Trabajo Fin de Grado
Ingeniería de las Tecnologías de la Telecomunicación

Ajuste de la eficiencia en redes neuronales para la detección de señales de tráfico. Comparativa de rendimiento de Yolov-3 y EfficientDet.

Autor: Alberto García Hernández
Tutor: Antonio Jesús Sierra Collado

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Ajuste de la eficiencia en redes neuronales para la detección de señales de tráfico. Comparativa de rendimiento de Yolov-3 y EfficientDet.

Trabajo Fin de Grado

Ingeniería de las Tecnologías de la Telecomunicación

Ajuste de la eficiencia en redes neuronales para la detección de señales de tráfico. Comparativa de rendimiento de Yolov-3 y EfficientDet.

Autor:
Alberto García Hernández

Tutor:
Antonio Jesús Sierra Collado

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Ajuste de la eficiencia en redes neuronales para la detección de señales de tráfico. Comparativa de rendimiento de Yolov-3 y EfficientDet.

Proyecto Fin de Carrera: Ajuste de la eficiencia en redes neuronales para la detección de señales de tráfico. Comparativa de rendimiento de Yolov-3 y EfficientDet.

Autor: Alberto García Hernández

Tutor: Antonio Jesús Sierra Collado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

Ajuste de la eficiencia en redes neuronales para la detección de señales de tráfico. Comparativa de rendimiento de Yolov-3 y EfficientDet.

A mi familia

A mis amigos

Agradecimientos

A mi familia, de los que aprendo cada día. A mis amigos, las personas más importantes de mi vida, a todas las personas que de algún modo u otro me han ayudado a la concepción de este proyecto y a Alicia, por siempre apoyarme y estar a mi lado cuando más lo necesito.

Alberto García Hernández

Sevilla, 2021

Resumen

Los continuos avances científicos en redes neuronales e Inteligencia Artificial permiten que año tras año aparezcan nuevos modelos que mejoran las características de los anteriores y obtienen un mejor resultado en comparación a otras redes neuronales similares.

El equipo de Google Brain ha desarrollado una nueva red neuronal llamada EfficientDet, la cual mejora en eficiencia y resultados a otras redes neuronales como Yolov-3, ResNet e Inception. Esto supone un avance en el mundo del machine learning, ya que gracias a estas implementaciones y al continuo esfuerzo de los ingenieros por seguir mejorando no solo la respuesta computacional de los ordenadores, sino también los algoritmos que realizan estas operaciones, se ha conseguido lograr unos resultados nunca vistos hasta la fecha.

Esto conlleva una disminución del dataset (conjunto de imágenes) que precisa una red neuronal para realizar tareas complejas, disminuyendo así los FLOPS (Floating Points Operation per Second) que necesita un ordenador para llevar a cabo las operaciones necesarias.

Realizaremos un estudio teórico sobre el funcionamiento de las redes neuronales en general, para profundizar en las redes neuronales convolucionales. Estas están especializadas en la detección de objetos en imágenes en tiempo real; es por esto por lo que son las escogidas para detectar señales de tráfico.

Dentro de las redes neuronales convolucionales, se ha escogido EfficientDet, pues esta es una red eficiente en el uso de recursos, además de ser precisa. Para llegar a formarla, explicaremos el método de ajuste compuesto, el cual es uno de los motivos principales de la llegada de esta familia de redes neuronales.

Se va a realizar un estudio comparativo entre EfficientDet y Yolov-3 en la detección de señales de tráfico, con la intención de buscar mejoras en la seguridad vial. Según los resultados obtenidos, podremos recomendar el uso de EfficientDet para detectar señales de tráfico.

Abstract

The continuous scientific advances in neural networks and Artificial Intelligence allow new models to appear year after year that improve the characteristics of the previous ones and obtain a better result compared to other similar neural networks.

The Google Brain team has developed a new neural network called EfficientDet, which improves in efficiency and results to other neural networks such as Yolov-3, ResNet and Inception. This represents a breakthrough in the world of machine learning, since thanks to these implementations and the continuous effort of engineers to continue improving not only the computational response of computers, but also the algorithms that perform these operations, it has been possible to achieve results never seen to date.

This leads to a decrease in the dataset (set of images) required by a neural network to perform complex tasks, thus reducing the FLOPS (Floating Points Operation per Second) required by a computer to carry out the necessary operations.

We will carry out a theoretical study on the operation of neural networks in general, in order to go deeper into convolutional neural networks. These are specialized in the detection of objects in images in real time; this is why they are the ones chosen to detect traffic signals.

Within the convolutional neural networks, EfficientDet has been chosen, because it is an efficient network in the use of resources, besides being accurate. In order to train it, we will explain the composite adjustment method, which is one of the main reasons for the advent of this family of neural networks.

A comparative study will be carried out between EfficientDet and Yolov-3 in traffic sign detection, with the intention of seeking improvements in road safety. Based on the results obtained, we will be able to recommend the use of EfficientDet to detect traffic signs.

Índice

ÍNDICE DE ILUSTRACIONES

1 INTRODUCCIÓN

Las redes neuronales, la inteligencia artificial y el aprendizaje automático están más presentes en nuestras vidas de lo que solemos pensar. Cuando usamos redes sociales (Instagram, YouTube, Facebook…), estas estudian y analizan nuestro comportamiento para ofrecernos publicidad y contenido a medida. Cada día son más las empresas que solicitan que la información acerca de sus clientes se almacene en la nube (AWS, Azure, …) para que gracias a algoritmos de machine learning puedan automatizar sus procesos y realizar funciones automatizadas. Con lo cual, creamos o no, estamos rodeados de inteligencias artificiales.

Haciendo un buen uso de ellas, se obtiene un beneficio para nuestra sociedad, como es en el caso de detectar señales de tráfico en tiempo real, ya que la velocidad en la carretera es uno de los factores más importantes en la mortalidad de los accidentes. La detección de señales de tráfico y la correcta actuación ante estas supone un importante desafío para la seguridad vial.

En este proyecto, se va a hacer un estudio de la red neuronal EfficientDet, testeándola y estableciendo una base teórica y práctica de porqué EfficientDet supera a Yolov-3 en la detección de señales de tráfico.

Se estudiará el método de ajuste compuesto como modelo de mejora eficiente para las redes neuronales y en concreto su aplicación a EfficientDet.

EfficientDet es una red neuronal con un tiempo de respuesta, de entrenamiento y precisión mejores que Yolov-3. Vamos a comparar por separado esos tres parámetros en una simulación de entrenamiento de ambas redes, visualizando los resultados obtenidos.

La eficiencia dentro de las redes neuronales es un concepto muy importante, ya que en vez de desarrollar algoritmos cada vez más potentes, podríamos centrarnos en optimizar los ya existentes, pues buscamos ahorrar recursos computacionales y energéticos.

No debemos olvidar que el uso de menos recursos es muy importante hoy en día, ya que está en nuestra mano reducir las emisiones de CO_2 tanto individualmente, como en conjunto. Aunque ahora son recomendaciones y hábitos de uso para mejorar el planeta, cada vez se acerca más la fecha planteada por la Unión Europea como desafío para producir cero emisiones de CO_2 a la atmósfera en 2050.

1.1 Motivación y Objetivos

Como se ha comentado en el resumen, se van a realizar simulaciones de ambas redes neuronales, EfficientDet y Yolov-3. Vamos a entrenar las redes neuronales con un dataset (conjunto de datos que su usan para alimentar la red) de 1000 señales de tráfico, para que, con los resultados obtenidos, la hipótesis planteada en el estudio teórico pueda ser confirmada y obtengamos las conclusiones esperadas.

Este trabajo se ha realizado con el propósito de acercar el funcionamiento de EfficientDet y determinar por qué debemos usar sistemas que premien el ahorro energético de forma directa o indirecta como es

nuestro caso, ya que se van a utilizar menos recursos y por menos tiempo. Por lo tanto, el uso de redes neuronales como EfficientDet que como su nombre indica tratan de ser más eficientes. Esto puede ser el inicio de un futuro en el que tecnología y sostenibilidad vayan de la mano para darnos en los próximos años un planeta Tierra, más verde, eficiente y tecnológico.

Para la demostración de estos resultados propuestos se ha hecho uso de tres APIs (Application Programming Interface) que nos ayudarán a obtener los resultados propuestos en este trabajo.

1.1.1 Herramientas utilizadas

- **TensorFlow**: Plataforma de código abierto que permite facilitarnos la tarea de ejecutar y realizar aprendizaje automático; cuenta con multitud de librerías y redes disponibles para nuestro uso.

Ilustración 1 – Logo TensorFlow

Es una plataforma para la creación y el entrenamiento de redes neuronales, que pueden descifrar patrones y encontrar características como la que hemos usado en este experimento.

TensorFlow puede ser usado en multitud de aplicaciones prácticas, como puede ser en salud, para ayudar a los médicos en la detección de patologías mediante la inspección de radiografías, ecografías o para la detección de señales de tráfico en tiempo real, como es nuestro caso.

Fue creado por el equipo de Google Brain y liberado como plataforma de código abierto con el nombre de TensorFlow en 2015. Dentro de las librerías que tiene, están los pesos y la configuración de las redes neuronales utilizadas hoy en día. Basta con llamar a la librería deseada y esta carga y ejecuta la red que deseamos y así está disponible para nosotros de manera sencilla y sin necesidad de tener que programar la red desde cero, lo cual puede ser muy tedioso y acarrear muchos fallos de programación.

En definitiva, gracias a esta plataforma tenemos a nuestra disposición la posibilidad de tener al alcance de la mano la mayoría de las redes neuronales que se utilizan en la actualidad.

- **RoboFlow**: Plataforma de código abierto que ayuda a etiquetar las imágenes de un dataset de manera rápida e intuitiva y ayuda a generar el dataset en un formato que pueda ser leído por otras plataformas como TensorFlow.

  Gracias a RoboFlow he realizado la etiquetación de las imágenes utilizadas para este estudio.

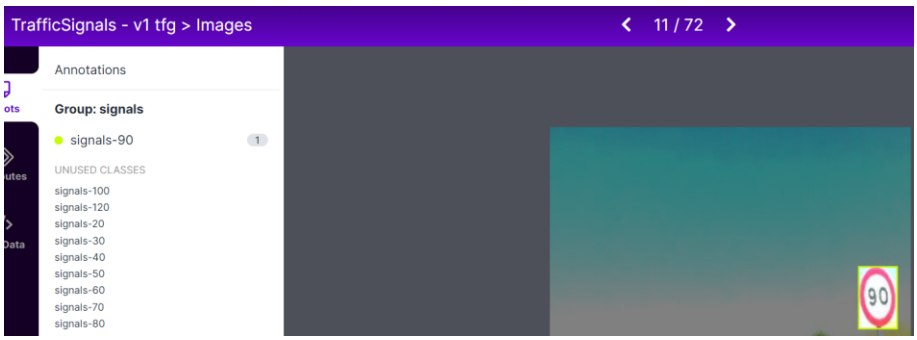


Ilustración 2 – Clases para las señales

Como se puede ver en la ilustración 2, se crea una clase para cada señal de tráfico diferente y se hace un recuadro en la imagen que corresponda con la clase correspondiente. Después de realizar el etiquetado de todas las imágenes una a una, se generan los metadatos que leerá la red neuronal para que sea capaz de identificar en qué parte de la imagen está la información importante para nosotros.

Este proceso es muy importante, ya que un incorrecto etiquetado de las imágenes puede conducir a posteriores errores de la red. Es fundamental este trabajo previo de etiquetar las imágenes de manera correcta, para que luego la red neuronal aprenda y pueda detectarlas de manera automática. Si se realiza un etiquetado incorrecto o ineficiente, la red neuronal aprenderá a partir de las imágenes que le hemos pasado etiquetadas de manera incorrecta y como consecuencia los resultados obtenidos serán erróneos.

Tras etiquetar las imágenes, la web de Roboflow, cuenta con una pequeña API auxiliar que entrena nuestro modelo de imágenes etiquetadas y con la webcam del ordenador nos permite probar en tiempo real si las imágenes que mostramos a la cámara se etiquetan de manera correcta.

Ilustración 3 – Prueba de inferencia con móvil

Como podemos observar en la Ilustración 3, tras enseñar a la cámara del ordenador varias señales de 40 km/h, es capaz de identificarlas correctamente y de manera casi inmediata.

- **GoogleColab**: Entorno de ejecución que nos permite usar la GPU y TPU de un ordenador de Google en remoto y tener así la posibilidad de realizar nuestras simulaciones más rápido que con la potencia de cálculo más reducida que tiene un portátil.

Ilustración 4 – Logo de Google Colab

Estas herramientas nos sirven de apoyo para obtener los resultados de nuestro estudio y determinar que usar EfficientDet es más seguro para detectar señales de tráfico que Yolov-3.

1.2 Antecedentes

Se parte de la base de otros proyectos realizados para dotar de más autonomía y seguridad a la conducción del vehículo, como son los proyectos realizados por:

- Lucia Reina López con *Aplicación Android y Servicio Web Spring para la detección y registro de señales de tráfico de velocidad usando Deep Learning con tiny-yolov3 y OpenCV*
- Ángel Moreno Prieto con *Detección de Objetos con TinyYOLOv3 sobre Raspberry Pi 3*
- Sergio Mellado Contigioso titulado *Aplicación Android y Servicio Web Spring para la monitorización de datos obtenidos en un vehículo haciendo uso de la plataforma FIWARE*
- Álvaro Carmona Palomares con *Securización mediante Keyrock y Wilma de Aplicación y Servicios Web para Vehículo Inteligente*

Dentro del marco de la seguridad vial y la detección de señales de tráfico en un vehículo hay que resolver una necesidad muy importante, como es la de encontrar un algoritmo más eficiente y rápido que los que estaban siendo usados hasta la fecha. El objetivo es evidenciar que EfficientDet es más preciso y eficiente que otras redes neuronales y en concreto que Yolov-3.

En primera instancia, se ha realizado una introducción teórica al mundo de las redes neuronales en global, dando una idea general de cómo funcionan, para luego centrarnos en las redes neuronales convoucionales (CNN). Nos centraremos en el método de ajuste compuesto y cómo este se aplica para formar la familia de redes neuronales de EfficientDet, para finalizar simulando esta red neuronal y comprobar si la hipótesis formulada durante la teoría es cierta.

2 ESTADO DEL ARTE

En este capítulo se presenta la investigación que hemos realizado sobre las redes neuronales, para explicar cuál son sus bases de funcionamiento y posteriormente comentar cuáles son las mejoras que aporta EfficientDet, cuáles son las características que la hacen más eficiente y por qué recomendamos su uso frente a otras redes neuronales.

El uso de las redes neuronales es muy popular hoy en día y seguirá creciendo exponencialmente en los próximos años, por lo que la eficiencia es un aspecto que debería preocuparnos más cada día. El blockchain y el mundo de las criptomonedas usa algoritmos de machine learning, muy pesados, que consumen enormes recursos para conseguir resultados, lo que produce que los ordenadores en los que estos algoritmos están alojados estén siempre ejecutándose, gastando recursos computacionales y energéticos que se podrían reducir.

Si hacemos un repaso por la historia e introducción de las redes neuronales podremos entender cómo funcionan estos algoritmos y cómo podemos trabajar para que eficiencia y precisión sean dos parámetros que vayan de la mano en el futuro, para promover un mundo mejor.

Después de sentar las bases de cómo funcionan las redes neuronales en general, pasaremos a explicar las especializadas en detectar imágenes, que son las redes neuronales convolucionales CNN (Convolutional Neural Network) y en nuestro caso el funcionamiento de la red EfficientDet, para explicar qué mecanismos utiliza que la hace más eficiente que otras redes neuronales.

2.1. Redes neuronales

En esta sección se va a desarrollar cómo es el funcionamiento general y la base de las redes neuronales, para luego poder entender y comparar por qué en este caso nos hemos decantado por EfficientDet como nuestro modelo de estudio.

Con el paso del tiempo poco a poco cada vez son más las personas que se acercan a este mundo y conocen más de cerca las redes neuronales. Como su nombre indica, las redes neuronales intentan copiar el funcionamiento neuronal de un cerebro humano. Podríamos entenderlo como un conjunto de unidades llamadas neuronas artificiales que trabajan de manera conjunta para realizar un objetivo común.

Como ya se ha mencionado anteriormente el propósito de las redes neuronales es muy diverso. Puede ser utilizado para ofrecer un contenido determinado a los usuarios de una red social según el contenido que hayan visualizado, ofrecer publicidad personalizada, simular comportamientos y respuestas humanas en robots, reconocimiento de voz y texto, prevención de fraudes, conducción autónoma y miles de aplicaciones más que podríamos seguir enumerando sin contar las que cada día a algún ingeniero/a desarrolla.

En el caso de la detección de objetos en imágenes a tiempo real, la unidad mínima de información que requiere nuestra atención son los pixels de las imágenes y el comportamiento que tiene nuestra red cuando tiene que analizarlos. Este trabajo lo realizan las redes neuronales convolucionales CNN que luego explicaremos.

Una neurona artificial es la unidad de procesamiento básica, que puede realizar una de las funciones

comentadas anteriormente de manera sencilla, como podemos ver en la Ilustración 5.

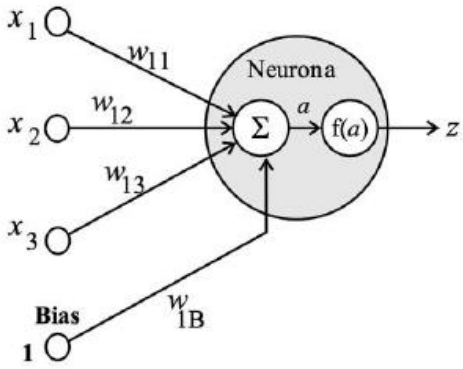


Ilustración 5 – Modelo de neurona

Lo que entendemos como neurona, no es más que una función matemática que recibe como parámetros de entrada, una serie de valores a los cuales se les hace la suma ponderada y se obtiene un resultado. Se denomina suma ponderada porque cada entrada está multiplicada por un valor, que es el peso o la importancia que se le da a cada entrada. Este peso determina qué entrada tiene más importancia y será transportado a lo largo de la red por todas las neuronas.

2.1.1 Recta de regresión

Si analizamos un poco más la ecuación de una neurona, podríamos darnos cuenta de otro concepto importante, ya que la operación interna que realiza una neurona es análoga a lo que conocemos como recta de regresión. Por lo tanto, lo que realiza la neurona es establecer una región de decisión sobre la que orientar la variable de entrada con los valores que nos aportan información y ser estos los que se transmiten a la siguiente neurona.

La ecuación de la recta de regresión tiene la siguiente fórmula, que es la suma ponderada de los valores de entrada de la neurona:

$Z = X_1 * W_{11} + X_2 * W_{12} + X_3 * W_{13} + W_{1B}$

El parámetro independiente W_{1B} se define como sesgo (bias). Este término nos permite ajustar la recta de regresión para aumentar la precisión con la que acotar nuestra región de decisión.

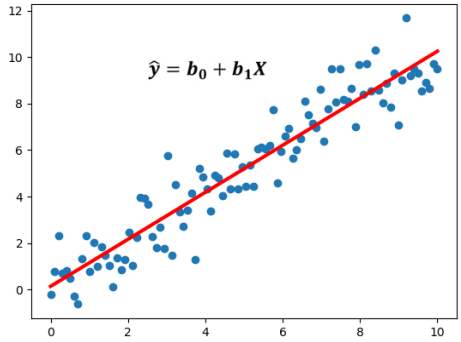


Ilustración 6 – Recta de regresión

La ecuación de una recta de regresión mantiene la misma estructura que la que realiza la neurona de manera interna. Con lo cual, podríamos decir que internamente la neurona está actuando como un modelo de regresión lineal, el cual podemos ajustar poniendo los valores deseados en los pesos (W_{li}) y en el sesgo (bias).

El concepto de neurona a nivel lógico es parecido al funcionamiento de una puerta AND o una OR en el que se toma una decisión binaria simple como pudiera ser Verdadero o Falso. Se evoluciona al concepto de redes neuronales con la unión de estas neuronas para que formen una red, que en conjunto pueden realizar operaciones más complejas.

Para completar el funcionamiento de las neuronas es necesario introducir los conceptos que explicaremos a continuación, como son las funciones de activación o los filtros de detección de características.

2.1.2 Funciones de activación

Si solo concatenamos neuronas, nuestro problema se vería reducido a suma de rectas de regresión, las cuales nos aportan poca información. Necesitamos un parámetro no lineal para que las capas en su conjunto al realizar operaciones regresivas tengan un valor añadido.

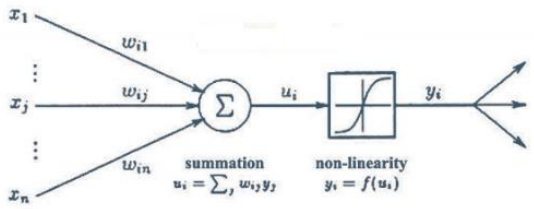


Ilustración 7 – Función de Activación

La función de activación podemos entenderla como un filtro que da valor al resultado de la neurona y dará más o menos importancia al resultado según el valor que decidamos darle.

Existen varios tipos de filtros o funciones de activación como son la ReLu, la Función Escalonada, la Función Sigmoide o la Función Tangente Hiperbólica, que son las más importantes. Variando los valores de estas y combinándolas en diferentes capas se pueden obtener resultados significativos.

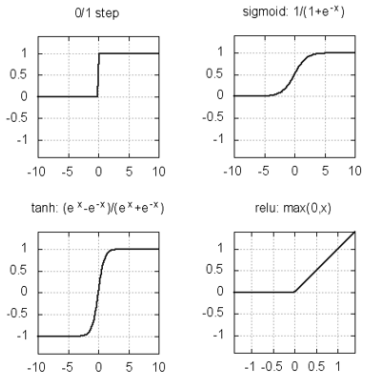


Ilustración 8 – Filtros

2.1.3 Capas de la red neuronal

Sabiendo todo esto, podemos decir que, si colocamos las neuronas de manera sucesiva, ya sea una detrás de otra en serie o en paralelo por columnas, estamos formando la denominada red neuronal que, realizando operaciones iterativas, forman una red inteligente capaz de realizar muchas de las funciones que hemos comentado en la introducción.

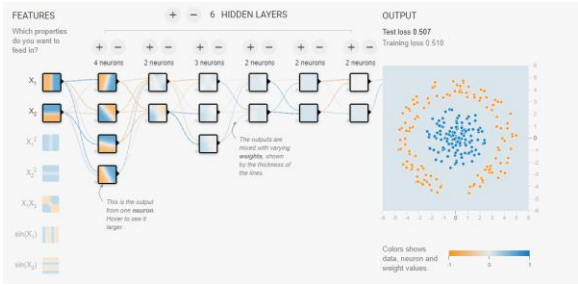


Ilustración 9 – Capas de una red neuronal

Como podemos observar en la Ilustración 9, las neuronas que se encuentran en la misma columna forman una capa. La entrada de las neuronas de la capa siguiente está ligada a las salidas de las neuronas de la capa anterior, de forma que, si se van añadiendo más capas intermedias, la red será capaz de tomar decisiones más complejas. Esto se llama Deep Learning y es el concepto fundamental que permite a las redes neuronales realizar funciones complejas.

2.2 Redes neuronales convolucionales

Las redes neuronales convolucionales son un tipo de red que se especializa en la detección de objetos en tiempo real. La diferencia principal con las redes normales es que estas realizan operaciones de convolución en sus capas. Pero sin olvidar todo lo mencionado anteriormente como es el cálculo de los errores, establecer pesos en las entradas y el sesgo.

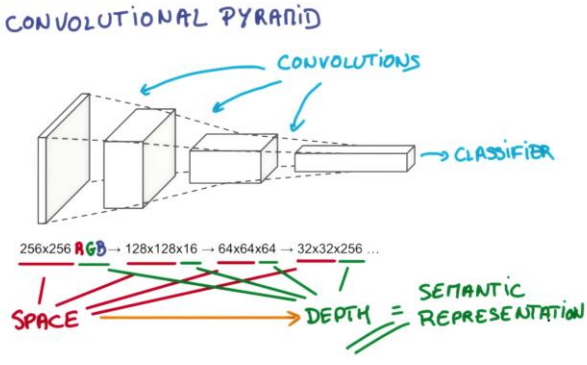


Ilustración 10 – Esquena CNN

Como se puede ver en la ilustración 10, lo que ocurre en las redes neuronales convolucionales es que estas crean un embudo, de manera que al principio toman toda la resolución de la imagen al completo y poco a poco la resolución se va disminuyendo, para que capa tras capa se puedan ir detectando patrones o características cada vez más complejos, aumentando así la profundidad (depth) de nuestra

red. En las primeras capas se detectarán texturas, colores, cambios de contraste, etc. Conforme vamos avanzando en estas detecciones, al final se convierte realmente en detectar objetos, que para nosotros tienen relevancia visual, como pueden ser las señales de tráfico que queremos detectar.

Podremos afirmar que según avanzamos en las capas podrá haber filtros especializados en detectar alguna característica en concreto.

Ilustración 11 - Filtro especializado

En la Ilustración 11, se puede apreciar un filtro diseñado para detectar filamentos, que se activará cuando reciba la imagen adecuada. Si echamos un vistazo al dataset de la Ilustración 12, este filtro se activa y genera un mapa de características positivo.

Ilustración 12 – Mapa de características

Toda esta información acerca de filtros y datasets está disponible en la plataforma *microscope.openAI.com* que permite visualizar de manera intuitiva diversas redes neuronales y cuáles son los filtros que utilizan.

Al principio, en las capas iniciales, se observan filtros que se activan con cosas simples y a medida que avanzamos, estos van aumentando la complejidad. Como pasaba anteriormente, de manera análoga una neurona sola no hace nada, de igual manera un único filtro no hace nada, es el Deep Learning de la red lo que hace que las sucesivas iteraciones permitan la detección de objetos.

2.2.1 Capas de convolución

Entrando un poco más en profundidad sobre cómo se realizan las operaciones de convolución y las diferentes capas de una red neuronal convolucional, en primer lugar, tendremos la capa de entrada, la cual recibe la imagen entera, con todos sus pixels.

Si por ejemplo tenemos una imagen con un tamaño de pixels 32x32 de alto y acho, tendríamos una capa de entrada de unas 1024 neuronas. Esto sería solo en el caso de que la imagen estuviera en un solo color (escala de grises) si tenemos una imagen a color, se separa la imagen según la configuración RGB (Red Green Blue) por lo tanto tendríamos 32x32x3 que serían 3072 neuronas en la capa de entrada. Estos parámetros serán variables, según los pixels, el tipo de imagen con la que se esté trabajando y la configuración de cada red, pues dependiendo del modelo, existen redes que aceptan más pixeles de entrada o menos, siendo este parámetro uno de los fundamentales en la eficiencia, por lo cual se desarrollará de manera más profunda en adelante.

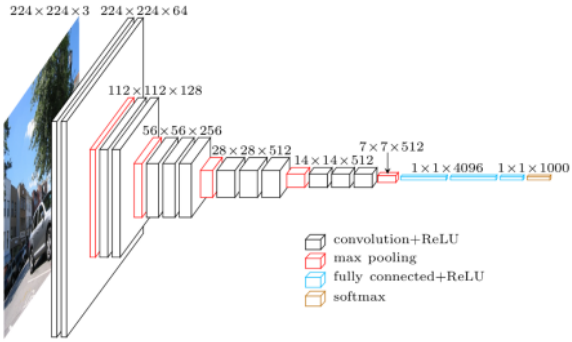


Ilustración 13 – Pirámide de convolución

Las operaciones de convolución se realizan a través de filtros matriciales que según los valores de la matriz tendrán un valor u otro de salida. Serán estos valores de las matrices los que la red tendrá que ir aprendiendo y modificando en este caso para que sean más adecuados al objetivo final propuesto.

Este concepto de filtros que se basan en una matriz es muy importante, ya que en una imagen el valor que tenga un píxel está muy ligado al de sus vecinos. La imagen no puede convertirse en un vector de entradas plano, ya que, si no, daría igual como estuviese ordenada la imagen y para nosotros no tendría ningún valor. Por lo cual es muy importante la disposición espacial de los pixels y por eso se utilizan matrices.

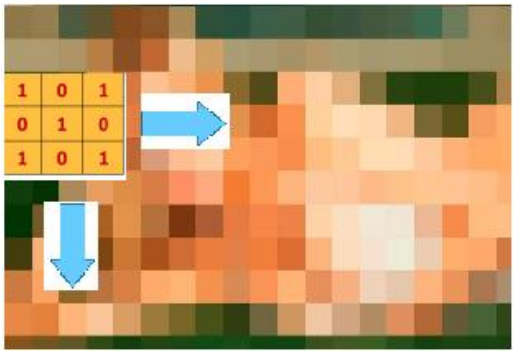


Ilustración 14 – Operación de convolución

En la Ilustración 15, podemos observar la aplicación de un filtro de convolución por las diferentes partes de una imagen, en la que los pixels no son más que información que también está dentro de una matriz. Se realizan operaciones de convolución entren matrices por las diferentes partes de la imagen, que al hacer la convolución darían otra matriz de pixels y que podrá servir para haber detectado alguna característica interesante.

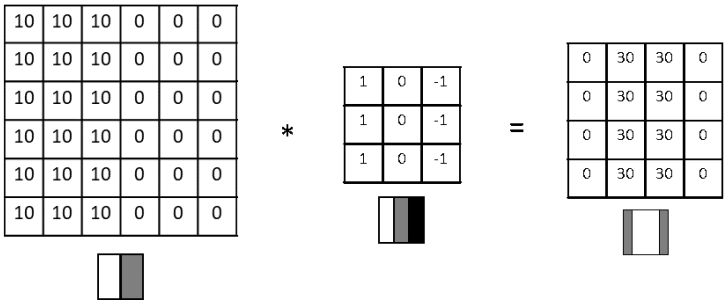


Ilustración 15 – Pasos Operación de Convolución

En la Ilustración 15, se puede ver un filtro especializado en detectar bordes verticales o lo que podríamos llamar también cambios de contraste, como este hay infinidad de filtros, será la red neuronal la encargada de establecer los valores adecuados a las matrices para hacer aproximaciones cada vez más exactas.

Estos filtros matriciales junto con las operaciones de convolución asociadas se denominan, mapas de características. Las asociaciones y uniones entre las diferentes capas de convolución son muy importantes, ya que es la arquitectura de la red lo que hace que una característica que queramos destacar atraviese toda la red para convertirse en un objeto que detectar más adelante, o termine por ser descartada porque tiene poca importancia.

2.3 EfficientDet

En este apartado, vamos a explicar el funcionamiento de la red neuronal EfficientDet. Esta red está especializada en detectar objetos, sus puntos más importantes a destacar son la utilización de un concepto denominado ajuste compuesto (compound scaling), una red piramidal bidireccional (BiFPN) para la detección de características y que su arquitectura base viene de otra red llamada EfficientNet, que mejora en rendimiento y eficiencia a otras redes neuronales, como son Yolov, Inception, Resnet.

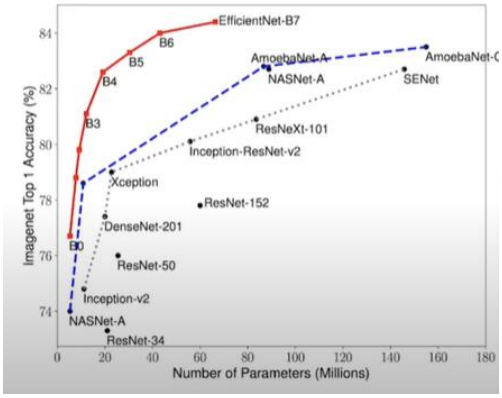


Ilustración 16 – Precisión vs Numero de parámetros

Como se observa en la Ilustración 16, con un menor número de parámetros conseguimos más precisión con la serie de redes de EfficientNet. Si observamos casos de precisión similares, la red AmoebaNet-A tiene una precisión en torno el 83 % a costa de tener aproximadamente 90 Millones de parámetros de entrada, mientras que EfficientNet-B5 obtiene una precisión del 83% aproximadamente, pero con muchos menos parámetros de entrada, 40 millones de parámetros de entrada. Por tanto, elegimos EfficientNet, ya que requiere de muchos menos parámetros de entrada y en consecuencia menos tiempo de entrenamiento.

Ilustración 17 – Arquitectura de EfficientNet-B0

Tras decidir que usamos EfficientNet, se escoge su versión más sencilla o menos pesada que es B0, la cual tiene la arquitectura que vemos en la Ilustración 18, y dentro de cada etapa podemos ver el análisis en cuanto a canales, resolución y numero de capas de cada etapa.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Ilustración 18 – Capas de EfficientNet-B0

Teniendo elegida la red que se va a usar se establece como ajustar las dimensiones de la red y como se relacionan las diferentes dimensiones entre sí. A continuación, explicaremos este procedimiento.

2.3.1 Dimensiones de una red neuronal convolucional

Las redes neuronales convolucionales tienen tres dimensiones, ancho(width), profundidad (depth) y resolución de entrada (input resolution). Normalmente lo que se suele hacer es maximizar uno o dos de estos parámetros para conseguir una mejor eficiencia y rendimiento. Lo más común es aumentar la profundidad de la red, es decir, tener más capas de neuronas en la red, lo que hace que el tiempo de computación sea mucho mayor.

También existen otras redes que aumentan la resolución de entrada como por ejemplo Inception–v3, que usa una resolución de pixels de entrada de 299x299 con lo que toma más información inicial y se detectan más patrones al principio, lo que supone más información inicial.

En el método de ajuste compuesto utilizado por EfficientDet, se actúa sobre las 3 dimensiones de la

red de forma balanceada, para que ajustándola a los valores adecuados el rendimiento y la eficiencia sean mucho más elevados, pero a su vez intentando no hacer uso de recursos computacionales innecesarios. El objetivo es que el tamaño del modelo no sea muy elevado y el número de FLOPS (operaciones de coma flotante por segundo) no se dispare.

2.3.2 Método de ajuste compuesto

Antes de entrar a explicar el método de ajuste compuesto, cabe decir, que escalar en una dimensión requiere el ajuste de las demás dimensiones en consecuencia. Como podemos ver en la Ilustración 19, si escogemos la imagen de la derecha, que tiene una resolución mucho mayor, es decir, contiene más información, necesitaremos más canales o filtros de características para ser capaces de detectar toda esta información y transportarla por la red.

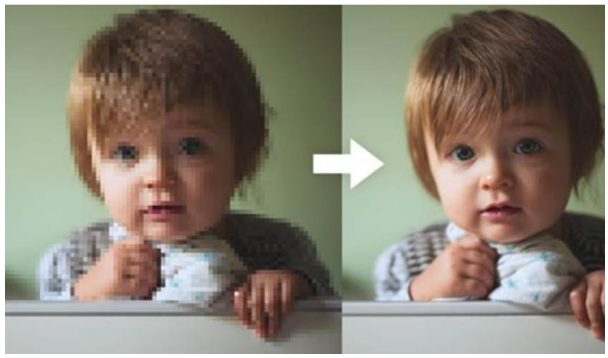


Ilustración 19 – Baja resolución vs Alta resolución

El método de ajuste compuesto (compound scaling) es muy importante, ya que no se ajusta una sola dimensión de la red, ya sea profundidad (depth), ancho (width) o la resolución de entrada (imput resolution), sino que se ajustan las tres a la vez y además no se hace de forma arbitraria, sino siguiendo unos coeficientes y parámetros determinados que hacen que este ajuste se realice de forma balanceada.

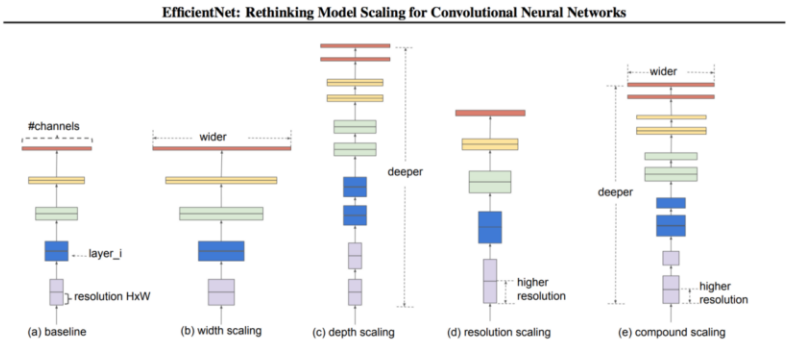


Ilustración 20 – Métodos de escalado

Como se puede ver en la Ilustración 20, el (a) es un ejemplo de red arbitraria y estas son las diferentes maneras de ajustar las 3 dimensiones de la red, primero de manera individual como se ve en la sección (b), (c) y (d) y de ajuste compuesto que se usa en la sección (e) (compound scaling)

- Ajuste en ancho (with scaling): incrementar el número de canales en cada una de las capas.

- Ajuste en profundidad (depth scaling): incrementar el número de capas de la red.
- Ajuste de resolución (resolution scaling): tomar una imagen de entrada con más resolución.
- Ajuste compuesto (compound scaling): consiste en realizar los 3 ajustes anteriores al mismo tiempo y de forma balanceada.

Para poder realizar este ajuste compuesto, se establecen 3 parámetros constantes, cada uno correspondiente a una de las 3 dimensiones a escalar. Estos son:

- α, referente a la profundidad (depth).
- β, referente al ancho (width).
- γ, referente a la resolución (resolution).

Se plantea el siguiente desafío, si queremos hacer el ajuste en las 3 dimensiones (compound scaling) para utilizar 2^N veces más recursos computacionales, entonces habrá que incrementar la profundidad (depth) α^N, el ancho (width) en β^N, y el tamaño de la imagen de entrada (resolution) en γ^N. Esto se puede entender de forma muy intuitiva, por ejemplo, si tenemos una imagen muy grande, tendremos más información inicial que antes y detectaremos más patrones, con lo cual estamos aumentando el ancho(width). La importancia de este ajuste compuesto es que se establece una relación empírica entre las relaciones de los tres parámetros ancho (width), profundidad (depth) y resolución (resolution). No se hace un ajuste de forma arbitraria y manual, lo cual es muy tedioso y requiere mucho tiempo de cálculo.

Lo comentado anteriormente, se puede argumentar matemáticamente de la siguiente manera, si definimos una capa convolucional (ConvNet) i como una función $Yi = F_i(X_i)$, donde F_i es el operador, Y_i es el tensor de salida y X_i es el tensor de entrada que está definido de la siguiente forma (H_i, W_i, C_i) , donde H_i , W_i , son tensores que expresan dimensiones espaciales de la red y C_i es un tensor que expresa la dimensión del canal.

Un conjunto de capas convolucionales (ConvNet N) se pueden expresar como una composición de estas capas de la siguiente manera $N = F_k \odot \dots F_2 \odot F_1(X_i)$. que al final, no son más que capas que realizan operaciones de convolución.

También sabemos, que las capas convolucionales dentro de una red se repiten sucesivamente, tienen sus etapas de convolución, pero capa tras capa es la misma capa de convolución la que aparece. Por lo que se puede definir una capa convolucional (ConvNet).

$$N = \bigodot_{i=1\dots s} F_i^{Li}(X(H_i, W_i, C_i))$$

Donde Li denota el número de veces que se repiten las capas convolucionales.

Teniendo esta formulación matemática en cuenta, en lugar de cambiar la arquitectura de la red, sea cual sea la arquitectura de la red, lo que se realiza es un ajuste de las dimensiones, para que esta trabaje de forma eficiente. El objetivo es lograr la máxima eficiencia y precisión, lo cual podría formularse como un problema de optimización en el que tendremos que maximizar nuestras 3 dimensiones fundamentales.

$$\max_{d,w,r} \quad Accuracy(\mathcal{N}(d,w,r))$$
$$s.t. \quad \mathcal{N}(d,w,r) = \bigodot_{i=1\ldots s} \hat{\mathcal{F}}_i^{d\cdot\hat{L}_i}(X_{\langle r\cdot\hat{H}_i, r\cdot\hat{W}_i, w\cdot\hat{C}_i\rangle})$$
$$\text{Memory}(\mathcal{N}) \leq \text{target\_memory}$$
$$\text{FLOPS}(\mathcal{N}) \leq \text{target\_flops} \qquad (2)$$

Ilustración 21 – Ecuaciones de escalado

Las 3 dimensiones y formas de escalar en una red neuronal no son independientes, sino que están relacionadas entre sí, como se puede ver en esta ecuación matemática de la Ilustración 21. Además, que para resolver este problema en su manera óptima se establece un máximo de FLOPS que no se puede sobrepasar, así como la memoria disponible para guardar las operaciones.

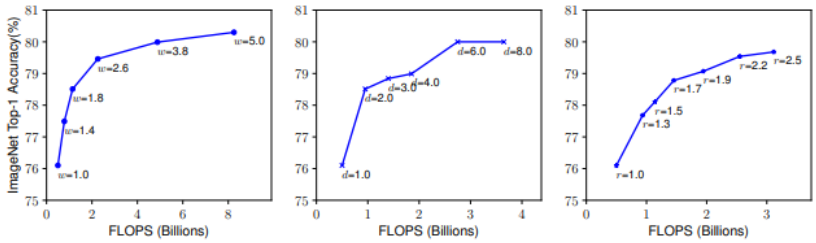


Ilustración 22 – Escalado de dimensiones

En esta Ilustración 22, se puede apreciar cómo actuar sobre cada una de las tres dimensiones de la red mejora la precisión (accuracy) considerablemente. Para modelos muy grandes, que requieren mucho calculo la ganancia de la precisión satura entorno al 80% como se aprecia en la ilustración 22. Con lo cual también habrá que tener cuidado para no saturar y no perder potencia de cálculo de manera inútil.

La clave del ajuste compuesto es hacerlo de manera balanceada. Se propone realizarlo de la siguiente manera, se define un coeficiente de ajuste Φ, que ajustara la profundidad, ancho y resolución.

$$\text{depth: } d = \alpha^\phi$$
$$\text{width: } w = \beta^\phi$$
$$\text{resolution: } r = \gamma^\phi$$
$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$
$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

Ilustración 23 – Función de escalado

Esta ecuación nos relaciona los parámetros nombrados antes, como son γ, β, γ que establecen cuantos recursos podemos dedicar al ajuste de las dimensiones y Φ que es la cantidad de recursos que disponemos en total para hacer el ajuste y se establece por el usuario. Con esta relación de valores se establece una manera sencilla e intuitiva de administrar los recursos computacionales disponibles para ajustar la red. Después de tener los parámetros sobre la máquina que vamos a utilizar, el usuario puede establecer el parámetro Φ y así determinar cómo se reparte el escalado de la red según los recursos

disponibles. Por norma general, se pueden tomar las siguientes consideraciones: doblar el ancho de la red, dobla las operaciones de coma flotante por segundo, pero duplicar el ancho o la resolución de entrada, multiplica por cuatro las operaciones de coma flotante por segundo.

En el caso de EfficientNet, se ha realizado un estudio de los parámetros de su red y se ha establecido que la configuración óptima para el ancho, profundidad y resolución es:

- d=1.2
- w=1.15
- r=1.10

Esto significa, que si aumento la resolución en un 10% el ancho tiene que aumentar en un 15% y la profundidad en un 20 % y así estaríamos trabajando con una EfficientNet óptima.

Lo que se quiere decir, es que el método de ajuste compuesto utilizado no modifica la arquitectura de la red, solamente modifica sus dimensiones ya que estas están fuertemente ligadas a la eficiencia de la red. Por lo que este método es aplicable y debería ser utilizado de ahora en adelante para ajustar las dimensiones de la red antes de ponerlas a funcionar y ser así más eficientes a la hora de realizar sus funciones.

2.3.3 BiPFN (Bidirectional Feature Piramid Network)

Esto es un tipo de arquitectura piramidal que utiliza la red neuronal EfficientDet, pero existen muchas otras las cuales vamos a explicar.

2.3.3.1 Arquitectura de redes piramidales

Algo muy común y utilizado dentro de las redes neuronales convolucionales, son las FPN (Feature Piramid Network) es una estructura dentro de la red, que se especializa en detectar características. Dentro de esta arquitectura englobamos las diferentes operaciones de convolución y los mapas de características para detectar objetos a lo largo de la pirámide de detección. Se crea una estructura piramidal que, trabajando conjuntamente, facilita y da una mejor precisión a la hora de detectar objetos en una imagen.

Este concepto de red piramidal es independiente a la red utilizada y es aplicable a todas las redes neuronales para la detección de objetos. Existe un flujo de información a lo largo de esta pirámide y podemos observar lo que se conoce como 2 caminos.

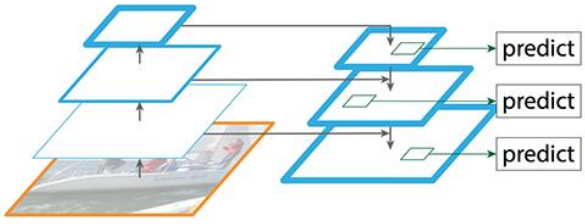


Ilustración 24 – Caminos de detección

Como se puede ver en la ilustración 23 tenemos 2 caminos, uno ascendente, otro descendente y unas conexiones laterales:

- Botton-up pathway: es el camino ascendente, en él se realizan las operaciones de convolución de manera sucesiva. La salida de la convolución de cada etapa capa será la entrada de los mapas de características utilizados en el camino descendente.
- Top-down pathway: en este camino descendente las características de más alta resolución se muestrean y son entradas de los mapas de características, las conexiones laterales hacen una fusión de los mapas de características con la convolución de la capa correspondiente.

La operación de fusión, y los mapas de características no son más que otra operación de convolución, ya sea matricial, como en el caso de los mapas de características o convolución 1x1, para fusionar una operación concreta.

2.3.3.2 Arquitecturas piramidales

Teniendo esto en cuenta, existen diferentes acercamientos y algoritmos que realizan esta función de diferente manera, cada vez se va buscando más precisión y sobre todo eficiencia. El camino de la eficiencia intenta reducir las operaciones de convolución innecesarias y que solo llevan el acarreo de una característica deseada.

Para entender este problema se necesita entender cómo se realiza la fusión de características en la red, se trata de fusionar diferentes características de diferente resolución.

$$\vec{P}^{in} = (P_{l_1}^{in}, P_{l_2}^{in}, \ldots)$$

Ilustración 25 – Características de entrada

Donde cada elemento es una característica de nivel l_i y se intenta dar solución a la agregación de características en la salida de la capa.

$$\vec{P}^{out} = f(\vec{P}^{in})$$

Ilustración 26 – Características de salida

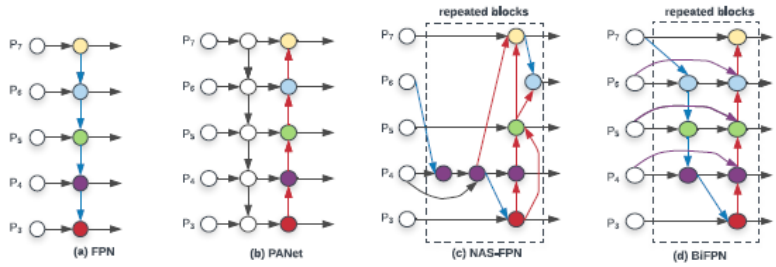


Ilustración 27 – aquitecturas de fusión de características

Como podemos observar en la Ilustración 26, existen diferentes topologías que intentan cada una a su manera arreglar el problema de la eficiencia y precisión dentro de la red neuronal.

2.3.3.2.1 PaNet

PaNet (Path Aggregation Network): como vemos en la figura, esta red da una segunda oportunidad a las características de bajo nivel, para que puedan subir y ser reevaluadas de nuevo. Explicado de manera más profunda.

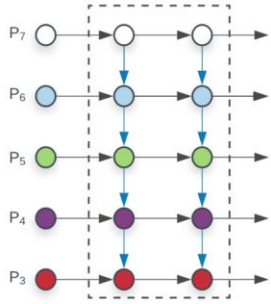


Ilustración 28 – Arquitectura de FPN

Como se ve en la ilustración 27, esta arquitectura, cuenta con 5 entradas P3-P7 donde cada nodo de entrada representa una característica de entrada con un nivel de resolución determinado y como se ha dicho, esta arquitectura fusiona las características de manera descendente.

$$P_7^{out} = Conv(P_7^{in})$$
$$P_6^{out} = Conv(P_6^{in} + Resize(P_7^{out}))$$
$$\dots$$
$$P_3^{out} = Conv(P_3^{in} + Resize(P_4^{out}))$$

Ilustración 29 – Características de salida FPN

Lo que ocurre es que en FPN, solo tenemos camino descendente, luego las características de bajo nivel no tienen oportunidad de ser reevaluadas y pierden importancia en el camino. Esto se resuelve con la arquitectura de PANet.

2.3.3.2.2 PaNet y NAS-FPN

En estos modelos se da una nueva oportunidad y se agrega un camino ascendente para que las características de bajo nivel puedan ser reevaluadas de nuevo.

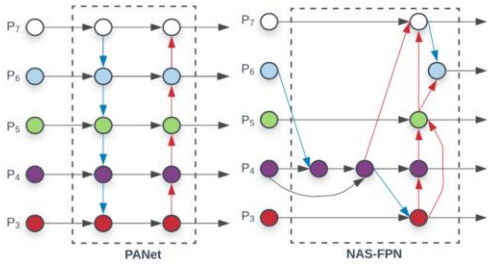


Ilustración 30 – Aquitectura PaNet y FPN

En el caso de PaNet es justo lo que acabamos de explicar y en NAS-FPN (Neural Archithecture Search–Feature Piramid Network), se realiza una búsqueda de cuáles son los mejores caminos para

hacer la fusión de características mediante machine learning, es decir la propia red hace machine learning para ver como fusionar mejor sus características y tomar mejores decisiones, pero esto es un proceso lento y hace que el tiempo de ejecución sea alto.

Luego se decide simplificar PaNet y crear PaNet Simplified aplicando un concepto muy fácil, aquellos nodos a los que solo les llega una característica de entrada se pueden quitar, con otras palabras, aquellos que no hacen fusión de características no aportan información relevante para el objetivo final de la red.

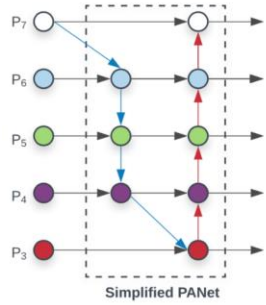


Ilustración 31 – Simplified PANet

2.3.3.2.3 BiFPN

Partiendo de la premisa de Simplified PANet, se eliminan los nodos que no realizan fusión de características. Cómo se puede observar, se han quitado aquellos nodos que solo reciben un input de entrada o característica de entrada ya que no aportan mucho y se dejan solo aquellos que tienen varios inputs o lo que podríamos decir una fusión de características. Además, el bloque básico de la red se repite tantas veces como hace falta, lo que hace más potente y precisa la detección de características.

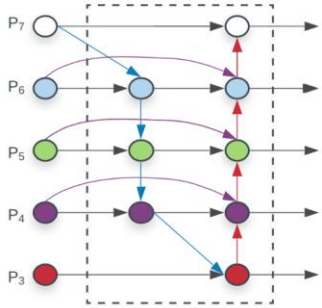


Ilustración 32 – Aquitectura BiFPN

Además, es interesante, porque como se ve en la Ilustración 31 existen unos saltos entre neuronas (skip connections) esto permite que, si tenemos una característica muy buena que queremos seguir manteniendo, en lugar de mantenerla en una operación iterativa que lo único que sirve es para recordar esa característica y que no se pierda, se guarda y se lleva de forma rápida a la fusión con otra característica deseada en un punto de la red. Lo cual es un ahorro de cálculo computacional.

La última de las optimizaciones y lo que da nombre a esta red, existe un flujo bidireccional de información, un camino ascendente y otro descendente, que se tratan de manera independiente y hacen más potente la fusión de características.

2.3.4 EfficientDet

Usando el método de ajuste compuesto, la arquitectura básica de EfficientNet y la arquitectura de fusión de características de características de BiFPN, llegamos por fin a tener la deseada EfficientDet

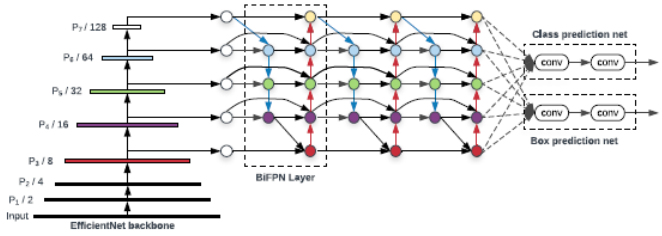


Ilustración 33 – Arquitectura EfficientDet

Como podemos ver en la Ilustración, se toma como red base EfficientNet y para clasificar las características utilizamos la arquitectura de fusión de BiFPN.

Usando el método de ajuste compuesto que se explicó anteriormente, si tomamos el coeficiente Φ, para escalar las 3 dimensiones de la red a la vez y de forma balanceada. Por último, usando las fórmulas anteriores de las Ilustraciones 22 y 20.

Se obtienen las siguientes fórmulas para escalar partiendo de EfficientNet.

$$W_{bifpn} = 64 \cdot \left(1.35^{\phi}\right), \qquad D_{bifpn} = 2 + \phi$$

Ilustración 34 – Formulas para escalar ancho y profundidad

En profundidad y ancho como se observa en la Ilustración 33.

$$R_{input} = 512 + \phi \cdot 128$$

Ilustración 35 – Fórmula para escalar resolución

Para hacerlo en resolución se ha tomado la fórmula que se ve en la ilustración 34. Así llegamos a la familia de redes neuronales de EfficientDet, las diferentes versiones salen de dar a Φ valores entre 0 y 6. De forma que las dimensiones de la red quedan de la siguiente manera:

	Input size R_{input}	Backbone Network	BiFPN #channels W_{bifpn}	BiFPN #layers D_{bifpn}	Box/class #layers D_{class}
D0 ($\phi = 0$)	512	B0	64	2	3
D1 ($\phi = 1$)	640	B1	88	3	3
D2 ($\phi = 2$)	768	B2	112	4	3
D3 ($\phi = 3$)	896	B3	160	5	4
D4 ($\phi = 4$)	1024	B4	224	6	4
D5 ($\phi = 5$)	1280	B5	288	7	4
D6 ($\phi = 6$)	1408	B6	384	8	5
D7	1536	B6	384	8	5

Ilustración 36 Familia de redes de EfficientDet

Como podemos ver, se realiza un ajuste de las dimensiones de la red eficiente y de forma balanceada

obteniendo una nueva red, que consigue resultados de eficiencia mejores además de conseguir un menor uso de los recursos y de tiempo de ejecución.

Model	mAP	#Params	Ratio	#FLOPS	Ratio	GPU LAT(ms)	Speedup	CPU LAT(s)	Speedup
EfficientDet-D0	**32.4**	**3.9M**	**1x**	**2.5B**	**1x**	**16** ±1.6	**1x**	**0.32** ±0.002	**1x**
YOLOv3 [26]	33.0	-	-	71B	28x	51†	-	-	-
EfficientDet-D1	**38.3**	**6.6M**	**1x**	**6B**	**1x**	**20** ±1.1	**1x**	**0.74** ±0.003	**1x**
MaskRCNN [8]	37.9	44.4M	6.7x	149B	25x	92†	-	-	-
RetinaNet-R50 (640) [17]	37.0	34.0M	6.7x	97B	16x	27 ±1.1	1.4x	2.8 ±0.017	3.8x
RetinaNet-R101 (640) [17]	37.9	53.0M	8x	127B	21x	34 ±0.5	1.7x	3.6 ±0.012	4.9x
EfficientDet-D2	**41.1**	**8.1M**	**1x**	**11B**	**1x**	**24** ±0.5	**1x**	**1.2** ±0.003	**1x**
RetinaNet-R50 (1024) [17]	40.1	34.0M	4.3x	248B	23x	51 ±0.9	2.0x	7.5 ±0.006	6.3x
RetinaNet-R101 (1024) [17]	41.1	53.0M	6.6x	326B	30x	65 ±0.4	2.7x	9.7 ±0.038	8.1x
NAS-FPN R-50 (640) [5]	39.9	60.3M	7.5x	141B	13x	41 ±0.6	1.7x	4.1 ±0.027	3.4x
EfficientDet-D3	**44.3**	**12.0M**	**1x**	**25B**	**1x**	**42** ±0.8	**1x**	**2.5** ±0.002	**1x**
NAS-FPN R-50 (1024) [5]	44.2	60.3M	5.1x	360B	15x	79 ±0.3	1.9x	11 ±0.063	4.4x
NAS-FPN R-50 (1280) [5]	44.8	60.3M	5.1x	563B	23x	119 ±0.9	2.8x	17 ±0.150	6.8x
EfficientDet-D4	**46.6**	**20.7M**	**1x**	**55B**	**1x**	**74** ±0.5	**1x**	**4.8** ±0.003	**1x**
NAS-FPN R50 (1280@384)	45.4	104 M	5.1x	1043B	19x	173 ±0.7	2.3x	27 ±0.056	5.6x
EfficientDet-D5 + AA	**49.8**	**33.7M**	**1x**	**136B**	**1x**	**141** ±2.1	**1x**	**11** ±0.002	**1x**
AmoebaNet+ NAS-FPN + AA(1280) [37]	48.6	185M	5.5x	1317B	9.7x	259 ±1.2	1.8x	38 ±0.084	3.5x
EfficientDet-D6 + AA	**50.6**	**51.9M**	**1x**	**227B**	**1x**	**190** ±1.1	**1x**	**16**±0.003	**1x**
AmoebaNet+ NAS-FPN + AA(1536) [37]	50.7	209M	4.0x	3045B	13x	608 ±1.4	3.2x	83 ±0.092	5.2x
EfficientDet-D7 + AA	**51.0**	**51.9M**	**1x**	**326B**	**1x**	**262** ±2.2	**1x**	**24** ±0.003	**1x**

Ilustración 37 – Precisión de EfficientDet

2.3.5 Conclusión del estudio teórico

En base a la teoría expuesta anteriormente, se podría afirmar que EfficientDet es una red precisa y que requiere menos parámetros que otras redes neuronales similares, ya que usa menos potencia computacional para hacer los cálculos.

En la siguiente sección aplicaremos los conocimientos teóricos anteriormente expuestos, para testear EfficientDet y verificar que los resultados obtenidos son tal y como hemos formulado hasta ahora.

Realizaremos dos simulaciones diferentes, primero entrenaremos EfficientDet en comparación con los resultados obtenidos por Lucía Reina López en su trabajo de *Aplicación Android y Servicio Web Spring para la detección y registro de señales de tráfico de velocidad usando Deep Learning con tiny-yolov3 y OpenCV* y en segundo lugar simularemos EfficientDet y Yolov-3 en un mismo entorno y con la misma configuración inicial y verificar que los resultados obtenidos confirman la teoría expuesta.

3 SIMULACIÓN COMPARATIVA

La simulación que se va a realizar en este trabajo final de grado consiste en tratar de obtener unos mejores resultados en cuanto a precisión y eficiencia sobre un dataset de imágenes de señales de tráfico obtenido gracias a la ayuda de María Reina López en su trabajo *Aplicación Android y Servicio Web Spring para la detección y registro de señales de tráfico de velocidad usando Deep Learning con tiny-yolov3 y OpenCV*, que nos fue cedido para poder realizar este estudio.

En este estudio se seleccionó dentro del dataset de 5000 imágenes un lote de 1000 imágenes de señales de tráfico, que se etiquetaron utilizando el software de RoboFlow, para después ser utilizado en entrenar la red y obtener los resultados deseados.

Debido a que RoboFlow no permite etiquetar más de 1000 imágenes, se limitó el dataset inicial y se seleccionaron de manera equitativa 100 imágenes de cada tipo de señal de tráfico disponible (10 tipos de señales).

En definitiva, comprobaremos de manera experimental la hipótesis planteada, si EfficientDet es mejor en cuanto a precisión y eficiencia para la detección de señales de tráfico. Si se confirma el resultado esperado, deberíamos fomentar el uso de tecnologías que sean más eficientes y no malgasten recursos computacionales innecesarios, ya que, aunque todos queremos resultados inmediatos, es relevante que estos puedan ser obtenidos de manera eficiente.

3.1 Entrenamiento EfficientDet

Para poder realizar el entrenamiento de una red neuronal, esta tiene que ser alimentada con un gran número de imágenes. Cuanta más información reciba, mejor ajustará sus parámetros en el proceso de entrenamiento, para que luego cuando sea necesario, pueda trabajar sin supervisión tomando decisiones inteligentes. En nuestra simulación el objetivo final de esta red es que pueda detectar señales de tráfico en tiempo real.

Para poder realizar este estudio se ha hecho uso de dos plataformas que ofrecen APIs, una para la correcta etiquetación de imágenes, RoboFlow y TensorFlow, librería de código abierto que nos proporciona el código necesario para ejecutar las redes neuronales. Además, hemos utilizado Google Colab, herramienta de Google que nos permite usar la GPU de un ordenador con conexión en remoto, y así poder realizar operaciones complejas mucho más rápido.

En primer lugar, se etiquetaron las imágenes distinguiendo 10 clases de señales diferentes.

Ilustración 38 – Clases de señales

En RoboFlow, se etiquetaron una a una de manera manual. Hay que marcar dentro de un recuadro la zona de la imagen que es interesante para nosotros y darle una clase determinada y así con las 1000 imágenes.

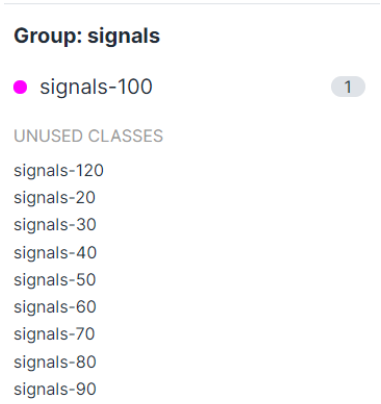


Ilustración 39- Clases en RoboFlow

Ilustración 40 – Imagen etiquetada

Tras realizar el etiquetado de todas las imágenes se genera una dataset que es exportable a otras plataformas en diferentes formatos según la red que queramos entrenar.

3.1.1 Entrenamiento en Google Colab

Gracias al uso de la plataforma de Google Colab ha sido posible ejecutar el código y desarrollar la idea expuesta en este proyecto. Al final lo que pretendemos explicar es por qué la arquitectura de EfficientDet y sobre todo el modelo de ajuste compuesto propuesto para este caso obtiene mejores resultados que Yolov3 detectando señales de tráfico.

Hemos usado un código disponible en la web de Roboflow, la cual cuenta con múltiples ayudas para que cualquier usuario pueda experimentar con las redes neuronales y entrenarlas.

3.1.1.1 Ejecución de EfficientDet con respecto al trabajo previo

Como se ha comentado anteriormente se ha utilizado la plataforma de Google Colab como entorno de ejecución. Se ha usado en este proyecto para entrenar nuestra red neuronal EfficientDet y Yolov-3 ya que podemos usar la potente 12GB NVIDIA Tesla K80 GPU durante 12h seguidas.

1. En primer lugar, hemos importado desde TensorFlow todas las librerías necesarias para llevar a cabo el entrenamiento de la red necesario.

```
import os
import pathlib

# Clone the tensorflow models repository if it doesn't alread
y exist
if "models" in pathlib.Path.cwd().parts:
  while "models" in pathlib.Path.cwd().parts:
    os.chdir('..')
elif not pathlib.Path('models').exists():
  !git clone --depth 1 https://github.com/tensorflow/models
```

   En este código se clona el repositorio de TensorFlow en el que está la estrucuta y librerías de la red.

2. Descargamos el enlace con el dataset de imágenes de RoboFlow.

```
#Downloading data from Roboflow
%cd /content
!curl -
L "https://app.roboflow.com/ds/Yu5BX8HLjP?key=VC82kha3Mh" > r
oboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

   Con el comando curl descargamos desde RoboFlow todo el dataset de señales de tráfico para poder entrenar la red.

3. Establecemos la carpeta de donde viene la información de nuestro proyecto.

```
test_record_fname = '/content/valid/signals.tfrecord'
train_record_fname = '/content/train/signals.tfrecord'
label_map_pbtxt_fname = '/content/train/signals_label_map.pbt
xt'
```

Se establecen las carpetas en las que se van a guardar las etiquetas de las imágenes y los resultados del entrenamiento.

4. Establecemos los parámetros de la red neuronal EfficientDet para que queden fijados antes de comenzar el entrenamiento.

```
##change chosen model to deploy different models available in
 the TF2 object detection zoo
MODELS_CONFIG = {
    'efficientdet-d0': {
        'model_name': 'efficientdet_d0_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d0_512x512_co
co17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d0_coco17_tpu-
32.tar.gz',
        'batch_size': 8
    },
  'efficientdet-d1': {
        'model_name': 'efficientdet_d1_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d1_640x640_co
co17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d1_coco17_tpu-
32.tar.gz',
        'batch_size': 16
    },
    'efficientdet-d2': {
        'model_name': 'efficientdet_d2_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d2_768x768_co
co17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d2_coco17_tpu-
32.tar.gz',
        'batch_size': 16
    },
        'efficientdet-d3': {
        'model_name': 'efficientdet_d3_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d3_896x896_co
co17_tpu-32.config',
        'pretrained_checkpoint': 'efficientdet_d3_coco17_tpu-
32.tar.gz',
        'batch_size': 16
    }
}

#in this tutorial we implement the lightweight, smallest stat
e of the art efficientdet model
#if you want to scale up tot larger efficientdet models you w
ill likely need more compute!
chosen_model = 'efficientdet-d0'
```

```
num_steps = 30000 #The more steps, the longer the training. I
ncrease if your loss function is still decreasing and validat
ion metrics are increasing.
num_eval_steps = 500 #Perform evaluation after so many steps

model_name = MODELS_CONFIG[chosen_model]['model_name']
pretrained_checkpoint = MODELS_CONFIG[chosen_model]['pretrain
ed_checkpoint']
base_pipeline_file = MODELS_CONFIG[chosen_model]['base_pipeli
ne_file']
batch_size = MODELS_CONFIG[chosen_model]['batch_size'] #if yo
u can fit a large batch in memory, it may speed up your train
ing
```

En este fragmento de código, se establecen los valores de configuración de la red neuronal. Elegimos EffientDet-0 como red base, además de seleccionar el batch_size a 8, para reducir el tiempo de entrenamiento.

5. Por último, se llevó a cabo el entrenamiento, con lo parámetros que hemos configurado en el punto anterior.

```
!python /content/models/research/object_detection/model_main_
tf2.py \
    --pipeline_config_path={pipeline_file} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --sample_1_of_n_eval_examples=1 \
    --num_eval_steps={num_eval_steps}
```

Tras haber realizado la configuración pasamos a entrenar la red, la cual estuvo entrenándose alrededor de 3horas.

```
 'learning_rate': 0.07899241}
INFO:tensorflow:Step 23900 per-step time 0.548s
I0701 20:47:02.185724 139632468211584 model_lib_v2.py:700] Step 23900 per-step time 0.548s
INFO:tensorflow:{'Loss/classification_loss': 0.26493952,
 'Loss/localization_loss': 0.026256211,
 'Loss/regularization_loss': 0.060006697,
 'Loss/total_loss': 0.35120243,
 'learning_rate': 0.078982964}
I0701 20:47:02.186014 139632468211584 model_lib_v2.py:701] {'Loss/classification_loss': 0.26493952,
 'Loss/localization_loss': 0.026256211,
 'Loss/regularization_loss': 0.060006697,
 'Loss/total_loss': 0.35120243,
 'learning_rate': 0.078982964}
```

Ilustración 41 – Resultado del entrenamiento

En esta ilustración vemos el resultado final de haber entrenado la red durante 3 horas aproximadamente que llevó de tiempo de ejecución, analizando los resultados podemos decir lo siguiente.

Tras 23900 iteraciones., tenemos que resaltar los siguientes parámetros importantes.

- Regularization loss: Es la función de perdida media que tiene un valor de 0.06000697 el cual es un valor muy positivo y se encuentra por debajo del límite de pérdidas de 0.060730 por el cual habría que dejar de entrenar.
- Total loss: es la suma de todas las pérdidas durante las 23900 iteraciones realizadas para entrenar el modelo.

En definitiva, si miramos los resultados a nivel de perdida obtenidos en el trabajo realizado por Lucia Reina López se obtuvo un dato de perdida más grande.

> *0.101819 avg*, es la Función de Pérdida (Loss Function), que debe ser lo más bajo posible. Como regla general, una vez que llegue a menos de 0.060730 avg, podemos dejar de entrenar, que es lo que hemos hecho en este caso.

Ilustración 42 – Resultados de Lucía

Como se ve en esta Ilustración 42 nuestra pérdida fue 0.06000697 < 0.101819, por lo que la precisión obtenida por nuestro modelo es mayor y mejora la precisión obtenida en el trabajo de Lucía Reina López incluso con un dataset de entrenamiento menor que el usado con Yolov3 tal y como se vaticinaba en el estudio teórico previo.

Para realizar la validación de los resultados comparativos de EfficientDet y Yolov-3, vamos a realizar un segundo experimento en el que vamos a simular ambas redes a la vez y con la misma configuración inicial, para añadir más datos y que los resultados obtenidos hasta ahora puedan ser confirmados.

4 VALIDACIÓN DE RESULTADOS

Para verificar la congruencia de los resultados obtenidos y sacar unas conclusiones más contrastadas, aparte de entrenar EfficientDet y compararlo con los resultados obtenidos en el trabajo de Lucía Reina López, se ha decidido hacer un nuevo experimento, en el cual se van a entrenar la red neuronal EfficientDet y la red neuronal Yolov-3 de manera simultánea y con la misma configuración inicial y así confirmar que EfficientDet es más eficiente para detectar señales de tráfico.

En estas simulaciones se verá la ejecución del código necesario para medir el tiempo de respuesta comparativo entre ambas redes, el tiempo de entrenamiento con un mismo número de parámetros de entrada y la precisión media obtenida.

4.1 Simulación de EfficientDet vs Yolov-3

El objetivo de hacer estas nuevas simulaciones es verificar que, con las mismas condiciones iniciales, los resultados que se obtengan sean lo más fieles posibles a la teoría expuesta durante este trabajo.

En estas simulaciones escogimos dos modelos, uno de EfficientDet y otro de Yolov-3, preparados de forma que tengan la misma configuración inicial. Cada red tiene su arquitectura propia y sus características, pero realizarán las mismas iteraciones y con el mismo dataset de 1000 imágenes de señales de tráfico, para poder así obtener unos resultados adecuados.

1. En primer lugar, preparamos el entorno para EfficientDet y para Yolov3 de manera que tengamos disponible las librerías y pesos necesarios para empezar el entrenamiento.

```
#first set up for EfficientDet
#our fork of the Tessellate-Imaging image detection library
#!rm -rf Monk_Object_Detection
! git clone https://github.com/roboflow-
ai/Monk_Object_Detection.git
# For colab use the command below
# Set up library requirments
! cd Monk_Object_Detection/3_mxrcnn/installation && cat requiremen
ts_colab.txt | xargs -n 1 -L 1 pip install
#fixed version of tqdm output for Colab
!pip install --
force https://github.com/chengs/tqdm/archive/colab.zip
#IGNORE restart runtime warning, it is indeed installed
#missing a few extra packages that we will need later!
!pip install efficientnet_pytorch
!pip install tensorboardX
```

En este código, clonamos el repositorio de GitHub que contiene la arquitectura de la red de EfficientDet e instalamos las dependencias de TensorFlow para el entrenamiento de la red.

```
#second set up for YOLOv3
import os
import torch
from IPython.display import Image, clear_output
print('PyTorch %s %s' % (torch.__version__, torch.cuda.get_device_
properties(0) if torch.cuda.is_available() else 'CPU'))
!git clone https://github.com/roboflow-ai/yolov3  # clone
```

De igual manera que con EfficientDet, clonamos del repositorio de GitHub la arquitectura y dependencias de Yolov-3 necesarias para tener disponible en nuestro entorno de ejecución.

2. . Descargamos las imágenes de RoboFlow, para que estén disponibles para nuestras redes.

```
#input your COCO json link here
!mkdir ed_data
%cd ed_data
!curl -
L "https://app.roboflow.com/ds/5coCYP3JKa?key=HNs1N0PWx
B" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip

#we'll download yolo data into content file directory
      %cd ..
!curl -   "https://app.roboflow.com/ds/lQcDSTcYb9?key=O
JnxQ5yGeC" > roboflow.zip; unzip roboflow.zip; rm robof
low.zip
```

Descargamos el dataset de imágenes de señales de tráfico creado con RoboFlow para EfficientDet y para Yolov-3.

3. A continuación, empezamos el entrenamiento de EfficientDet.

```
#training YOLOv3 model
#on Tesla P100-PCIE-16GB
#based on previous experiments on our validaiton set we think t
he model converges completely in 100 epochs
!python3 train.py --data data/roboflow.data --epochs 40
```

Después de realizar el entrenamiento de EfficientDet, obtenemos el primer resultado importante, el tiempo de entrenamiento de EfficientDet fue de 0.664 horas, o lo que es lo mismo 37 minutos. Proseguiremos las simulaciones, para luego interpretar los resultados de manera conjunta.

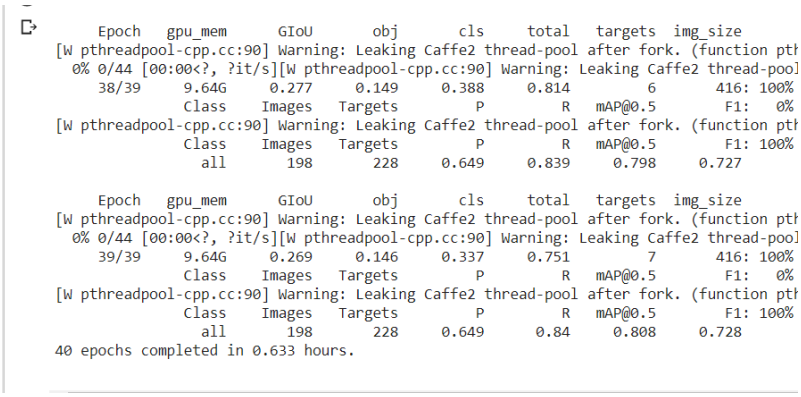


Ilustración 43 – Resultado entrenamiento EfficientDet

4. A continuación, hicimos lo mismo con Yolov-3, iniciamos el entrenamiento de la red.

```
#training YOLOv3 model
#on Tesla P100-PCIE-16GB
#based on previous experiments on our validaiton set we think t
he model converges completely in 100 epochs
!python3 train.py --data data/roboflow.data --epochs 40
```

Código que da comienzo al entrenamiento de la red neuronal Yolov-3, en la siguiente ilustración podemos observar el resultado del entrenamiento.

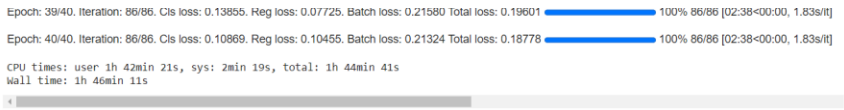


Ilustración 44 – Resultados entrenamiento Yolov3

En la Ilustración 43, se observa la salida de la ejecución del anterior código, el tiempo de entrenamiento de Yolov.3 fue de 1h y 44 minutos.

4.1.1 Análisis de tiempo de entrenamiento

Después de realizar el entrenamiento de Yolov-3 y EfficientDet tenemos que comparar el tiempo que tardó cada red en completar el entrenamiento completo.

Para el mismo número de imágenes de entrata (dataset), el tiempo que tarda en realizar el entrenamiento EfficientDet es mucho menor que Yolov3. Teniendo un mismo número de iteraciones, 40 en ambos casos, EfficieDet tardó 37 minutos, mientras que Yolov3 tardó en realizar el entrenamiento 1 hora y 46 minutos, lo que viene a ser un aumento de un 186% en tiempo de entrenamiento. Esto supone muchos más recursos computacionales usados, con lo cual confirmamos que EfficientDet requiere menos tiempo de entrenamiento que Yolov3.

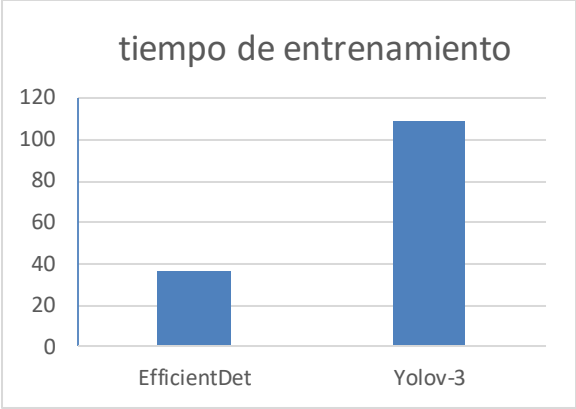


Figure 1

El tiempo de entrenamiento es un parámetro muy a tener en cuenta a la hora de escoger una red, pues el gasto de recursos utilizados en entrenar durante más tiempo la red neuronal es bastante mayor en Yolov-3 con respecto a EfficientDet.

4.1.2 Tiempo de respuesta

Una vez entrenadas ambas redes, vamos a obtener los resultados acerca del tiempo de respuesta. Vamos a medir cuánto tarda en detectar una señal de tráfico cada red cuando le pasamos una imagen nueva. Para ello hicimos lo siguiente.

1. A la red EfficientDet ya entrenada, le pasamos una imagen con una señal de tráfico de 80 km/hora, con el siguiente código a ejecutar.

```
2. img_path = "TrafficSignals/test/signals_00369_jpg
   .rf.99ae139e5530a25980b7acdd56a4317c.jpg";

3. duration, scores, labels, boxes = gtf.Predict(img
   _path, class_list, vis_threshold=0.2);
```

Y se obtiene el siguiente resultado.

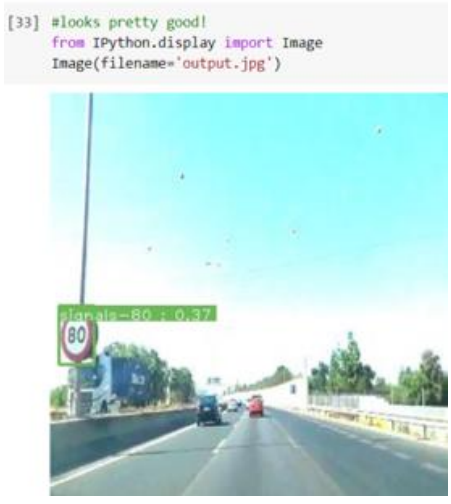


Ilustración 45 – Tiempo de respuesta

El modelo ha detectado una señal de 80 km/hora correctamente en 0.37 segundos.

2. Si realizamos la misma operación, pero con Yolov-3.

```
img_path = "TrafficSignals/test/signals_00369_jpg.rf.99ae139e5530a
25980b7acdd56a4317c.jpg";
duration, scores, labels, boxes = gtf.Predict(img_path, class_list, vis_
                                  threshold=0.2);
```

Obtenemos el resultado que se puede ver en la ilustración.

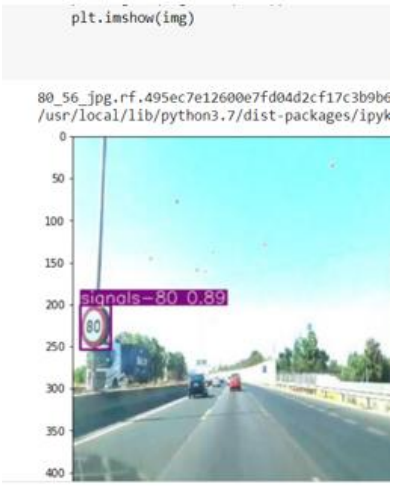


Ilustración 46 – Tiempo de respuesta Yolov-3

Se ha detectado la señal de 80km/hora en 0,89 segundos, tiempo superior al de EfficientDet.

4.1.2.1 Análisis tiempo de respuesta

Después de testear ambas redes, pasándole la misma imagen a ambas, se ha obtenido que EfficientDet tardó en detectar una señal de 80km/h en 0.37 segundos y Yolov-3 tardó 0.89 segundos.

Por tanto, EfficientDet ha sido un 140% más rápido que Yolov3 en cuanto al tiempo que tardan en clasificar una imagen.

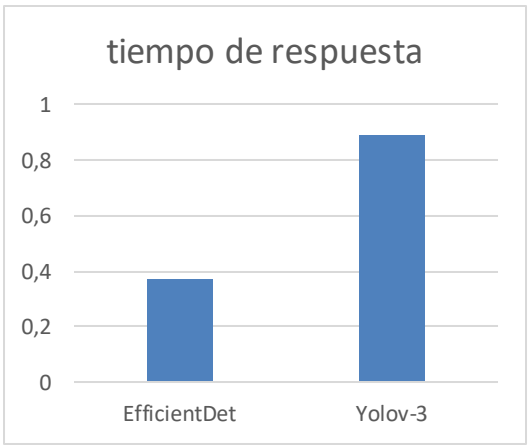


Figure 2

Este parámetro analizado es muy importante, pues si detectamos antes un objeto, en nuestro caso una señal de velocidad, el algoritmo encargado de procesar esta información podrá actuar antes, con lo cual estaríamos creando un sistema más seguro para las personas. Cuanto antes se detecten los objetos mejor.

4.1.3 Precisión

Por último, para demostrar y cerrar el eje de comparaciones entre EfficientDet y Yolov-3 se mostrarán los resultados de precisión obtenidos durante el entrenamiento. Para ello seguiremos los siguientes pasos.

1- Ejecutamos el siguiente código para obtener la precisión de cada señal durante el entrenamiento.

```
for f in os.listdir('test'):
  if f.endswith('.jpg'):
    duration, scores, labels, boxes = gtf.Predict('test/'+f,
class_list, vis_threshold=0.2);
    scores = scores.tolist()
    boxes = boxes.tolist()
    labels = labels.tolist()
    with open('mAP/input/detection-results/' + f[:-
4] + '.txt', 'w') as out_file:
      #write detection result
      for i in range(len(scores)):
        #print(scores)
        line = class_list[labels[i]] + ' ' + str(scores[i]) +
 ' ' + str(' '.join([str(j) for j in boxes[i]]))
        out_file.write(line)
        if i < len(scores) - 1:
          out_file.write('\n')
```

Se obtuvo el siguiente resultado.

```
#evaluation results for EfficientDet
!pyhton main.py -na

No back up required for /content/mAP/inp
total ground-truth backup files: 7
total intersected files: 92
Intersection completed!
66.67% = signals-100 AP
88.24% = signals-120 AP
75.00% = signals-20 AP
56.00% = signals-30 AP
80.05% = signals-40 AP
99.45% = signals-50 AP
76.67% = signals-60 AP
85.71% = signals-70 AP
65.11% = signals-80 AP
56.55% = signals-90 AP
mAP = 74.94%
<Figure size 640x480 with 1 Axes>
```

Ilustración 47 – Precisión EfficientDet

2. Si hacemos lo mismo para Yolov-3.

```
%cd ..
%cd yolov3/
!python3 detect.py --weights weights/last.pt --
source=../test/ --names=../train/roboflow_data.names --save-txt
 #reorder and write to mAP calc
 %cd ..
 %rm -rf mAP/input/detection-results/
 %mkdir mAP/input/detection-results/

 for f in os.listdir('yolov3/output/'):
  if f.endswith('txt'):
    with open('mAP/input/detection-results/' + f[:-
8] + '.txt', 'w') as out_file:
 with open('yolov3/output/' + f) as read_file:
   lines = read_file.readlines()
   for line in lines:
     line = line.split()
     out_file.write(class_list[int(line[4]) + 1] + ' ' + lin
e[5] + ' ' + ' '.join(line[0:4]))
     out_file.write('\n')
%cd mAP
#!python scripts/extra/intersect-gt-and-dr.py
!python main.py -na
```

En este fragmento de código se lee el archivo correspondiente que contiene los valores de precisión obtenidos durante el entrenamiento y se obtiene el siguiente resultado.

```
#evaluation results for Yolov3
!python main.py -na

    55.67% = signals-100 AP
    82.15% = signals-120 AP
    64.25% = signals-20 AP
    71.37% = signals-30 AP
    79.24% = signals-40 AP
    61.85% = signals-50 AP
    58.32% = signals-60 AP
    67.21% = signals-70 AP
    68.63% = signals-80 AP
    59.21% = signals-90 AP
    mAP = 66.79%
    <Figure size 640x480 with 1 Axes>
```

Ilustración 48 – Precisión de Yolov-3

4.1.3.1 Análisis de resultados de precisión

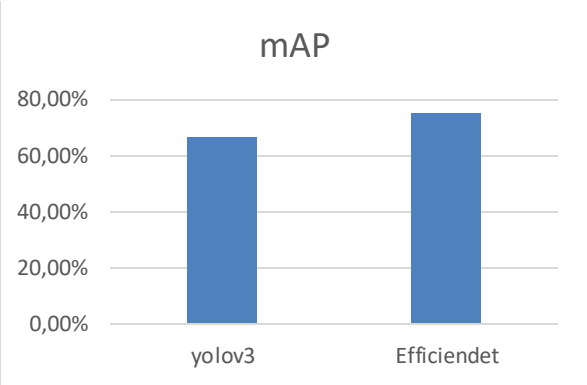
Tras observar los resultados, se puede ver que EfficientDet ha obtenido más precisión tanto

individual, si vemos los datos de cada señal por separado, como precisión media total.

Tenemos un 74,94% de precisión media en EfficientDet, frente a un 66,79 % de precisión media total en Yolov-3. Esto quiere decir que EfficientDet se va a equivocar menos que Yolov-3 a la hora de detectar una señal de tráfico.

EfficientDet va a cometer menos errores a la hora de detectar señales de tráfico, siendo más seguro que Yolov-3.

Figure 3



4.2 Reevaluación de resultados

Tras haber realizado todas las simulaciones necesarias, hemos obtenido que EfficientDet es más rápida en la detección de señales de tráfico, cualidad vital, ya que detectar señales de tráfico en menos tiempo en seguridad vial es crucial. En cuestión de milisegundos se tienen que tomar decisiones de suma importancia.

También es un sistema que requiere menos tiempo de entrenamiento, luego puede ponerse a funcionar antes y es más eficiente en el uso de recursos.

Por último y no menos importante es más preciso, ya que comete menos errores a la hora de etiquetar y detectar las señales de tráfico correspondientes.

	Tiempo de ejecución	Tiempo de respuesta	mAp
EfficientDet	37 minutos	0,37	67,79 %
Yolov-3	109 minutos	0,89	74,94 %

Figure 4

En esta tabla podemos ver a modo de resumen cómo EfficientDet es superior en todos los aspectos analizados; por lo tanto, sería más conveniente usar EfficientDet antes que Yolov-3 para la detección de señales de tráfico.

5 CONCLUSIÓN Y LÍNEAS FUTURAS

Las redes neuronales han supuesto un avance en lo que a resolución de problemas de manera automática y rápida se refiere, pues cuestiones que antes requerían de la actuación humana, ahora pueden ser realizadas por un algoritmo inteligente que funcione de manera autónoma.

EfficientDet es una red neuronal que mejora en prestaciones a otras redes neuronales y en concreto a Yolov-3, por lo que para futuras implementaciones de software para la detección de señales de tráfico en tiempo real conviene usar redes que tengan un mejor comportamiento ante estímulos externos.

La aplicación del método de ajuste compuesto para convertir otras redes neuronales en su versión eficiente supondría un importante ahorro en recursos computacionales. En lugar de crear nuevos algoritmos con mayor potencia, se podría mejorar los ya existentes ajustando las dimensiones de las redes neuronales.

Asimismo, la consecuente agilización del tiempo de uso de los equipos que albergan los algoritmos de machine learning, y por tanto la optimización de la energía consumida, conduciría a una reducción del impacto medioambiental y de la huella de carbono.

Como líneas futuras dentro del marco de este estudio, podría realizarse el entrenamiento de la red para la detección de señales de tráfico con dos datasets etiquetados de diferente manera, cruzando los resultados para obtener una mayor precisión. En primer lugar, un dataset preparado para que la red detecte la propia señal, círculo de bordes rojos con interior blanco, con otro dataset especializado en detectar los números de las señales; así, con ambos resultados de entrenamiento, se podrían tomar decisiones con mayor precisión.

Otra potencial mejora sería la de aplicar el método de ajuste compuesto a otras redes más complejas y de mayor precisión para tratar de convertirlas en su versión eficiente.

Para darle una continuación a los resultados obtenidos en este trabajo, sería muy interesante probar la aplicación en condiciones reales, ya que en este escenario podríamos demostrar la fiabilidad real de este algoritmo especializado en identificar señales de tráfico.

ANEXO A: CÓDIGO DE EJECUCIÓN DE LAS REDES NEURONALES

Código para la ejecución de EfficientDet en Google Colab

A.1 Código de EficientDet

1. Importamos desde TensorFlow las dependencias para crear EfficientDet.

```
import os
import pathlib

# Clone the tensorflow models repository if it doesn't alread
y exist
if "models" in pathlib.Path.cwd().parts:
  while "models" in pathlib.Path.cwd().parts:
    os.chdir('..')
elif not pathlib.Path('models').exists():
  !git clone --depth 1 https://github.com/tensorflow/models

# Install the Object Detection API
%%bash
cd models/research/
protoc object_detection/protos/*.proto --python_out=.
cp object_detection/packages/tf2/setup.py .
python -m pip install .
```

2. Importamos todas las librerías necesarias para poder realizar los cálculos y visualizar los resultados.

```
import matplotlib
import matplotlib.pyplot as plt

import os
import random
import io
import imageio
import glob
import scipy.misc
import numpy as np
from six import BytesIO
from PIL import Image, ImageDraw, ImageFont
from IPython.display import display, Javascript
from IPython.display import Image as IPyImage

import tensorflow as tf
```

```
from object_detection.utils import label_map_util
from object_detection.utils import config_util
from object_detection.utils import visualization_utils as viz
_utils
from object_detection.utils import colab_utils
from object_detection.builders import model_builder

%matplotlib inline
```

3. Ejecutamos el builder de testeo de la red, para ver si todo los ejecutado anteriormente fue correctamente.

```
#run model builder test
!python /content/models/research/object_detection/builders/mo
del_builder_tf2_test.py
```

4. Instalamos más dependencias para la correcta visualización de imágenes y traducir las anotaciones que hace RoboFlow en nuestro dataset.

```
def load_image_into_numpy_array(path):
  """Load an image from file into a numpy array.

  Puts image into numpy array to feed into tensorflow graph.
  Note that by convention we put it into a numpy array with s
hape
  (height, width, channels), where channels=3 for RGB.

  Args:
    path: a file path.

  Returns:
    uint8 numpy array with shape (img_height, img_width, 3)
  """
  img_data = tf.io.gfile.GFile(path, 'rb').read()
  image = Image.open(BytesIO(img_data))
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape(
      (im_height, im_width, 3)).astype(np.uint8)

def plot_detections(image_np,
                    boxes,
                    classes,
                    scores,
                    category_index,
                    figsize=(12, 16),
                    image_name=None):
  """Wrapper function to visualize detections.

  Args:
```

```
    image_np: uint8 numpy array with shape (img_height, img_w
idth, 3)
    boxes: a numpy array of shape [N, 4]
    classes: a numpy array of shape [N]. Note that class indi
ces are 1-based,
      and match the keys in the label map.
    scores: a numpy array of shape [N] or None.  If scores=No
ne, then
      this function assumes that the boxes to be plotted are
groundtruth
      boxes and plot all boxes as black with no classes or sc
ores.
    category_index: a dict containing category dictionaries (
each holding
      category index `id` and category name `name`) keyed by
category indices.
    figsize: size for the figure.
    image_name: a name for the image file.
  """
  image_np_with_annotations = image_np.copy()
  viz_utils.visualize_boxes_and_labels_on_image_array(
      image_np_with_annotations,
      boxes,
      classes,
      scores,
      category_index,
      use_normalized_coordinates=True,
      min_score_thresh=0.8)
  if image_name:
    plt.imsave(image_name, image_np_with_annotations)
  else:
    plt.imshow(image_np_with_annotations)
```

5. Descargamos nuestro dataset de RoboFlow.

```
#Downloading data from Roboflow
%cd /content
!curl -
L "https://app.roboflow.com/ds/Yu5BX8HLjP?key=VC82kha3Mh" > r
oboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

6. Configuramos el modelo de EfficientDet que vamos a entrenar.

```
##change chosen model to deploy different models available in
 the TF2 object detection zoo
MODELS_CONFIG = {
    'efficientdet-d0': {
        'model_name': 'efficientdet_d0_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d0_512x512_co
co17_tpu-8.config',
```

```
        'pretrained_checkpoint': 'efficientdet_d0_coco17_tpu-
32.tar.gz',
        'batch_size': 8
    },
  'efficientdet-d1': {
        'model_name': 'efficientdet_d1_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d1_640x640_co
co17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d1_coco17_tpu-
32.tar.gz',
        'batch_size': 16
    },
    'efficientdet-d2': {
        'model_name': 'efficientdet_d2_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d2_768x768_co
co17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d2_coco17_tpu-
32.tar.gz',
        'batch_size': 16
    },
        'efficientdet-d3': {
        'model_name': 'efficientdet_d3_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d3_896x896_co
co17_tpu-32.config',
        'pretrained_checkpoint': 'efficientdet_d3_coco17_tpu-
32.tar.gz',
        'batch_size': 16
    }
}

#in this tutorial we implement the lightweight, smallest stat
e of the art efficientdet model
#if you want to scale up tot larger efficientdet models you w
ill likely need more compute!
chosen_model = 'efficientdet-d0'

num_steps = 30000 #The more steps, the longer the training. I
ncrease if your loss function is still decreasing and validat
ion metrics are increasing.
num_eval_steps = 500 #Perform evaluation after so many steps

model_name = MODELS_CONFIG[chosen_model]['model_name']
pretrained_checkpoint = MODELS_CONFIG[chosen_model]['pretrain
ed_checkpoint']
base_pipeline_file = MODELS_CONFIG[chosen_model]['base_pipeli
ne_file']
batch_size = MODELS_CONFIG[chosen_model]['batch_size'] #if yo
u can fit a large batch in memory, it may speed up your train
ing
```

7. Descargamos los pesos preentrenados para EfficientDet.

```
#download pretrained weights
%mkdir /content/models/research/deploy/
%cd /content/models/research/deploy/
import tarfile
download_tar = 'http://download.tensorflow.org/models/object_
detection/tf2/20200711/' + pretrained_checkpoint

!wget {download_tar}
tar = tarfile.open(pretrained_checkpoint)
tar.extractall()
tar.close()
```

8. Descargamos el fichero básico de configuración de la red.

```
O#download base training configuration file
%cd /content/models/research/deploy
download_config = 'https://raw.githubusercontent.com/tensorfl
ow/models/master/research/object_detection/configs/tf2/' + ba
se_pipeline_file
!wget {download_config}
```

9. Obtenemos los datos necesarios para sobre escribir el fichero de configuración adaptado a nuestro caso.

```
#prepare
pipeline_fname = '/content/models/research/deploy/' + base_pi
peline_file
fine_tune_checkpoint = '/content/models/research/deploy/' + m
odel_name + '/checkpoint/ckpt-0'

def get_num_classes(pbtxt_fname):
    from object_detection.utils import label_map_util
    label_map = label_map_util.load_labelmap(pbtxt_fname)
    categories = label_map_util.convert_label_map_to_categori
es(
        label_map, max_num_classes=90, use_display_name=True)
    category_index = label_map_util.create_category_index(cat
egories)
    return len(category_index.keys())
num_classes = get_num_classes(label_map_pbtxt_fname)
```

10. Escribimos en el fichero de configuración los datos necesarios.

```
#write custom configuration file by slotting our dataset, mod
el checkpoint, and training parameters into the base pipeline
 file

import re
```

```
%cd /content/models/research/deploy
print('writing custom configuration file')

with open(pipeline_fname) as f:
    s = f.read()
with open('pipeline_file.config', 'w') as f:

    # fine_tune_checkpoint
    s = re.sub('fine_tune_checkpoint: ".*?"',
               'fine_tune_checkpoint: "{}"'.format(fine_tune_
checkpoint), s)

    # tfrecord files train and test.
    s = re.sub(
        '(input_path: ".*?)(PATH_TO_BE_CONFIGURED/train)(.*?"
)', 'input_path: "{}"'.format(train_record_fname), s)
    s = re.sub(
        '(input_path: ".*?)(PATH_TO_BE_CONFIGURED/val)(.*?")'
, 'input_path: "{}"'.format(test_record_fname), s)

    # label_map_path
    s = re.sub(
        'label_map_path: ".*?"', 'label_map_path: "{}"'.forma
t(label_map_pbtxt_fname), s)

    # Set training batch_size.
    s = re.sub('batch_size: [0-9]+',
               'batch_size: {}'.format(batch_size), s)

    # Set training steps, num_steps
    s = re.sub('num_steps: [0-9]+',
               'num_steps: {}'.format(num_steps), s)

    # Set number of classes num_classes.
    s = re.sub('num_classes: [0-9]+',
               'num_classes: {}'.format(num_classes), s)

    #fine-tune checkpoint type
    s = re.sub(
        'fine_tune_checkpoint_type: "classification"', 'fine_
tune_checkpoint_type: "{}"'.format('detection'), s)

    f.write(s)
```

11. Mostramos el fichero de configuración para que se vea lo que hemos configurado.

```
%cat /content/models/research/deploy/pipeline_file.config
```

12. Por último, pasamos a ejecutar el módulo que entrenará nuestra red.

```
pipeline_file = '/content/models/research/deploy/pipeline_fil
e.config'
model_dir = '/content/training/'

!python /content/models/research/object_detection/model_main_
tf2.py \
    --pipeline_config_path={pipeline_file} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --sample_1_of_n_eval_examples=1 \
    --num_eval_steps={num_eval_steps}
```

A.2 Código de ejecución de EfficientDet vs Yolov3

En esta parte veremos el código correspondiente al entrenamiento de ambas redes de manera simultánea.

1. En primer lugar, establecemos las dependencias necesarias para ambas redes.

```
#first set up for EfficientDet
#our fork of the Tessellate-Imaging image detection library
#!rm -rf Monk_Object_Detection
! git clone https://github.com/roboflow-
ai/Monk_Object_Detection.git
# For colab use the command below
# Set up library requirments
! cd Monk_Object_Detection/3_mxrcnn/installation && cat requi
rements_colab.txt | xargs -n 1 -L 1 pip install
#fixed version of tqdm output for Colab
!pip install --
force https://github.com/chengs/tqdm/archive/colab.zip
#IGNORE restart runtime warning, it is indeed installed
#missing a few extra packages that we will need later!
!pip install efficientnet_pytorch
!pip install tensorboardX


#second set up for YOLOv3
import os
import torch
from IPython.display import Image, clear_output
print('PyTorch %s %s' % (torch.__version__, torch.cuda.get_de
vice_properties(0) if torch.cuda.is_available() else 'CPU'))
!git clone https://github.com/roboflow-ai/yolov3  # clone
```

2. Descargamos el dataset de RoboFlow para EfficientDet y para RoboFlow.

```
        #input your COCO json link here
        !mkdir ed_data
        %cd ed_data
        !curl -
        L "https://app.roboflow.com/ds/5coCYP3JKa?key=HNs1N0PWxB" > r
        oboflow.zip; unzip roboflow.zip; rm roboflow.zip
        #we'll download yolo data into content file directory
        %cd ..
!curl -   "https://app.roboflow.com/ds/lQcDSTcYb9?key=OJnxQ5yGeC"
> roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

3. Establecemos la estructura de ficheros para ambas redes.

   Para EfficientDet.

```
        #set up EfficientDet folder structure
        !mkdir ed_data/Signals
        !mkdir ed_data/Signals/annotations
        !mkdir ed_data/Signals/Annotations
        !mkdir ed_data/Signals/Images

        %cp ed_data/train/_annotations.coco.json ed_data/Signals/anno
        tations/instances_Images.json
        %cp ed_data/train/*.jpg ed_data/Signals/Images/
```

   Para Yolov-3.

```
#set up yolo folder structure
%cd train
%mkdir labels
%mkdir images
%mv *.jpg ./images/
%mv *.txt ./labels/
%cd images
# create Ultralytics specific text file of training images
file = open("train_images_roboflow.txt", "w")
for root, dirs, files in os.walk("."):
    for filename in files:
      # print("../train/images/" + filename)
      if filename == "train_images_roboflow.txt":
        pass
      else:
        file.write("../train/images/" + filename + "\n")
file.close()


%cd ../../valid
%mkdir labels
%mkdir images

%mv *.jpg ./images/
```

```
%mv *.txt ./labels/
%cd images


# create Ultralytics specific text file of validation images
file = open("valid_images_roboflow.txt", "w")
for root, dirs, files in os.walk("."):
    for filename in files:
      # print("../train/images/" + filename)
      if filename == "valid_images_roboflow.txt":
        pass
      else:
        file.write("../valid/images/" + filename + "\n")
file.close()

%cd ../../yolov3/data
%cat ../../train/_darknet.labels > ../../train/roboflow_data.names

def get_num_classes(labels_file_path):
    classes = 0
    with open(labels_file_path, 'r') as f:
      for line in f:
        classes += 1
    return classes

# update the roboflow.data file with correct number of classes
import re

num_classes = get_num_classes("../../train/_darknet.labels")
with open("roboflow.data") as f:
    s = f.read()
with open("roboflow.data", 'w') as f:

    # Set number of classes num_classes.
    s = re.sub('classes=[0-9]+',
               'classes={}'.format(num_classes), s)
    f.write(s)

%cat roboflow.data
```

4. Empezamos el entrenamiento de Yolov-3.

```
#training YOLOv3 model
#on Tesla P100-PCIE-16GB
#based on previous experiments on our validaiton set we think
 the model converges completely in 100 epochs
!python3 train.py --data data/roboflow.data --epochs 40
```

5. Empezamos el entrenamiento de EfficientDet.

```
        import os
        import sys
        sys.path.append("Monk_Object_Detection/4_efficientdet/lib/");
        from train_detector import Detector
        gtf = Detector();

        #directs the model towards file structure
        root_dir = "./";
        coco_dir = "ed_data/Chess";
        img_dir = "./";
        set_dir = "Images";
gtf.Train_Dataset(root_dir, coco_dir, img_dir, set_dir, batch_size
=8, image_size=512, use_gpu=True)
gtf.Model();
%%time
gtf.Set_Hyperparams(lr=0.0001, val_interval=1, es_min_delta=0.0, e
s_patience=0)
gtf.Train(num_epochs=40, model_output_dir="trained/");
```

6. A continuación, encontramos el código que nos muestra el tiempo de respuesta de cada red neuronal, ante una imagen nueva.

```
        ##first inference with effficientDet
        import os
        import sys
        sys.path.append("Monk_Object_Detection/4_efficientdet/lib/");
        from infer_detector import Infer
        gtf = Infer();
        #our trained model weights are in here in onxx format
        gtf.Model(model_dir="trained/")
        #extract class list from our annotations
        import json
        with open('ed_data/train/_annotations.coco.json') as json_fil
        e:
            data = json.load(json_file)
        class_list = []
        for category in data['categories']:
          class_list.append(category['name'])
```

7. Le pasamos una imagen a nuestra red EfficentDet Entrenada.

```
        img_path = "TrafficSignals/test/signals_00369_jpg.rf.99ae139e
        5530a25980b7acdd56a4317c.jpg";
        duration, scores, labels, boxes = gtf.Predict(img_path, class
        _list, vis_threshold=0.2);
```

8. Mostramos los resultados obtenidos.

```
        from IPython.display import Image
        Image(filename='output.jpg')
```

9. Se hace lo mismo para Yolov-3.

```
#now inference with YoloV3
#inference time is defined within detect.py
!python3 detect.py --weights weights/last.pt --
source=../test/signals_00369_jpg.rf.99ae139e5530a25980b7acdd5
6a4317c.jpg --names=../train/roboflow_data.names
```

10. Importamos las librerías para poder enseñar la imagen y los resultados.

```
# import libraries for display
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, Image
from glob import glob
import random
import PIL
# plot just one random image prediction
filename = random.choice(os.listdir('./output'))
print(filename)
Image('./output/' + filename)

# grab all images from our output directory
images = [ PIL.Image.open(f) for f in glob('./output/*') ]
# convert images to numPy
def img2array(im):
    if im.mode != 'RGB':
        im = im.convert(mode='RGB')
    return np.fromstring(im.tobytes(), dtype='uint8').reshape
((im.size[1], im.size[0], 3))

# create array of numPy images
np_images = [ img2array(im) for im in images ]

# plot ALL results in test directory (NOTE: adjust figsize as
 you please)
for img in np_images:
    plt.figure(figsize=(8, 6))
    plt.imshow(img)
```

A continuación, podemos ver el código para medir la precisión.

11. Clonamos el repositorio necesario para medir la precisión en las redes.

```
%cd /content/
!git clone https://github.com/Cartucho/mAP
%rm -rf mAP/input/ground-truth/
%mkdir mAP/input/ground-truth/
%mkdir mAP/input/images
%mkdir mAP/input/images-optional/
%cp test/*txt mAP/input/ground-truth/
```

```
%cp test/*jpg mAP/input/images-optional/
%cp test/*jpg mAP/input/images/
#%ls test
%cp test/*jpg mAP/input/images-optional/
%cd mAP/scripts/extra/
!python convert_gt_yolo.py
%cd /content/mAP
%cat input/ground-
truth/1c0060ef868bdc326ce5e6389cb6732f_jpg.rf.a07af6147d4a79376c18
2d2d95c639ec.txt
%rm -rf mAP/input/detection-results/
%mkdir mAP/input/detection-results/
```

12. Código para obtener la precisión de EfficientDet.

```
for f in os.listdir('test'):
  if f.endswith('.jpg'):
    duration, scores, labels, boxes = gtf.Predict('test/'+f,
class_list, vis_threshold=0.2);
    scores = scores.tolist()
    boxes = boxes.tolist()
    labels = labels.tolist()
    with open('mAP/input/detection-results/' + f[:-
4] + '.txt', 'w') as out_file:
      #write detection result
      for i in range(len(scores)):
        #print(scores)
        line = class_list[labels[i]] + ' ' + str(scores[i]) +
 ' ' + str(' '.join([str(j) for j in boxes[i]]))
        out_file.write(line)
        if i < len(scores) - 1:
          out_file.write('\n')
```

13. Código para enseñar los resultados.

```
%cd mAP/
#evaluation results for EfficientDet
!python main.py -na
```

14. Si hacemos lo mismo para Yolov-3.

```
%cd ..
%cd yolov3/
!python3 detect.py --weights weights/last.pt --
source=../test/ --names=../train/roboflow_data.names --save-txt
#reorder and write to mAP calc
%cd ..
%rm -rf mAP/input/detection-results/
%mkdir mAP/input/detection-results/

for f in os.listdir('yolov3/output/'):
```

```
  if f.endswith('txt'):
    with open('mAP/input/detection-results/' + f[:-
8] + '.txt', 'w') as out_file:
      with open('yolov3/output/' + f) as read_file:
        lines = read_file.readlines()
        for line in lines:
          line = line.split()
          out_file.write(class_list[int(line[4]) + 1] + ' ' + line
[5] + ' ' + ' '.join(line[0:4]))
          out_file.write('\n')
%cd mAP
#!python scripts/extra/intersect-gt-and-dr.py
!python main.py -na
```

REFERENCIAS

[1] S. Mellado Contigioso, *Aplicación Android y Servicio Web Spring para la monitorización de datos obtenidos en un vehículo haciendo uso de la plataforma FIWARE,* Sevilla: Universidad de Sevilla, 2020

[2] Lucia Reina Lopez, *Aplicación Android y Servicio Web Spring para la detección y registro de señales de tráfico de velocidad usando Deep Learning con tiny-yolov3 y OpenCV,* Sevilla: Universidad de Sevilla, 2020

[3] Ángel Moreno Prieto, *Detección de Objetos con TinyYOLOv3 sobre Raspberry Pi 3,* Sevilla, Universidad de Sevilla, 2020

[4] https://github.com/AlexeyAB/darknet

[5] https://arxiv.org/pdf/1911.09070.pdf

[6] https://arxiv.org/pdf/1905.11946.pdf

[7] https://microscope.openai.com/

[8] https://towardsdatascience.com/review-fpn-feature-pyramid-network-object-detection-262fc7482610

[9] https://github.com/google/automl/master/efficientdet://github.com/google/automl/tree/master/efficientdet

[10] https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html

[11]https://colab.research.google.com/driveAX3ol7gcZ7qmKuwn8zUld524sUZ#scrollTo=D9MM2WA49zyj

[12] https://paperswithcode.com/paper/efficientdet-scalable-and-efficient-object#code

[13] https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

[14]https://jonathan-hui.medium.com/understanding-feature-pyramid-networksobjectdetectionfpn227b9106c

[15]https://playground.tensorflow.org