *-Learning rate:* 0.5e-4 -Beta 1: 0.9 -Beta 2: 0.999
Batch size	16

Tabla 4, valores iniciales de los parámetros fundamentales

4.2.1 Primer acercamiento en la optimización de Yolo v2.

A la hora de realizar las pruebas se fue jugando con los valores de algunos parámetros para ver si el sistema era capaz de obtener buenos resultados. Las primeras pruebas que se realizaron fueron para adquirir una idea general sobre qué valor sería el más adecuado para el número de épocas y el *batch size.* Para ello, se realizaron 3 pruebas sencillas y se observó los resultados sobre el val_loss final y los resultados en la predicción de armas sobre elementos que se encontraban sobre el conjunto de entrenamiento o validación. En la primera prueba se escogió 10 épocas y un *batch size* de 16 pero usando unos anchor boxes por defecto y no extrayéndolos del propio conjunto. De esta forma se obtuvo un val_loss final de 0.29 y resultados bastante malos como se puede observar en la Figura 45. En la segunda prueba se aplicó el k-means 100 veces obteniendo los 5 anchor boxes más representativos en nuestro conjunto y se elevó el número de épocas a 30, obteniendo un val_loss de 0.2389 y detectando mejor las armas en las fotos como se puede ver en la Figura 46, aunque falla en algunas imágenes, Figura 47. Y la tercera prueba que se realizó fue con los mismos anchors y el mismo número de épocas, pero con un *batch size* de 4 obteniendo un val_loss de 0.2379 y un resultado algo superior en las imágenes aleatorias en las cuales se probó.

Figura 45, ejemplo de detección de armas en la primera prueba.

Figura 46, ejemplo de detección de armas en la segunda prueba.

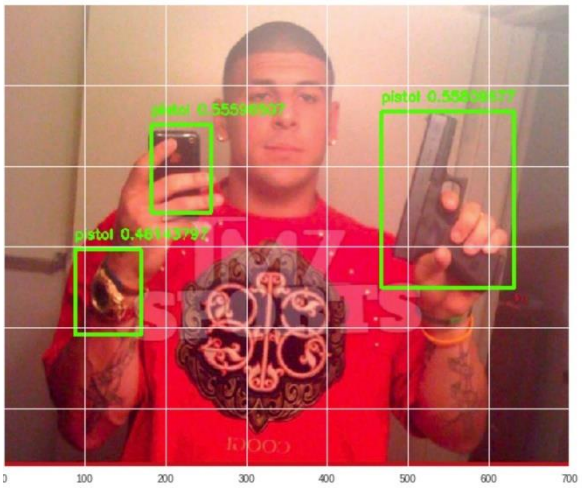


Figura 47, ejemplo de detección de armas en la segunda prueba.

Figura 48, ejemplo de detección de armas en la tercera prueba.

Hay que remarcar que las imágenes son del conjunto de entrenamiento o del conjunto de validación, por tanto, son ejemplo que el sistema de alguna forma ya ha utilizado para ser entrenado. Estas pruebas se realizaron simplemente para hacerse una idea sobre qué valores pueden ser mejores para nuestro clasificador a grandes rasgos. Gracias a estas pruebas se dedujo que el número de épocas podría ser más elevado ya que hay mejora de 10 a 30, que los anchors generados sobre el conjunto de datos daba buenos resultados y que un *batch size* de 4 parece ser mejor opción que uno de 16, debido a que en la segunda prueba en la imagen de la Figura 48 no detectaba arma, y en la tercera prueba sí. Por tanto, en las siguientes pruebas en las que hay una forma más rigurosa de medir cómo de bien funciona nuestro clasificador, se decidió aumentar el número de épocas, trabajar con los anchors del conjunto y modificar el *batch size* además de modificar otros parámetros como se verá en el siguiente punto.

4.2.2. Pruebas para la optimización de Yolo v2 y Data Augmentation.

Una vez que tenemos cierta intuición de qué valores podemos modificar para obtener mejores resultados en nuestro sistema se realizaron numerosas pruebas. Solo que esta vez se utiliza la **AP** para saber cómo de bien funciona nuestro sistema sobre datos que no conoce. Además del AP se utilizó otra medida. Como se menciona en el punto sobre la BBDD, en el enlace de donde se obtuvo el grueso de la BBDD también tenían un conjunto de test no etiquetado. En estas fotos no está etiquetada la posición de las armas pero la primera mitad del conjunto (de la imagen 1 a la 304) son imágenes en las que no aparece armas, y en la segunda mitad si.Debido a esto no se puede calcular la AP sobre dicho conjunto, pero podemos utilizarlo para ver si en las imágenes en las que hay armas nuestro sistema detecta algo y si se abstiene en las imágenes en las cuales no hay armas. Por tanto, usamos también esta medida, la **precisión**, que consiste en dividir el número de imágenes correctas (detectar algo si hay arma o nada en caso contrario) entre el número total de imágenes. La precisión se calcula tanto en el conjunto de imágenes que no están etiquetadas, el de 608 imágenes, como sobre nuestro conjunto de Test. En las siguientes pruebas, que se muestran en la Tabla 5, se mantuvieron los valores por defecto y se indica solamente aquellos que se modificaron. Recordar que nuestro conjunto de entrenamiento en este punto está formado por 2500 imágenes, el conjunto de validación por 500 y el conjunto de test por 333.

Prueba	Épocas	Batch Size	AP	Precisión 1	Precisión 2	Val_loss
1	30	4	61.01008%	82.28228%	88.48684%	0.25273
2	50	16	53.89231%	69.96996%	85.85526%	0.25470
3	50	8	70.09447%	88.88888%	85.19736%	0.24560
4	60	4	70.961039%	87.38738%	85.19736%	0.23485

Tabla 5, pruebas iniciales sobre Yolo

Después de estas pruebas se decidió introducir un *Data Augmentation* algo más fuerte del que había hasta ese momento, que era un zoom pequeño y un flip horizontal. Las transformaciones que se aplicaron sobre los datos fueron:

- Añadir ruido gaussiano a las imágenes.
- Eliminar el 10% de los pixeles de manera aleatoria.
- Cambiar el brillo de las imágenes ligeramente.
- Y aumentar o disminuir el contraste.

Se realizó una prueba con estas modificaciones y el resultado obtenido se puede ver en la Tabla 6.

Prueba	Épocas	Batch Size	AP	Precisión 1	Precisión 2	Val_loss
5	60	4	62.08183%	85.88588%	83.22368%	0.25306

Tabla 6, prueba con Augmentation Data

Como se puede observar en la Tabla 4, estas transformaciones sobre los datos más que ayudar perjudicaron el resultado. En este punto se estudió los casos en los cuales fallaba el sistema detector de armas. Varias imágenes en las cuales fallaba tenían el arma en una posición en la cual el cañón apuntaba hacia el suelo de manera vertical o ligeramente inclinada. Por tanto, se consideró que sería buena idea aumentar la variedad de los datos añadiendo nuevos ejemplos. Pero en vez de obtener imágenes de armas y etiquetarlas de manera manual se decidió **generar ejemplos de manera artificial**, una técnica que está dentro del *Data Augmentation*. El motivo por el cuál se decantó por esta opción y no la tradicional, fue que una vez generado el código que permite crear estas imágenes artificiales, generar los xml correspondientes que etiquetan las armas en las imágenes es muy sencillo. Siendo capaz de generar muchísimos ejemplos en un breve periodo de tiempo. También uno de los motivos fue estudiar si estos ejemplos artificiales ayudarían al sistema a generalizar la idea de arma.

Para ello se seleccionaron 100 imágenes de fondos aleatorios distintos, y 20 imágenes de armas. Cada fondo se utilizó para generar 10 imágenes. El proceso de generación de ejemplos artificiales consistía en los siguientes pasos:

1. Selección de una imagen de fondo
2. Selección de 1 a 5 armas de manera aleatoria sobre el conjunto de 20 armas.
3. Colocación de las armas elegidas sobre la imagen.
4. Guardar la imagen, generar el xml correspondiente que indica la posición de cada una de las ramas dentro de la imagen y guardar el xml.

Los pasos 2-4 se realizaban 10 veces por cada fondo, como había 100 fondos se generaron 1000 imágenes. Cada fondo se usó 10 veces y no un número más elevado , ya que no queríamos que el sistema detector de objetos sobreentrenara sobre esos fondos. En las Figuras 49 y 50 se pueden ver ejemplos de imágenes artificiales que se generaron.

Figura 49, ejemplo de imagen artificial

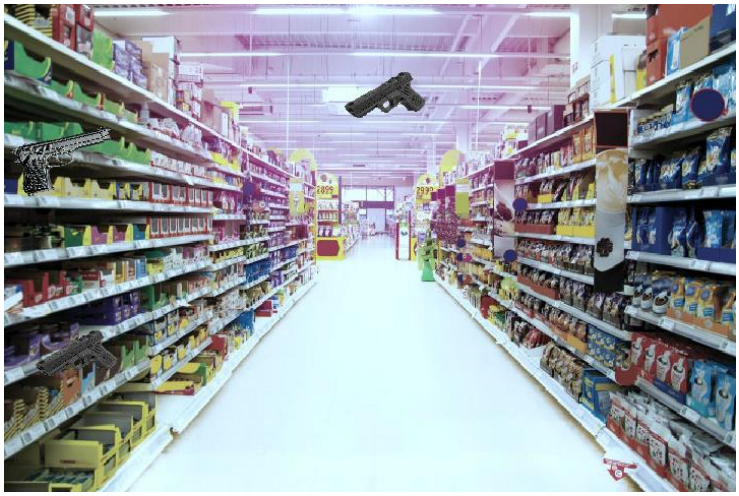


Figura 50, ejemplo de imagen artificial

Una vez generados los ejemplos, se añadieron al conjunto de entrenamiento, por tanto, nuestros datos quedaron repartidos como se puede ver en la tabla 7:

Tipo de conjunto	% de imágenes sobre el total	Nº de imágenes
Conjunto de entrenamiento	80.7% del total	3500 imágenes
Conjunto de Validación	11.54% del total	500 imágenes
Conjunto de Test	7.68% del total	333 imágenes

Tabla 7, distribución del nuevo conjunto

Todos los datos artificiales se añadieron sobre el conjunto de entrenamiento y no sobre el de test o el de validación ya que se quería ver si estos nuevos datos ayudaban a mejorar la predicción sobre los datos originales. Si los añadiéramos al conjunto de test puede que nuestro sistema predijera muy bien estos nuevos ejemplos, pero no los "antiguos" dando un mayor porcentaje

de AP y creyendo erróneamente que estos nuevos ejemplos están ayudando a nuestro sistema. Con este nuevo conjunto de entrenamiento se realizaron las pruebas que se muestran en la Tabla 8.

Prueba	Épocas	Batch Size	AP	Precisión 1	Precisión 2	Val_loss
1	20	4	61.29256%	84.08408%	87.66447%	0.28491
2	60	4	70.15341%	87.68768%	86.67763%	0.23581
3	20	8	60.84498%	83.18318%	85.19736%	0.26391
4	20	12	52.71766%	69.06906%	90.13157%	0.27707

Tabla 8, pruebas sobre el nuevo conjunto de datos

Como se puede observar los resultados no mejoran respecto a los resultados obtenidos sin los ejemplos artificiales. Además, al haber 1000 datos más en el conjunto de entrenamiento los tiempos de entrenamiento del sistema son superiores. Debido a esto se decidió prescindir de estos ejemplos y continuar con el conjunto de datos originales. A partir de este punto se decidió modificar los valores del zoom, añadir un ángulo de rotación a las imágenes y modificar los parámetros del Adam (*learning rate*, beta 1, beta 2).

Algunas de las pruebas y resultados obtenidos cambiando el zoom se muestran en la Tabla 7.

Prueba	Épocas	Batch Size	Zoom	AP	Precisión 1	Precisión 2	Val_loss
1	60	8	0.8-1.2	70.21277%	87.98798%	84.21052%	0.22285
2	60	8	0.8-1.5	62.39167%	85.58558%	80.92105%	0.24507
3	60	8	1-1.2	70.38582%	89.18918%	83.55263%	0.23928
4	60	4	0.8-1.2	70.52176%	89.18918%	87.00657%	0.24751
5	60	4	1-1.2	71.15296%	88.88888%	84.53947%	0.23639

Tabla 9, pruebas sobre el conjunto inicial modificando el zoom

En las imágenes de las Figuras 51 y 52 se muestra un gráfico de cómo varía el *val_loss* y el *train_loss* a lo largo de las distintas épocas.

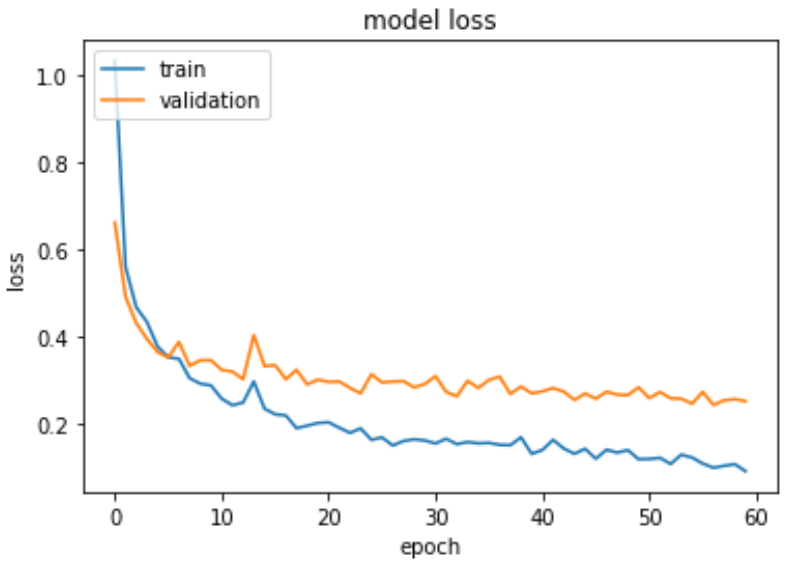


Figura 51, gráfico que representa el error en validación y en entrenamiento con zoom 0.8-1.5

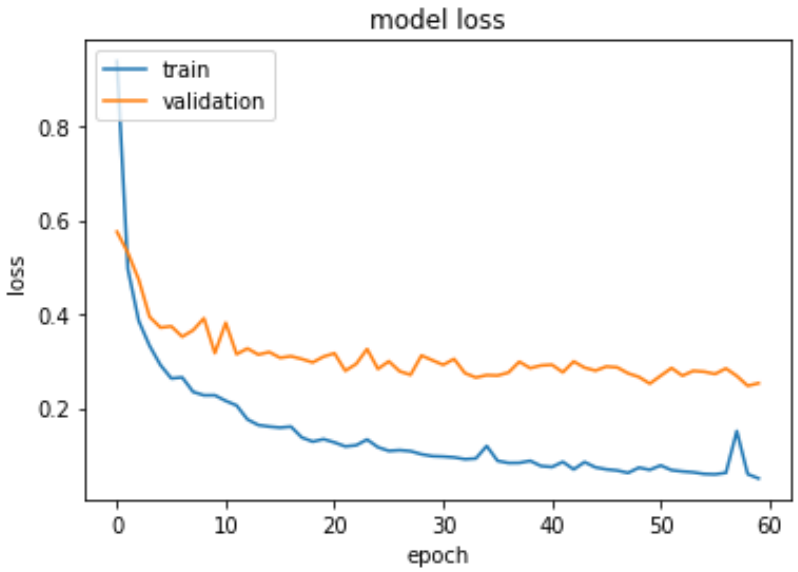


Figura 52, gráfico que representa el error en validación y en entrenamiento con zoom 0.8-1.2

Uno de los motivos por los cuales se decidió no sobrepasar las 60 épocas en cada prueba fue que a partir de las 50 épocas aproximadamente es que prácticamente no había cambios en el error en el conjunto de entrenamiento y en el conjunto de validación. Como se puede observar en los gráficos de las Figuras 51 y 52, en las primeras 10 épocas el error en ambos conjuntos se reduce drásticamente, entre la época 20 y la 50 ya se reduce paulatinamente y a partir de las épocas 60 los cambios son mínimos.

Respecto a la rotación se probó con distintos ángulos de rotación, desde ángulos pequeños como 10º hasta poner la imagen en vertical con un ángulo de rotación de 90º. También se hicieron pruebas con la probabilidad de rotar una imagen desde probabilidades alta como ½ hasta probabilidades más bajas como 1/11. Pero no se vio ningún tipo de mejora, es más, rotar las imágenes solía reducir la AP, así como la precisión. En cuanto a los parámetros del Adam modificar el *learning_rate* no produjo ninguna mejora, y tampoco modificar ligeramente los valores beta1 y beta 2, como se puede observar en la tabla 10:

P	E	Bs	Z	$\beta 1$	$\beta 2$	AP	Precisión 1	Precisión 2	Val_loss
1	60	4	1-1.2	0.95	0.999	69.94256%	85.58558 %	85.19736 %	0.24926
2	60	4	1-1.2	0.95	0.99	70.85664%	86.18618 %	91.47736 %	0.24360
3	60	4	1-1.2	0.95	0.9	70.46527 %	87.08708 %	89.30921 %	0.26395

Tabla 10, pruebas sobre el conjunto inicial cambiando los valores de beta 1 y beta 2 (P: Prueba, E: Época, BS: Batch size , Z: zoom)

En conclusión, los mejores resultados se obtuvieron con un Batch size de 4, un Zoom entre 1-1.2 y los demás parámetros con los valores que tenían al principio del estudio. El *Obj_threshold* se incrementó en alguna prueba, pero a pesar de que reduce los falsos positivos el sistema pasaba por alto muchos positivos verdaderos por lo cual se mantuvo en 0.4 un valor en el cual no se es muy exigente, pero sí que se mantiene bajo control los falsos positivos.

Como se ha mostrado anteriormente, aumentar el número de ejemplos de manera artificial no supuso ninguna mejora, pero ¿Si añadimos más ejemplos reales conseguiremos una mejor AP?

La opción más común sería obtener una gran cantidad de fotos de armas y etiquetarlas manualmente, lo cual sería tedioso y llevaría bastante tiempo. Por tanto, se pensó en una alternativa, intentar hacer esto de manera automática, aquí es donde entra en juego el *Object Tracking*.

Además de los datos obtenidos que conforman la base de datos para entrenar el sistema y los datos que no están del todo etiquetados que sirven para calcular la precisión, también se disponía de una serie de vídeos donde se puede observar a una persona posando con un arma en distintas posiciones. Esos vídeos se iban a utilizar para ver cómo de bien funcionaba nuestro sistema a la hora de detectar armas en un vídeo con el cual el sistema no tiene ninguna relación (ya que los frames del video no se han utilizado ni en el conjunto de validación ni en el de entrenamiento). Sin embargo, es un hecho que esos videos son una fuente muy interesante de ejemplos de persona con arma que se podrían utilizar para entrenar el sistema. El problema es que no están etiquetados.

El *Object Tracking* es una técnica que funciona de la siguiente manera: se le pasa un video al sistema de *Object Tracking* y se le indica donde se encuentra el objeto que se quiere localizar en el primer frame del vídeo. A partir de ese frame, el sistema de *Object Tracking* va localizando de manera automática el objeto. Por tanto, conseguimos un montón de ejemplos etiquetados correctamente de manera automática en un tiempo muy reducido. Mediante esta técnica y un vídeo de aproximadamente 9 minutos se consiguieron 490 ejemplos nuevos. Para no sobreentrenar el sistema con estos nuevos ejemplos se seleccionaba un frame por cada 30. Por tanto, la distribución de nuestro nuevo conjunto queda reflejada en la Tabla 11:

Tipo de conjunto	% de imágenes sobre el total	Nº de imágenes
Conjunto de entrenamiento	75% del total	2868 imágenes
Conjunto de Validación	16.27% del total	622 imágenes
Conjunto de Test	8.7% del total	333 imágenes

Tabla 11, distribución del nuevo conjunto, añadiendo los ejemplos del Object Tracking.

La distribución que se siguió fue, el 75% de los nuevos ejemplos para el conjunto de Train y el otro 25% para el conjunto de validación. Esta vez sí que se añadieron ejemplos al conjunto de validación ya que estos ejemplos eran reales. Los resultados que se obtuvieron con este nuevo conjunto usando la configuración de los parámetros que mejor resultado nos habían aportado fue el que se puede ver en la Tabla 12:

P	E	Bs	Z	AP	Precisión 1	Precisión 2	Val_loss
1	60	4	1-1.2	62.2814%	86.4865%	88.1579%	0.22123

Tabla 12, pruebas sobre el nuevo conjunto

Aunque el Object Tracking es una técnica que puede ahorrarnos mucho tiempo no es perfecta, ya que el video que le pasamos debe de cumplir ciertas características para que este funcione correctamente. En primer lugar, el objeto a detectar debe de estar continuamente visible. Es decir, si en un frame aparece una persona con un arma, en el siguiente hay un cambio de plano y se ve a lo que está apuntando la persona, por ejemplo, y en el siguiente frame se vuelve a enfocar a la persona con el arma es probable que el sistema no sea capaz de establecer donde está el arma ya que ha "desaparecido" en los frames intermedios. Es decir, si el objeto que queremos detectar desaparece completamente en un frame y luego vuelve a aparecer el resultado de la detección es muy malo.

Además, es preferible que el objeto no haga movimientos bruscos ni erráticos ya que en ese caso el *Object Tracking* tiende a fallar. Otro de los problemas que puede presentar es que cuanto mayor sea el número de frames que ha analizado, es probable que haya ido cometiendo pequeños errores en la localización del objeto en cada uno de ellos, y en los últimos frames del vídeo las detecciones sean algo imprecisas. Lo satisfactorio que sea el resultado que ofrece el Object Tracking no depende solo del video utilizado sino también de la localización del objeto a detectar que le indicamos al principio. En caso de que ajustemos mucho la BB al arma el sistema de *Object Tracking* acaba siguiendo más la mano de la persona que sujetaba el arma que el arma como tal. Y si la localización inicial del arma era muy amplia acababa volviéndose muy imprecisa. Por tanto, para que el *Object Tracking* funcione correctamente hay que seleccionar un video adecuado y proporcionarle una localización del objeto no muy estricta pero tampoco muy amplia.

4.3 Optimización de SSD

Una vez que se obtuvieron resultados satisfactorios con Yolo se creyó interesante estudiar, aunque en menos profundidad que Yolo, y adaptar el algoritmo SSD para hacer una comparativa de los resultados y ver si mejoraba a la hora de generalizar la idea de arma.

4.3.1 Pruebas de optimización para SSD

Para las primeras pruebas realizadas con este algoritmo se partió de la configuración por defecto del algoritmo ssd300 que se puede ver en la Tabla 13:

Parámetro	**Valor**
Tamaño de las imágenes de entrada	300*300
Número de anchors	4
Obj_threshold : IoU mínima de la BB predicha con la BB real.	0.5
Nms_threshold: IoU mínima entre distintas BB predichas para considerar que están prediciendo el mismo objeto	0.5
Data Augmentation	-Flip horizontal con una probabilidad de 0.5 -Zoom 1-4 -*Photometric distorsion:* cambiar brillo, saturación, contraste y colores de la imagen. -*Random crop:* recortar imágenes de manera aleatoria.
BB por celda	2
optimizador	Se utilizó el Adam con los siguientes parámetros: -*Learning rate:* 0.001 -Beta 1: 0.9 -Beta 2: 0.999

Tabla 13, valores por defecto de los parámetros más importantes

Las primeras pruebas que se realizaron fueron sobre el conjunto original, es decir, aquel que no posee ejemplos artificiales ni ejemplos obtenido mediante el *object Tracking*. La distribución de este conjunto se puede ver en la Tabla 3. Se decidió realizar las pruebas sobre este conjunto debido a que los otros 2 son más grandes, y al menos en Yolo no produjeron ninguna mejora, es

más si acaso empeoraron los resultados. Las primeras pruebas realizadas se pueden ver en la Tabla 14:

Prueba	Epochs	Batch size	AP
1	40	4	45.5%
2	40	16	47.4%

Tabla 14, resultados de las primeras pruebas SSD

Como se puede observar los primeros resultados obtenidos fueron bastante negativos. Debido a esto se realizaron pequeñas pruebas sobre el *Data Augmentation* que tenía el algoritmo por defecto, descubriendo que tanto el *Photometric distorsion* como el *random crop* perjudicaban mucho más de lo que ayudaban. Además, se decidió hacer una comparación del rendimiento entre un sistema entrenado con el ssd300 y otro con el ssd512. Y a su vez, entrenar cada uno de estos algoritmos con el conjunto original de entrenamiento, que lo denotaremos como conjunto 1 y con el conjunto con los ejemplos del *object Tracking,* que lo denotaremos como conjunto 2. Las pruebas se hicieron eliminando el *Photometric distorsion* y el *random crop* como opciones de *Data Augmentation*. Los resultados obtenidos se pueden observar en la Tabla 15:

Prueba	algoritmo	Epochs	conjunto	Batch Size	Val_loss	AP
1	ssd300	60	1	16	3.6297	59%
2	ssd300	60	2	16	3.6587	59.2%
3	ssd512	60	1	16	4.1002	59.3%
4	ssd512	60	2	16	3.9178	59.1%
5	Ssd300	60	2	32	3.5898	64%

Tabla 15, resultados de las pruebas sobre ssd300 y ssd512 con los conjuntos 1 y 2.

Destacar que el error en el conjunto de entrenamiento y validación se calculan de forma distinta que en Yolo, por tanto, la escala es distinta. En las Figuras 53-56 se puede ver las gráficas que representan como va decreciendo el error en el conjunto de train y en el de validación a lo largo de las épocas, y en las Figuras 57 y 58 se puede ver una representación gráfica de la medida AP para las pruebas realizadas sobre el conjunto 1 con el ssd300 y con el ssd512.

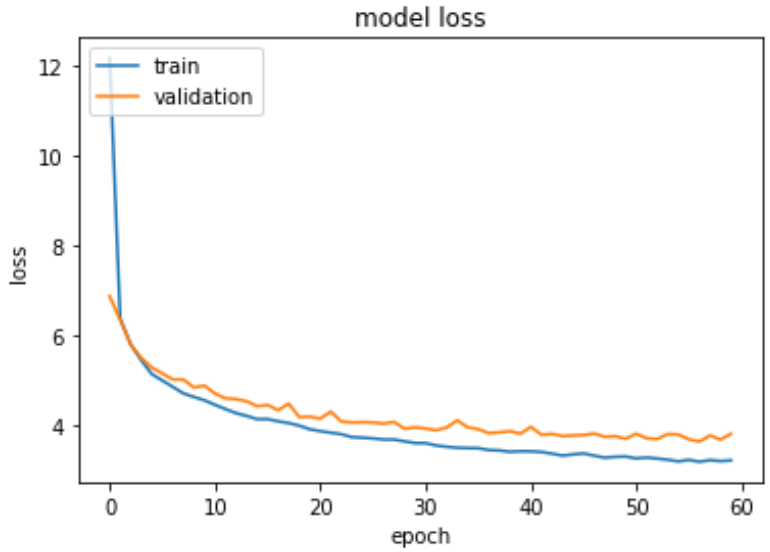


Figura 53, gráfico de la prueba 1

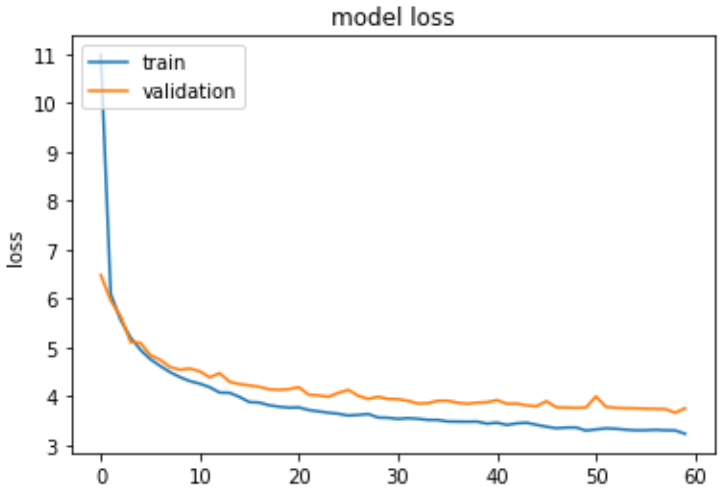


Figura 54, gráfico de la prueba 2

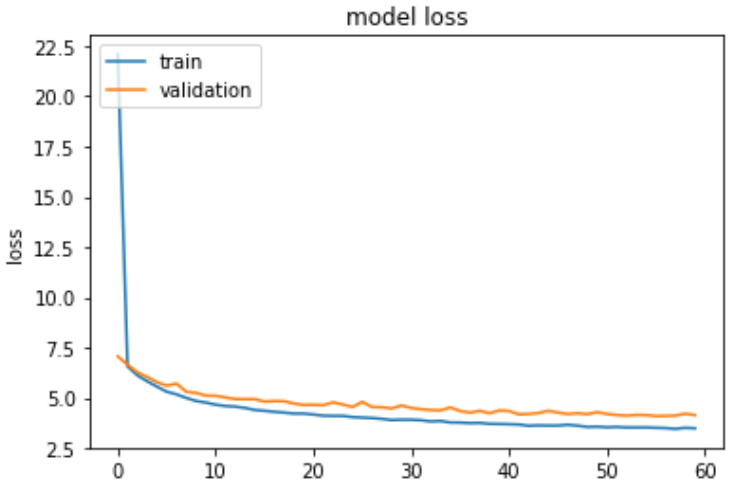


Figura 55, gráfico de la prueba 3

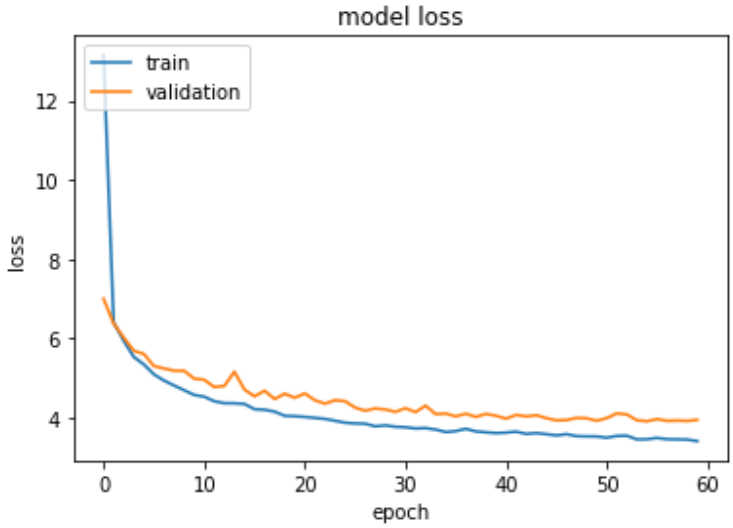


Figura 56, gráfico de la prueba 4

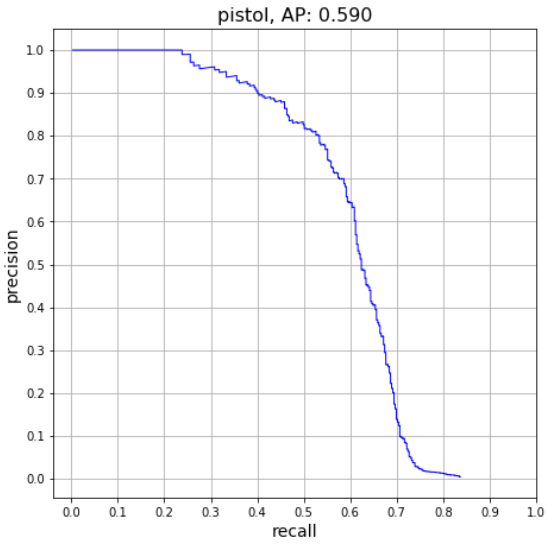


Figura 57, representación de la AP en la prueba 1

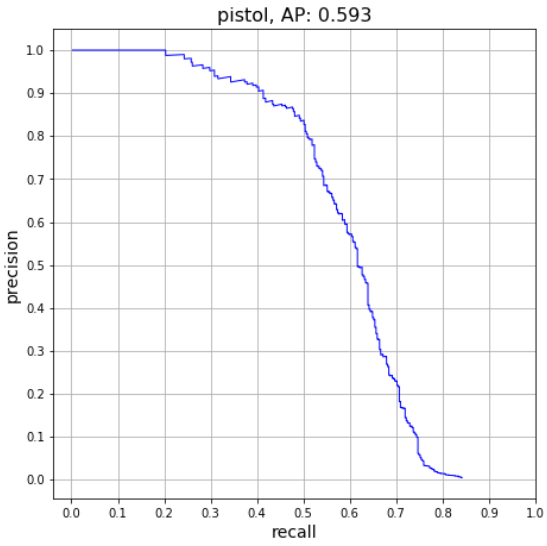


Figura 58, representación de la prueba 3

Como se puede observar, los resultados obtenidos, tanto el AP como el error en el conjunto de validación son bastante similares independientemente que usemos el conjunto 1 o el conjunto 2. Es verdad que tampoco perjudican los resultados, incluso pueden producir una pequeña mejora, como se puede ver al comparar la AP de la prueba 1 con la de la prueba 2. En cuanto a las diferencias entre el ssd512 y el ssd300 son casi despreciables en cuanto a los resultados, pero el ssd512 tarda el doble en realizar las mismas épocas ya que sus imágenes de entrada son bastante más grandes. Como se puede observar en los gráficos, a partir de la décima época el error en ambos conjuntos sigue decreciendo, aunque paulatinamente, por ello no se realizaron pruebas con un mayor número de épocas, ya que llevarían mucho tiempo y la mejora en cuanto a los resultados sería mínima. Sin embargo, comparando la AP entre la prueba 2 y 5 se deduce que el *batch size* juega un papel importante, ya que se consigue una mejora del 5% en el AP simplemente cambiando el tamaño del *batch size* de 16 a 32. En las Figuras 59 y 60 se puede observar tanto el gráfico del error en los conjuntos de entrenamiento y validación respecto a las épocas, y la representación gráfica de la AP de la prueba 5.

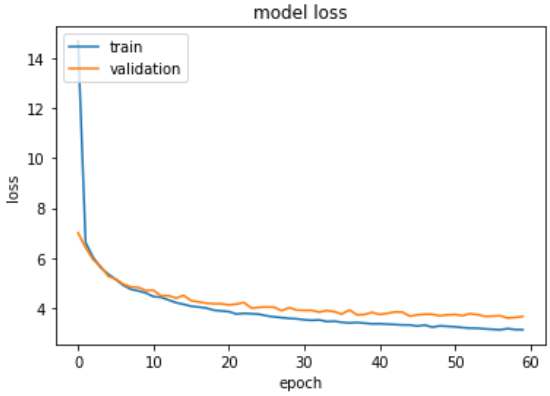


Figura 59, gráfico de la prueba 5

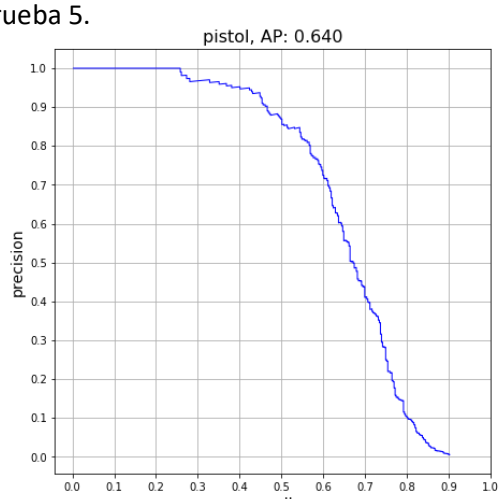


Figura 60, representación de la AP en la prueba 5

Se quiso realizar una prueba con los mismos parámetros que la prueba 5 pero cambiando el algoritmo ssd300 por ssd512. Desgraciadamente al ser el tamaño de las imágenes mayor, el *batch size* es demasiado grande lo que provoca que, por una limitación de hardware, falta de memoria RAM asociada a la tarjeta gráfica, no se pueda realizar esta prueba. También se creyó interesante, a pesar de que en Yolo no supusieron ninguna mejora, realizar una prueba en la que se comprobará qué tal funcionaría el ssd300 sobre el conjunto que tiene ejemplos generados de manera artificial, que lo denominaremos conjunto 3. Los resultados se pueden observar en la Tabla 16.

Prueba	algoritmo	Epochs	conjunto	Batch Size	Val_loss	AP
1	ssd300	60	3	16	3.7327	54.6%

Tabla 16, resultados de las pruebas sobre el conjunto 3 con ssd300.

Como se puede ver, al igual que en el algoritmo Yolo v2, los datos generados de manera artificial no suponen ninguna mejora, es más, empeoran los resultados sobre el conjunto de Test y hacen que el tiempo de entrenamiento sea mayor debido al mayor número de elementos en este conjunto.

4.4 Comparativa Yolo v2 Vs SSD (300-512)

Para finalizar este apartado se va a realizar una comparativa entre Yolo v2, ssd300 y ssd512. En la Tabla 17 se compara los distintos algoritmos siguiendo distintos criterios:

Característica	Yolo v2	SSD300	SSD512
Tiempo	Una prueba de 60 épocas sobre el conjunto 1 tarda aproximadamente 2 horas y media. El doble de rápido que el ssd512 y similar al tiempo que tarda en entrenar ssd300. En cuanto a la velocidad de las predicciones en test es muy rápido, Yolo v2 suele tardar aproximadamente 0.06 segundo por imagen.	Una prueba de 60 épocas sobre el conjunto 1 tarda aproximadamente 2 horas y media. Igual de veloz que Yolo v2 y el doble de rápido que ssd512. A la hora de realizar predicciones el ssd300 tarda alrededor de 0.77 segundos de media por imagen.	Una prueba de 60 épocas sobre el conjunto 1 tarda aproximadamente 5 horas. Bastante más lento que ssd300 y que Yolo v2. En cuanto al tiempo en realizar una predicción es el más lento de lejos, llegando a tardando 2.33 segundos de media en realizar una predicción sobre una imagen.
Data Augmentation	Después de varias pruebas se obtuvieron los mejores resultados con un zoom entre 1-1.2 y un flip horizontal de las imágenes. Y los ejemplos artificiales más que ayudar perjudicaban ligeramente los resultados.	Tanto el SSD 300 como el SSD 512 dieron mejores resultados eliminando la opción de recortan imágenes de manera aleatoria (*random crop*) y deshabilitando el cambio de saturación, contraste y color de las imágenes. (*photometric distorsions*). El cuanto, al zoom, da buenos resultados con un zoom entre 1-4. En cuanto al conjunto formado por datos artificiales, tampoco ayuda al sistema a generalizar la idea de arma empeorando ligeramente los resultados.	
Batch size	En Yolo v2, al contrario que en los otros dos algoritmos obtenemos los mejores resultados con un batch size relativamente pequeño.	Los mejores resultados obtenido son con un batch size relativamente alto de 32.	Los mejores resultados que se han obtenido ha sido con un *batch* size de 16. No se realizaron pruebas con un *batch size* de 32 por limitaciones de hardware pero se cree que mejoraría el resultado al igual que en el ssd300.
Resultados	El algoritmo que obtiene los mejores resultados de todos,	Los mejores resultados obtenidos con este algoritmo han	Resultados similares al ssd300, por tanto, algo peores que Yolo

	llegando a alcanzar más de una AP del 71% sobre los datos de test.	sido algo perores que los obtenidos con Yolo v2, pero no hay que olvidar que son similares y en algunos casos superiores que los resultados obtenidos con el ssd512 pero tardando la mitad en entrenar el sistema.El mejor AP obtenido ha sido del 64%.	v2 pero tardando lo mismo en entrenar el sistema. El mejor AP obtenido ha sido de 59.3%.

Tabla 17, comparativa entre Yolo v2, SSD300 y SSD512

A continuación, en las Figuras 61-66 se muestran algunas predicciones realizadas sobre el conjunto de test realizadas con Yolo v2 y con ssd300, utilizando la configuración que obtuvo mejor AP sobre el conjunto de test respectivamente. En las detecciones están en rojo la BB predicha y en verde la original.

Figura 61, ejemplo 1 detección con ssd300

Figura 62, ejemplo 1 detección con Yolo v2 (factor de confianza de 0.99)

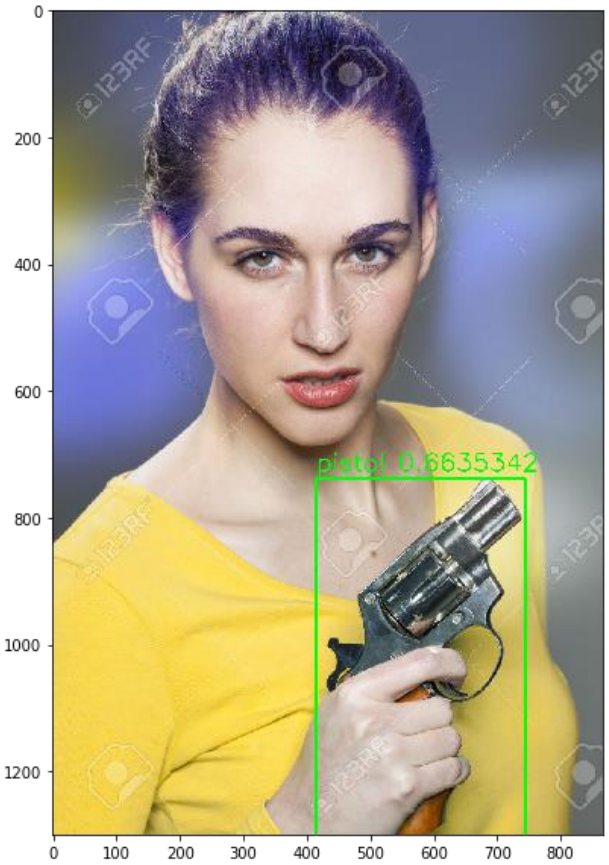


Figura 63, ejemplo 2 detección Yolo v2

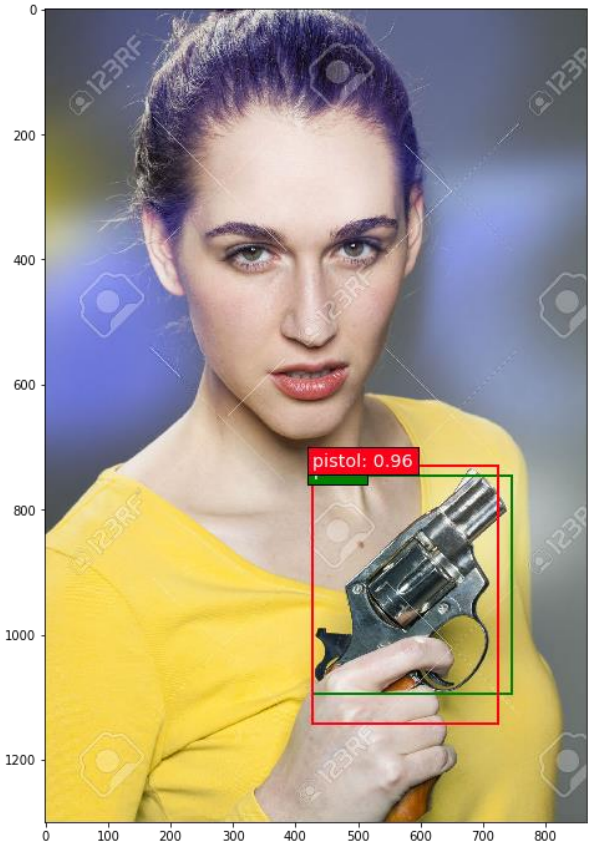


Figura 64, ejemplo 2 detección ssd300

Figura 65, ejemplo 3 detección ssd300

Figura 66, ejemplo 3 detección Yolo v2

Como se puede observar ambos clasificadores son capaces de detectar armas en fotos que no han sido utilizadas ni en el conjunto de entrenamiento ni en el de validación de manera satisfactoria. Aunque la diferencia de ambos clasificadores en el AP se nota en ciertos ejemplos, como los que se puede ver en las Figuras 67 y 68, en los cuales yolo es capaz de detectar el arma, y en cambio ssd no. Aun así hay ejemplos en los cuales ssd300 consigue una mejor detección que yolo como en el ejemplo que se puede ver en la Figuras 69 y 70.

Figura 67, ejemplo de arma detectada en Yolo v2 que no se detecta con el ssd300

Figura 68, ejemplo de arma detectada en Yolo v2 que no se detecta con el ssd300

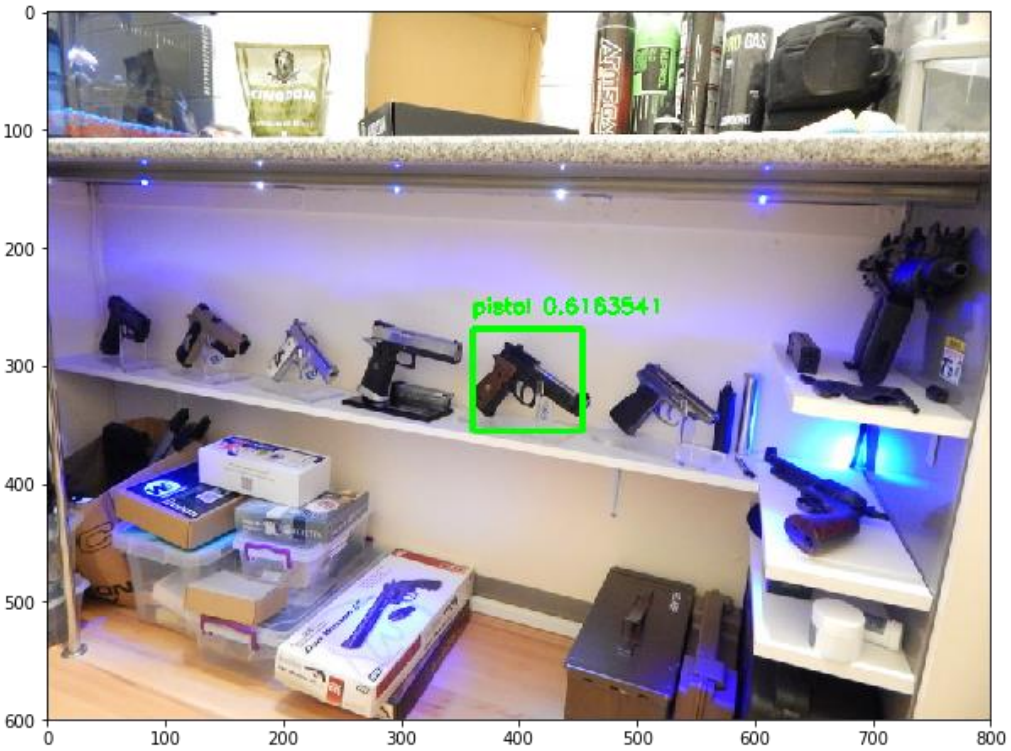


Figura 69, predicción de Yolo v2 en la cual solo detecta un arma

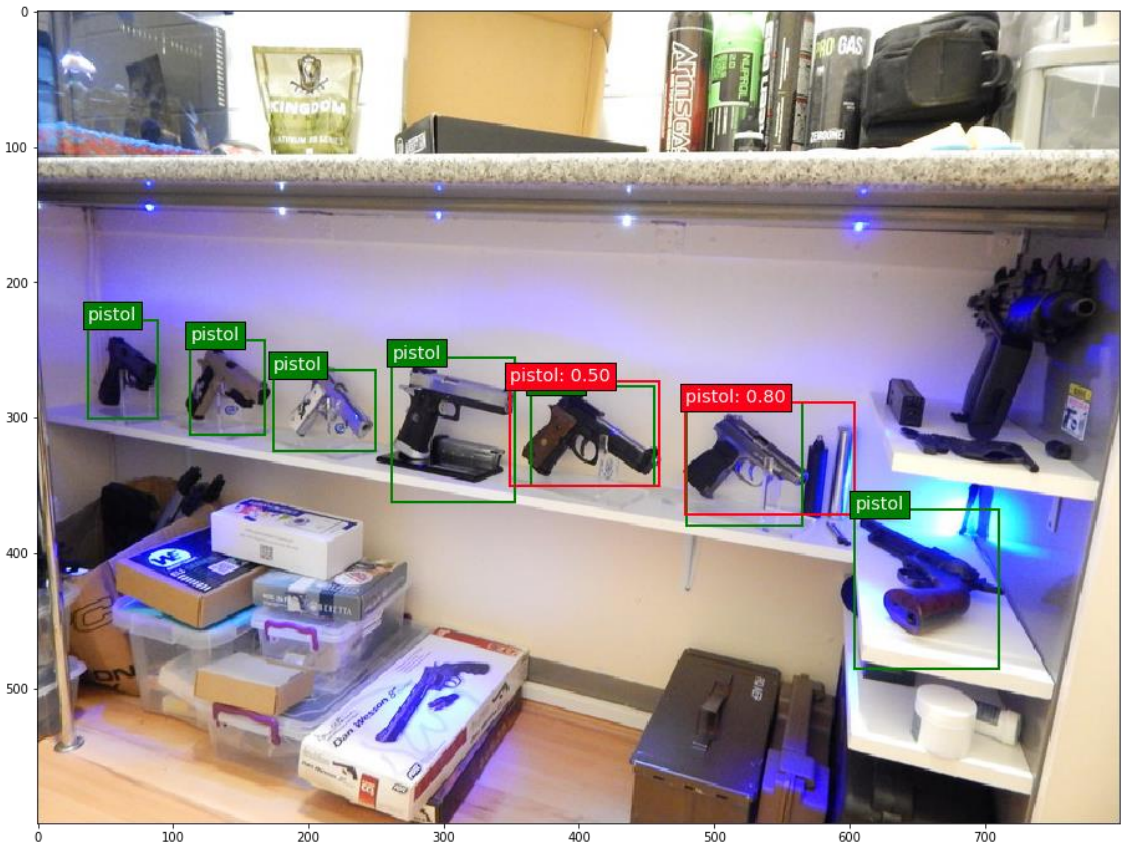


Figura 70, predicción de Yolo v2 en la cual solo detecta un arma

5.Conclusiones y líneas futuras

Después de todo el estudio realizado se han extraído ciertas conclusiones interesantes. En primer lugar, a pesar de que el Data Augmentation es un técnica extendida y muy útil para mejorar los resultados de los sistemas detectores de objetos, en este caso no nos ha proporcionado una gran mejora. Tanto rotar la imagen con distintos ángulos y distintas probabilidades, como añadir más zoom, cambiar contrastes, saturación o color a las imágenes no han proporcionado ninguna mejora en la AP. Incluso en el SSD eliminar parte del Data Augmentation proporcionaba mejores resultados. Sin embargo, es verdad que el Yolo ha dado mejores resultados con un zoom algo más elevado, entre 1-1.2.

A su vez, la creación de ejemplos artificiales a pesar de ser una vía de estudio interesante ya que permite crear una gran variedad de ejemplos, tampoco provocó ninguna mejora ni en Yolo v2 ni en los algoritmos SSD Curiosamente la generación de ejemplos reales a través del Object Tracking no beneficio al algoritmo Yolo v2, más bien lo perjudico. En cuanto a los algoritmos ssd300 y ssd512 les beneficio muy ligeramente o no supuso diferencia.

El algoritmo ssd300 es muy veloz comparado con el ssd512, pero a costa de eso pierde precisión. Sin embargo, Yolo v2 nos proporciona resultados bastante buenos con un tiempo de entrenamiento similar. El algoritmo ssd512 en teoría es más preciso al trabajar con imágenes de entrada más grandes, sin embargo, con nuestros datos no lo hace mejor que el SSD300 y tarda el doble de tiempo en entrenar y tres veces más en predecir. Además, debido al tamaño de entrada no se han podido realizar pruebas con un *batch size* de 32. En cuanto al tiempo de predicción Yolo v2 es el ganador absoluto consiguiendo ser aproximadamente 10 veces más veloz que el SSD300. El SSD512 es demasiado lento en este aspecto.

Varias de las imágenes en las que el sistema detector de objetos falla, son aquellas que tienen armas en una posición perpendicular o con un ángulo de unos 45º entre el cañón del arma y el suelo. Una de las líneas futuras para continuar con este estudio sería aumentar el conjunto de entrenamiento con datos reales con fotos muy variadas de personas sosteniendo armas en distintas posiciones. También sería interesante realizar alguna prueba más con el SSD, como rotar las imágenes o modificar los parámetros del Adam. Incluso se podría estudiar otras arquitecturas que fueran muy veloces para intentar crear un sistema que realizara predicciones en tiempo real.

Glosario Figuras

Figura 1:

https://i.ytimg.com/vi/6vwfT3-mBBw/maxresdefault.jpg

Figura 2:

https://upload.wikimedia.org/wikipedia/commons/c/c8/Mccullochpitts.png

Figura 3:

https://preview.redd.it/19bbl8mtr7221.jpg?width=638&auto=webp&s=6a2f1d43dd1bc1055c8caf7f4d6c300abb888f63

Figura 4:

https://cdn-images-1.medium.com/max/1600/1*uAeANQIOQPqWZnnuH-VEyw.jpeg

Figura 5:

https://cdn-images-1.medium.com/max/750/1*15yDvGKV47a0nkf5qLKOOQ.png

Figura 6:

https://cdn-images-1.medium.com/max/1000/1*NsiYxt8tPDQyjyH3C08PVA@2x.png

Figura 7:

https://cdn-images-1.medium.com/max/1000/1*7S266Kq-UCExS25iX_I_AQ.png

Figura 8:

https://cdn-images-1.medium.com/max/750/1*1VJDP6qDY9-ExTuQVEOlVg.gif

Figura 9:

https://cdn-images-1.medium.com/max/750/1*KQIEqhxzICU7thjaQBfPBQ.png

Figura 10:

https://cdn-images-1.medium.com/max/1000/1*kToStLowjokojIQ7pY2ynQ.jpeg

Figura 11:

https://cdn-images-1.medium.com/max/1600/0*qtiSV8B2__XR52XP.jpg

Figura 12:

https://www.jeremyjordan.me/content/images/2018/01/Screen-Shot-2018-01-23-at-2.27.20-PM.png

Figura 13:

https://cdn-images-1.medium.com/max/1600/0*QwE8M4MupSdqA3M4.png

Figura 14:

https://engmrk.com/wp-content/uploads/2018/05/Fig1-2.jpg

Figura 15:

https://engmrk.com/wp-content/uploads/2018/05/Fig3-2.jpg

Figura 16:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/12/Screenshot-from-2018-11-29-13-03-17.png

Figura 17:

https://cdn-images-1.medium.com/max/1000/1*EYFejGUjvjPcc4PZTwoufw.jpeg

Figura 18:

https://cdn-images-1.medium.com/max/1000/1*6qZXYCDUkC5Bc8nRolT0Mw.jpeg

Figura 19:

https://cdn-images-1.medium.com/max/1000/1*4Y1PaY3ZgxKt5w84_0pNxw.jpeg

Figura 20:

https://cdn-images-1.medium.com/max/1000/1*e0VY6U1_WMF2KBoKQNZvkQ.png

Figura 21:

https://cdn-images-1.medium.com/max/1000/1*0IPktA65WxOBfP_ULQWcmw.png

Figura 22:

https://www.pyimagesearch.com/wp-content/uploads/2016/09/iou_equation.png

Figura 23:

https://cdn-images-1.medium.com/max/1500/1*9ER4GVUtQGVA2Y0skC9OQQ.png

Figura 24:

https://cdn-images-1.medium.com/max/1000/1*lF6SCAVj5jMwLxs39SCogw.png

Figura 25:

https://cdn-images-1.medium.com/max/1000/1*BwhGMvffFfqtND9413oiwA.png

Figura 26:

https://cdn-images-1.medium.com/max/1000/1*QT7mwEbyLJYIxTYtOWClFQ.png

Figura 27:

https://cdn-images-1.medium.com/max/1000/1*Yc_OJIXOoV2WaGQ6PqhTXA.png

Figura 28:

https://cdn-images-1.medium.com/max/1000/1*aW6htqx4Q7APLrSQg2eWDw.png

Figura 29:

https://cdn-images-1.medium.com/max/1000/1*EXa-_699fntpUoRjZeqAFQ.jpeg

Figura 30:

https://cdn-images-1.medium.com/max/1000/1*9ordwhXD68cKCGzuJaH2Rg.png

Figura 31:

https://cdn-images-1.medium.com/max/1000/1*naz02wO-XMywlwAdFzF-GA.jpeg

Figura 32:

https://cdn-images-1.medium.com/max/1000/1*38-Tdx-wQA7c3TX5hdnwpw.jpeg

Figura 33:

https://cdn-images-1.medium.com/max/1000/1*gyOSRA_FDz4Pf5njoUb4KQ.jpeg

Figura 34:

https://cdn-images-1.medium.com/max/1000/1*3IdCKSzR5R0lIE1LSmN4Bg.png

Figura 35:

https://cdn-images-1.medium.com/max/1600/1*vNaiiFUVwCfzx1znKiFYYw.jpeg

Figura 36:

https://cdn-images-1.medium.com/max/2400/1*up-gIJ9rPkHXUGRoqWuULQ.jpeg

Figura 37:

http://jesusutrera.com/articles/img/vgg_model.png

Figura 38:

https://cdn-images-1.medium.com/max/1600/1*wCHRhpMRx_SaIUfeO9kJZg.jpeg

Figura 39:

https://cdn-images-1.medium.com/max/1600/1*7iwTsezrn-tSndx96twprA.jpeg

Figura 40:

https://cdn-images-1.medium.com/max/1600/1*-KVIXjvBO5m2MQZrzWx-wg.png

Figura 41:

https://cdn-images-1.medium.com/max/1600/1*cIE7bbicMOokWQ6w41I-NA.png

Figura 42:

https://cdn-images-1.medium.com/max/1600/1*OTVm8L9RoAKtwl3XEQNkzA.png

Figura 43:

https://cdn-images-1.medium.com/max/1600/1*QSXnP4qZsQp5R4eJu4sJKQ.jpeg

Figura 44:

https://cdn-images-1.medium.com/max/1600/1*upUWQsnSlAZD1VmeyjlreA.jpeg

Bibliografía

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection.”

[2] Jonathan Hui, “Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3.” [Online]. Available: https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088. [Accessed: 27-May-2019].

[3] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. Reed, “SSD: Single Shot MultiBox Detector.”

[4] Jonathan Hui, “SSD object detection: Single Shot MultiBox Detector for real-time processing.” [Online]. Available: https://medium.com/@jonathan_hui/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06. [Accessed: 27-May-2019].

[5] Jonathan Hui, “What do we learn from single shot object detectors (SSD, YOLOv3), FPN & Focal loss (RetinaNet)?” [Online]. Available: https://medium.com/@jonathan_hui/what-do-we-learn-from-single-shot-object-detectors-ssd-yolo-fpn-focal-loss-3888677c5f4d. [Accessed: 27-May-2019].

[6] satya mallick, “Object Tracking using OpenCV (C++/Python) | Learn OpenCV.” [Online]. Available: https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/. [Accessed: 27-May-2019].

[7] Adrian Rosebrock, “OpenCV Object Tracking - PyImageSearch.” [Online]. Available: https://www.pyimagesearch.com/2018/07/30/opencv-object-tracking/. [Accessed: 27-May-2019].

[8] Jason Brownlee, “Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.” [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. [Accessed: 27-May-2019].