Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Identificación Semántica de Objetos Mediante Deep Learning

Autor: Daniel Jiménez Jiménez

Tutor: Begoña C. Arrue Ullés

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Identificación Semántica de Objetos Mediante Deep Learning

Autor:
Daniel Jiménez Jiménez

Tutor:
Begoña C. Arrue Ullés
Profesor Titular

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado: Identificación Semántica de Objetos Mediante Deep Learning

Autor: Daniel Jiménez Jiménez
Tutor: Begoña C. Arrue Ullés

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mi familia por apoyarme tanto anímica como económicamente, ya que sin ellos no podría haber llegado a donde estoy. A mis abuelos, Alfonso y Paco, que no pudieron verme llegar al final de esta etapa. A todos los profesores que me han aportado algo y me han hecho continuar todo este tiempo. Por último, a mis amigos y compañeros, los cuales han tenido que aguantar mis quejas y desvaríos a lo largo de los años.

Daniel Jiménez Jiménez
Sevilla, 2019

Resumen

La visión artificial puede llegar a ser uno de los campos más complejos de la robótica. Tener que trabajar con imágenes a la vez que se reacciona a lo que se muestra en estas conlleva una gran capacidad de procesamiento. Además, si este campo se combina con el del *machine learning*, se puede dar lugar a sistemas con una gran capacidad de reacción al entorno en el que se manejan.

Este proyecto tiene como objetivo la creación de una red neuronal para la clasificación de objetos mediante el reconocimiento de sus formas,también conocido como segmentación semántica. Para ello, se hará uso tanto de *TensorFlow* como de *DeepLab*, herramientas desarrolladas por trabajadores de *Google* para *machine learning* y segmentación semántica, respectivamente. Además, se ha optado por la creación de un set de imágenes propio, en lugar de usar uno preparado como suele ser lo habitual en esta clase de proyectos.

Abstract

Artificial vision can become one of the most complex fields of robotics. Having to work with images while reacting to what is shown in these needs a great processing capacity. In addition, if this field is combined with machine learning, it can lead to systems with a great capacity to react to the environment in which they are handled.

The main theme of this project is to develop an artificial neural network for the classification of objects by recognizing their forms, also known as semantic segmentation. To do this, use will be made of *TensorFlow* and *Deeplab*, tools developed by workers of *Google* for machine learning and semantic segmentation, respectively. In addition, we have opted for the creation of our own set of images, instead of using one prepared as usual in this kind of projects.

Índice

Notación

SLAM	Simultaneous Localization and Mapping
ANN	Artificial Neural Network
MLP	Multilayer Perceptron Model
CNN	Convolutional Neural Network
ReLU	Rectified Linear Unit
SPP	Spatial Pyramid Pooling
ASPP	Atrous Spatial Pyramid Pooling

1 Introducción

En los útlimos años, los avances en el campo de la inteligencia artificial han sido significativos. Muchos de los conceptos matemáticos en los que se basa están desarrollados desde mediados del siglo XX pero la poca capacidad de computación de los sistemas de la época lo hacía inviable. Con la capacidad de computación actual, gracias al gran aumento en la calidad de los procesadores como los de *Nvidia*, *Intel* y *AMD* o a las modificaciones en su arquitectura con el paso de los años, es posible desarrollar soluciones para diferentes problemas desde al ámbito de la inteligencia artificial.

1.1 Motivación y objetivos

En la actualidad, una gran cantidad de servicios de nuestra sociedad, principalmente en entornos industriales, han sido automatizados. El siguiente paso está siendo el volver estos procesos completamente autónomos, es decir, convertir estos procesos lo más independientes posibles de la mano de obra. Incluso sectores que se han visto menos afectados por la automatización, como pudiesen ser distintos ámbitos del sector servicios, se han visto afectados por la conversión de ciertas tareas a procesos autónomos.

Figura 1.1 Modelos de coches autónomos realizados por diversas empresas.

Los intentos de empresas como *Google* [1], *Uber* o *Tesla* en la creación de coches autónomos o *Amazon* con sus drones de reparto [2] muestran que hay un interés real en esta conversión a la autonomía total de estos servicios. Esto, a su vez, implica un mayor interés en el desarrollo de

herramientas de *machine learning*, como pueden ser *TensorFlow*, por parte de *Google*, y *PyTorch* [3], en el caso de *Facebook*, y en los cursos de formación en este campo, como los que *Amazon* ofrece a sus ingenieros [4].

Figura 1.2 Dron de reparto de *Amazon*.

Un punto importante que tienen en común muchos de estos servicios de funcionamiento autónomo es la visión. Ya sea para la identificación de piezas defectuosas en una cadena de montaje o para evitar accidentes con coches autónomos, estos sistemas autónomos deben de disponer de unos sistemas de visión e identificación lo suficientemente eficientes y robustos.

Este proyecto está motivado por estos avances en la búsqueda de la autonomía de los sistemas y por el avance que se ha ido produciendo en los algoritmos de *Deep Learning* para la clasificación e identificación de objetos. En relación a esto último, cabe destacar *DeepLab*, un modelo desarrollado por trabajadores de *Google* y basado en *TensorFlow* para la segmentación semántica, es decir, el reconocimiento visual de objetos en base a su forma.

Figura 1.3 Ejemplo de segmentación semántica.

Estos avances al ser aplicados pueden dar lugar a una serie de ventajas en diversos campos, más allá de los nombrados previamente. Uno de estos es el mapeado y localización simultáneos, o SLAM (*Simultaneous localization and mapping*) [5], ya que una mayor capacidad de identificación de objetos, sean obstáculos móviles o muros y bloqueos permanentes, y sus siluetas permitiría una mayor precisión a la hora de realizar un mapa de la zona por donde transita el vehículo o el robot móvil.

Otro entorno en el que puede producir una gran serie de ventajas y avances son los entornos industriales. La identificación de los objetos a través de sus formas serviría para la localización

de productos defectuosos en una línea de montaje e incluso podría ayudar con la detección de piezas extraviadas en la maquinaria, las cuales podrían producir un mal funcionamiento. Todo esto permitiría la existencia de una linea de montaje industrial con un funcionamiento cuasi independiente, a excepción de ciertas supervisiones menores por parte de un personal cualificado.

El mayor inconveniente de los trabajos en este campo es que necesitan de una gran precisión incluso para pruebas de entrenamiento si es que se quiere aplicar estos avances. Esto se debe a que pequeños errores en aplicación real podría tener graves consecuencias, como pueden ser accidentes en el caso de los vehículos autónomos [6] o paradas de emergencia en las líneas de producción.

Por todo esto, el punto de interés en este proyecto es la compresión y entendimiento de la estructura y el funcionamiento de la herramienta *DeepLab*, la cuál puede ser la base de muchos de los proyectos que busquen la implementación de redes neuronales en el campo de la visión artificial. Esta conclusión es debida a que gran parte de los frameworks de este proyectos han sido desarrollados por trabajadores de *Google*, lo que muestra el gran interés de la compañía en el campo del *machine learning* y la visión artificial.

El objetivo final de este proyecto consiste en crear una red neuronal para la identificación de una diversa variedad de objetos a partir de las formas representadas en las imágenes que le serán cedidas a la red. A su vez, se quiere desarrollar un conjunto de imágenes de entrenamiento propio, aunque lo habitual es optar por usar uno ya predefinido.

1.2 Estructura del trabajo

El proyecto se ha estructurado en 6 capítulos. Empezaremos con una breve introducción al proyecto, dónde quedará expuesto el desarrollo de este, y se introducirán los conceptos básicos usados en el desarrollo del proyecto, que en este caso son las redes neuronales. Posteriormente, se hablará sobre el software utilizado y se verán algunos de sus rasgos fundamentales.

Una vez terminada la parte introductoria, se realizará una explicación de los elementos que se han utilizado para llegar a la resolución final. Entre estas hay que destacar el etiquetado de imágenes, y la organización y construcción del modelo.

A continuación, se pasará a presentar los resultados, los cuales se han dividido en 4 bloques. El primero corresponderá con los datos de la red después de ser entrenada. Los siguientes bloques serán los relacionados con las pruebas, en los cuales, se ha enfrentado la red a distintos escenarios con características distintas.

Por último, se expondrán las posibles mejoras a realizar sobre este proyecto, ya sea en rendimiento o funcionalidad. Además, se ha añadido una reflexión final, que puede servir de introducción sobre posibles trabajos futuros.

2 Descripción del problema

El objetivo de este proyecto es crear una red neuronal profunda capaz de poder identificar una serie de objetos en específico a través de diferentes imágenes, independientemente la posición y el tamaño del objeto.

En este proceso, se pueden diferenciar varias fases:

1. Toma de muestras y etiquetado de imágenes.
2. Creación y configuración del entorno de trabajo.
3. Entrenamiento y seguimiento de resultados.

En la primera fase, se ha obtenido una base amplia de imágenes (de alrededor de 1000 imágenes) para poder un correcto entrenamiento de la red neuronal. Para ello, posteriormente, se ha realizado un etiquetado, de forma manual, en todas las imágenes, marcando los objetos que se han de detectar con la red neuronal.

Para la segunda fase se han tomado las imágenes habilitadas para el entrenamiento y otra serie de datos para poder generar espacio de trabajo para el entrenamiento de la red neuronal. Además, se agrega la información necesaria para que dicho entorno no genere ningún tipo de incompatibilidad.

En la parte final, con todo ya configurado, se ha realizado el proceso de entrenamiento. Tras esto, se evalúan y visualizan los resultados producidos tras el entrenamiento.

2.1 Redes Neuronales

2.1.1 Conceptos básicos

Para poder hablar de redes neuronales [7], se debe de entender primero que es una neurona en el campo de la inteligencia artificial. Una neurona artificial es la unidad básica de procesamiento de información de una red neuronal artificial y que está basado en el funcionamiento de una neurona biológica. Las neuronas artificiales, al igual que sus homólogas biológicas, tienen conexiones de entrada (dendritas), conexiones de salida (axones) y un proceso interno que genera una señas de salida en respuesta a una señal de entrada.

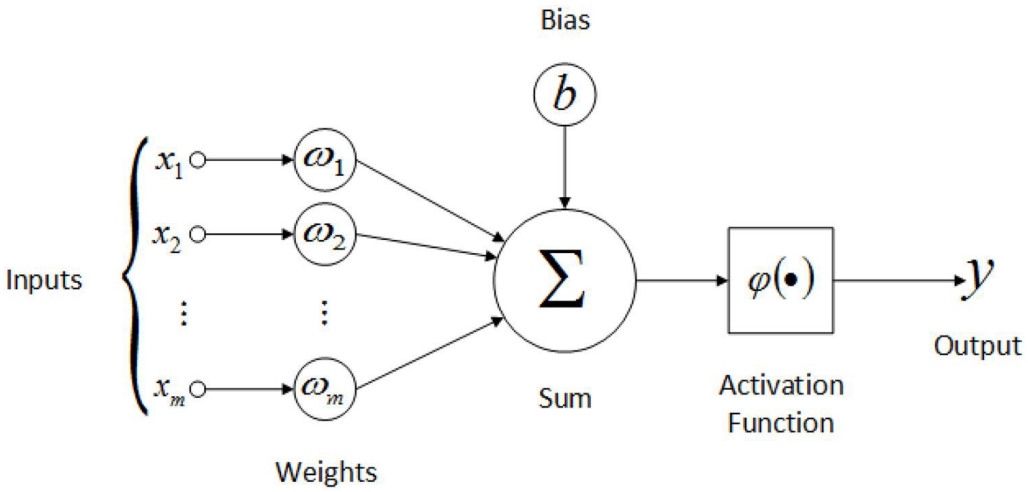


Figura 2.1 Esquema de la estructura de una neurona artificial.

Las neuronas artificiales están formadas por:

- Señales de entrada (x_1, x_2, ..., x_m) o información de entrada, que puede provenir de la salida de otras neuronas.
- Un conjunto de pesos (w_{k1}, w_{k2}, ..., w_{km}), los cuáles describen a la conexión entre fuerzas; esto puede ser positivo, representando uniones, o negativo, inhabilitando la activación de la neurona. Cuando no hay conexión entre dos neuronas, el peso sináptico es nulo.
- Bias (b_k) o sesgo, un parámetro externo a la neurona, cuya función es evitar la aparición de errores cuando se recibe una entrada nula.
- Función suma (Σ) o de entrada, que representa el sumatorio de las señales de entrada multiplicadas por sus respectivos pesos.
- Función de activación ($\varphi(.)$), que restringe la amplitud, o importancia, de la neurona, en un intervalo normalizado entre [0; 1] o [-1; 1].
- Señal de salida (y_k), que es el resultado generado por la neurona.

Las señales de entrada de las neuronas se multiplica su peso asociado, también llamado peso sináptico, y la suma de estos resultados añadido al bias da lugar a la información de entrada a la neurona:

$$u_k = \sum_{j=1}^{n} w_{kj} \cdot x_j + b_k$$

El modelo más común de ANN, es el modelo multicapa (*fully connected Multilayer Perceptron Model* o modelo MLP completamente conectado)[8] . La estructura de dicho modelo consiste en:

- Capa de entrada, las neuronas que reciben la información externa.
- Capa de salida, las neuronas que generan el resultado de la red neuronal.
- Capas intermedias u ocultas, ubicadas entre la capa de entrada y la de salida.

Basándonos en las operaciones para obtener la información de entrada a una neurona, el modelo matemático de una de las capas (L) de este tipo de ANN sería el siguiente:

$$y^L = \varphi\left(\begin{bmatrix} w_{0,0} & \cdots & w_{0,n} \\ \vdots & \ddots & \vdots \\ w_{k,0} & \cdots & w_{k,n} \end{bmatrix}\begin{bmatrix} x_0^{(L-1)} \\ \vdots \\ x_n^{(L-1)} \end{bmatrix} + \begin{bmatrix} b_0^{(L)} \\ \vdots \\ b_n^{(L)} \end{bmatrix}\right)$$

siendo k las neuronas capa anterior y n las neuronas de la capa actual. Lo que significa que $w_{k,n}$ sería el peso en la neurona k de la señal de entrada proveniente de la neurona n de la capa anterior.

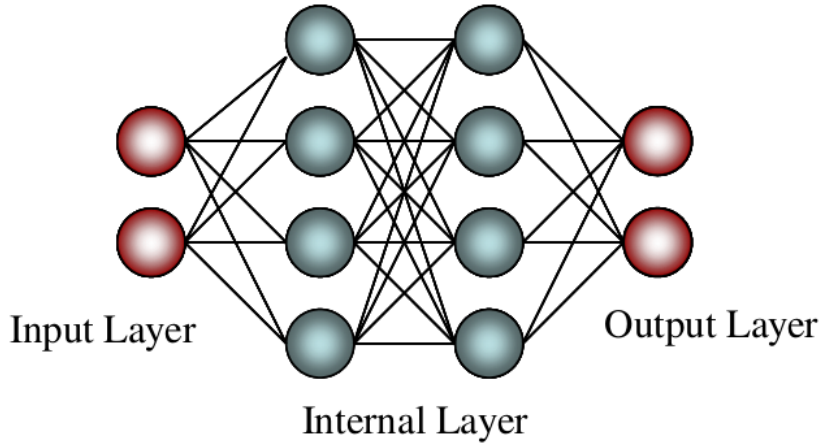


Figura 2.2 Estructura del modelo MLP para ANN.

El objetivo de estas redes es, mediante los pesos sinápticos y los sesgos, poder obtener el resultado deseado. Para ello, las capas internas se han de ajustar progresivamente para conseguir que, con determinadas salidas, se obtengan unos datos que nos den el resultado esperado en la capa de salida.

Para poder modificar pesos y bias, las redes deben de ser entrenadas. Existen diversos tipos de entrenamiento [9]:

- Aprendizaje supervisado: se le dan al modelo las respuestas correctas a las entradas dadas. Los problemas más comunes para los que se suele usar son los de clasificación.
- Aprendizaje sin supervisar: según el problema al que nos enfrentemos, es posible no tener salidas definidas para el modelo, por lo que se ha de trabajar solo con entradas. Esto hace que el modelo se enfoque en buscar características comunes en los datos de entrada y analizando su estructura. Una clase de problema en la que se podría aplicar este tipo de aprendizaje es el descubrimiento de anomalías en la información recibida.
- Aprendizaje semi-supervisado:como su propio nombre da a entender, consiste de un conjunto de datos de entrenamiento dónde algunos tienen asignada una salida y otros no. Es especialmente útil cuando extraer características puede resultar tedioso o complicado. Un caso en el que se podría aplicar es en las imágenes médicas, como las radiografías, para la detección de posibles aflicciones.
- Aprendizaje reforzado: es un sistema de aprendizaje basado en las recompensas. En un entorno aleatorio, se toman una serie de acciones aleatorias y, conforme nos acercamos al objetivo, se irán recibiendo recompensas. La idea de este tipo de modelo de aprendizaje consiste en la de obtener el funcionamiento más óptimo posible a través de múltiples iteraciones. El ambiente más común en el que se suele aplicar este tipo de aprendizaje es en los videojuegos.

2.1.2 Aprendizaje

La finalidad del aprendizaje es tener unas salidas en concreto a partir de una serie de entradas. Inicialmente, los pesos y los umbrales deben ser inicializados con valores aleatorios y se irán modificando para obtener los resultados deseados. Matemáticamente, la cercanía a los resultados deseados se puede realizar de distintas maneras. El método más común es hacer mínimos cuadrados. Si tenemos una red con n salidas y y una salidas a las que nos queremos aproximar, y_i, la función de coste sería:

$$c_j = \sum_{j=0}^{n-1} (y_j - y_{ij})^2$$

Los valores de esta función de coste serán más pequeños mientras más se parezcan los resultados y los objetivos que queremos obtener, y viceversa. Esta función nos permite saber como de buena es nuestra predicción.

Un vez tenemos la función de coste, se debe usar para poder modificar los parámetros, es decir, los pesos y umbrales. Las salidas obtenidas dependen matemáticamente de dichos parámetros, como se ha dado a entender anteriormente. Esto implica que la función de costes se puede escribir en función de estos valores, es decir, $C(w,b)$. EL siguiente paso es obtener los parámetros que nos hagan obtener en la función de coste el menor resultado posible. Una posible forma para minimizar el coste sería con la realización de las derivadas parciales de cada uno de los parámetros. De esta manera, se quiere obtener de la forma más directa el mínimo local de la función de coste al ir modificándose sus parámetros. El vector correspondiente a nuestros valores para pesos y umbrales, v, se verá modificado para llegar a este mínimo.

$$v = v - \nabla C(w,b)$$

Una forma de comprender cuan importante es el efecto de cada uno de estos parámetros en la solución es usar la derivada parcial de este y, si el valor es elevado, el efecto que tiene el parámetro en la red es relevante.

Para modificar cada uno de estos valores, se definen ciertos parámetros que relacionarán todo valor del que dependa la neurona.

$$u_k^{(L)} = \sum_{j=1}^{n} w_{kj}^{(L)} \cdot x_j^{(L-1)} + b_k^{(L)}$$

$$x_k^{(L)} = g(u_k^{(L)})$$

Empezando por las salidas se conoce cuáles son los errores cometidos en cada una de éstas. Dichas salidas dependerán de los resultados de la capa anterior, de la configuración de los pesos que conectan las capas entre sí y el umbral correspondiente a cada neurona. Cada uno de estos parámetros tiene una influencia diferente en el resultado, por lo que, analizando cada una de las salidas, se tendrá una serie de posibles cambios a los pesos y los umbrales para poder ajustar la salida. Así, se realizará la media de dichas variaciones para modificar los valores. Como en las capas finales tenemos dependencia de los parámetros de las capas anteriores, se irá modificando a partir de la capa de salida hasta llegar a la entrada. Este tipo de procedimiento se llama retropropagación, o *backpropagation*. Las derivadas que mide lo previamente descrito son:

$$\frac{\partial C_j}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C}{\partial a_j^{(L)}} \qquad \frac{\partial C_j}{\partial b_j^{(L)}} = \frac{\partial z_j^{(}L)}{\partial b_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C}{\partial a_j^{(L)}} \qquad \frac{\partial C_j}{\partial a_j^{(L-1)}} = \frac{\partial z_j^{(L)}}{\partial a_j^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C}{\partial a_j^{(L)}}$$

Haciendo uso de este procedimiento, es interesante hacer grupos de muestras para el entrenamiento y se haga una media de lo que debería cambiar los valores a nivel individual. Por ejemplo, si queremos clasificar números y solo se introduce datos de un número, la modificación de los parámetros de esta red hará que tienda a detectar dicho número. En cambio, si introducimos diferentes números a la vez, la media de los valores dará una aproximación mejor al comportamiento que se desea (clasificar números). Por lo tanto, siempre resulta en un entrenamiento más eficiente al tener un grupo de muestras de entrada diverso.

2.1.3 Redes neuronales convolucionales

Las Redes Neuronales Convolucionales (CNN) [10] son una versión regularizada del modelo MLP, es decir, una red completamente conectada (cada neurona está unida a todas las neuronas de la capa posterior) con los 3 tipos de capas nombradas anteriormente Para entender su funcionamiento [11], hay que comprender como funcionan las capas convolucionales y el *pooling*.

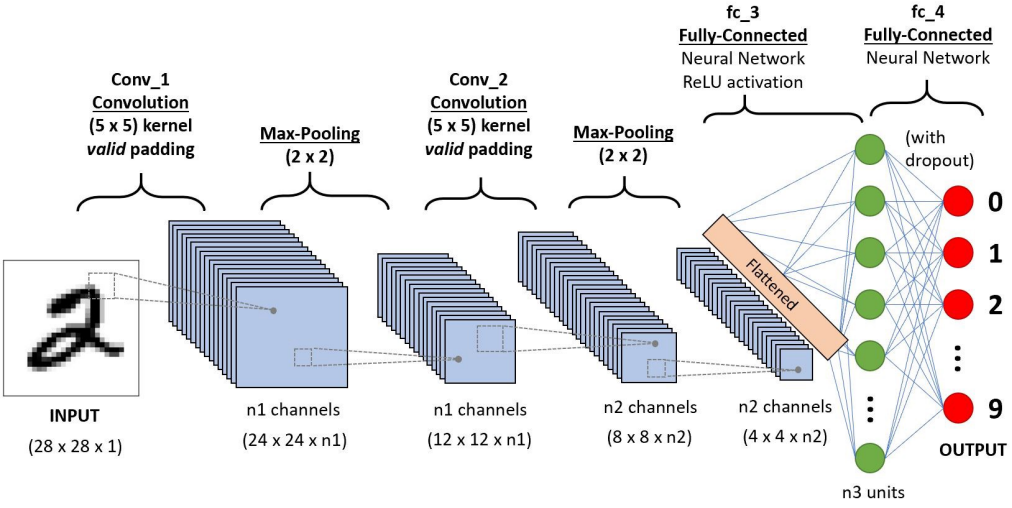


Figura 2.3 Ejemplo de la estructura de una CNN.

Un buen ejemplo de su funcionamiento es clasificar una forma en la imagen, concretamente una 'X'. El principal problema de este ejemplo es la dificultad para saber si 2 imágenes representan la misma forma con datos diferentes, es decir, discernir si imágenes distintas con rasgos similares representan lo mismo.

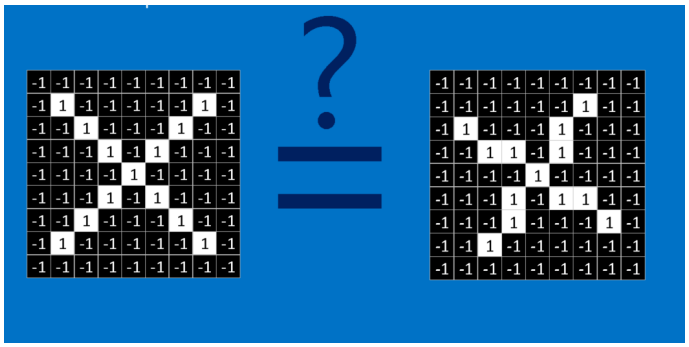


Figura 2.4 Ejemplo del problema a resolver con CNN.

2.1.3.1 Características

Las CNN comparan las imágenes parte por parte. Dichas partes son llamadas características. Cada característica funciona como una pequeña imagen, con su respectiva matriz de valores. Al encontrar coincidencias entre las características de diferentes imágenes, las CNN mejoran considerablemente, ya que se pueden encontrar patrones comunes, como podrían ser líneas diagonales para las 'X'.

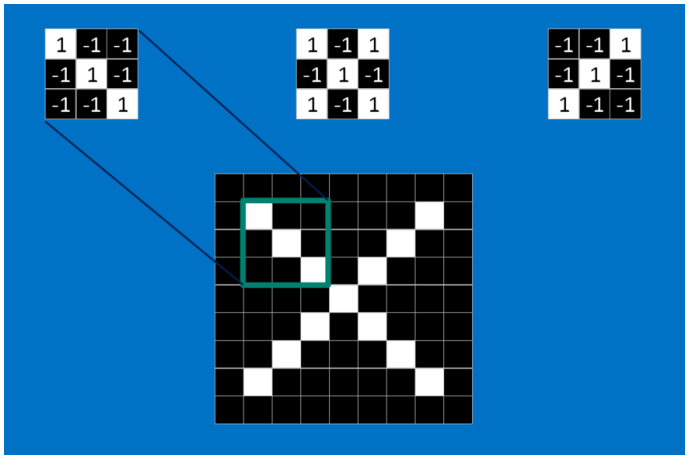


Figura 2.5 Detección de características de la 'X'.

2.1.3.2 Capas Convolucionales

La CNN no puede saber con antelación dónde se producen las similitudes por lo que intentará encontrarlas por toda la imagen. Para calcular las coincidencias de características por toda la imagen, se usan filtros. Dichos filtros son de tamaño pequeño (en el ejemplo de la imágenes, una matriz $3x3$) y se corresponden con patrones que han de ser comunes en las imágenes. Cada filtro es pasado por toda la imagen y se calcula la media en cada punto para poder hallar las regiones con mayor similitud.

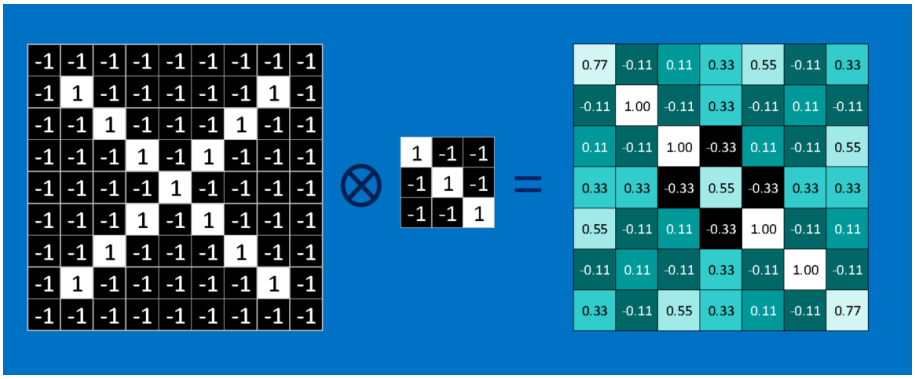


Figura 2.6 Resultados de la convolución en la imagen.

Esta operación se debe realizar con todos los filtros que se hayan creado para detectar los múltiples patrones. El proceso matemático es llamado convolución y es lo que da nombre a este tipo de redes.

2.1.3.3 Max pooling

Otras de las herramientas útiles de las CNN es el *max pooling*, que es una forma de tomar imágenes grandes y reducirlas mientras se conserva la información más importante de ellas. Consiste en pasar una pequeña ventana a través de toda la imagen y tomar el valor máximo de la ventana en cada paso.

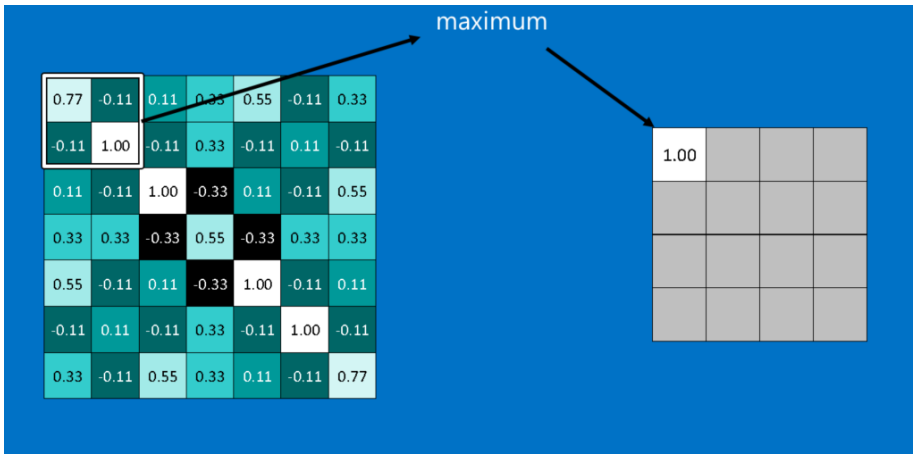


Figura 2.7 Resultados del *max pooling* en la imagen.

2.1.3.4 ReLU

Otra parte importante en este proceso es Unidad Lineal Rectificada o ReLU (*Rectified Linear Unit*). Su funcionamiento consiste en sustituir los valores negativos que puedan surgir por un 0. Esta tarea es simple pero ayuda a mantener la CNN matemáticamente saludable, evitando que los valores aprendidos se atasquen cerca del 0 o aumenten hasta el infinito.

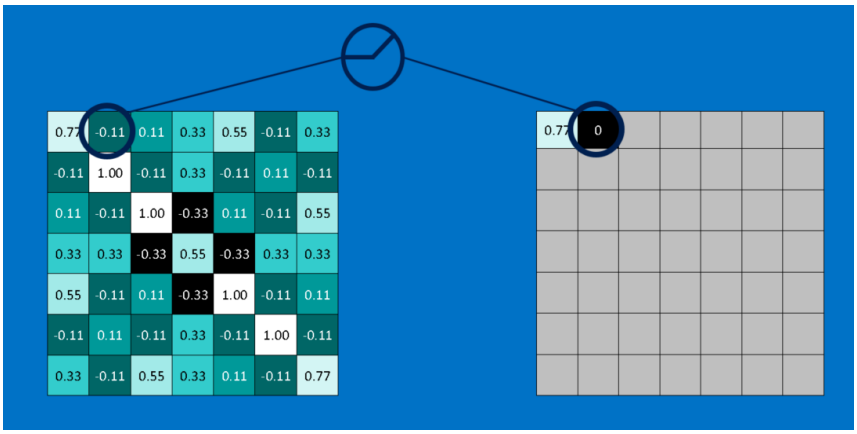


Figura 2.8 Resultados de la ReLU en la imagen.

3 Frameworks

En este capítulo nos centramos en los entornos de trabajo utilizados para la realización de este proyecto. Este abarca el uso de redes neuronales y el tratamiento de imágenes. Por tanto, la elección de herramientas adecuadas que cumplan con las necesidades del proyecto es importante.

3.1 TensorFlow & Keras

TensorFlow [12] es una plataforma de código abierto para *machine learning*. Tiene una amplia cantidad de herramientas, librerías y recursos que permite a los investigadores impulsar los estados del arte en *machine learning* y permite a los desarrolladores crear distintas aplicaciones.

3.1.1 ¿Por qué usar TensorFlow?

Al ser una plataforma de código abierto y tan usada a nivel global, *TensorFlow* tiene varias ventajas:

- Una amplia cantidad de tutoriales, que permiten comprender la plataforma y sus conceptos básicos.
- Una amplia comunidad trabajando con la plataforma, lo que da lugar a una gran cantidad de discusiones sobre las diversas tareas que se pueden realizar. Además, esto permite que se puedan resolver errores con relativa presteza.
- Documentación extensa y múltiples ejemplos que permiten entender con facilidad las funciones como los parámetros utilizados.
- Múltiples niveles de abstracción que permiten al usuario escoger el que más le convenga. La API Keras [13] permite abstraerse a los niveles más bajos, pudiendo crear proyectos de mayor complejidad de forma asequible.

3.2 DeepLab

DeepLab [14] es un un modelo de estado del arte de *deep learning* creado por *Google* para segmentación de imágenes semánticas, dónde el objetivo es el asignar etiquetas semánticas (ej: persona, perro, gato, etc) a cada píxel en la imagen de entrado.

3.2.1 ¿Cómo funciona Deeplab?

Para poder entender la arquitectura base de *DeepLab*, concretamente de su versión más reciente (*DeepLabv3+*) [15], vamos a dividir el modelo en 2 fases:

- **Fase de codificación**: el objetivo de esta fase es extraer la información esencial de la imagen. Esto será realizado por una CNN pre-entrenada, ya que, como se ha explicado anteriormente, son el mejor tipo de red para trabajar con imágenes.
- **Fase de decodificación**: la información extraída de la fase de codificación es usar para reconstruir la salida con dimensiones apropiadas.

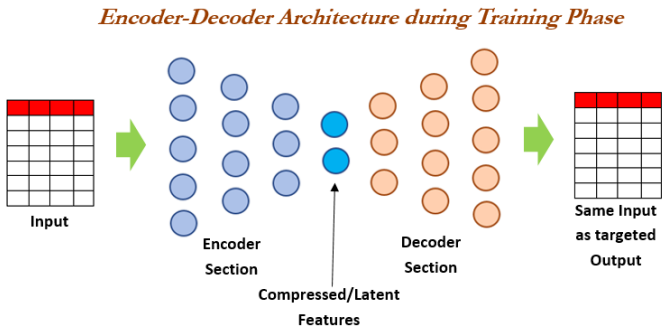


Figura 3.1 Esquema de la división en fases en *Deeplab*.

3.2.2 Convolución atrous y redes ASPP

Necesitamos un modelo robusto a los cambios de tamaño de los diferentes objetos cuando trabajamos con redes convolucionales. Esto se debe a que los objetos pequeños pueden no quedar bien representados cuando se escala las imágenes.

Este problema puede ser resuelto usando redes de agrupación piramidal de espacios o redes SPP (*Spatial Pyramid Pooling*). Estas usan diferentes versiones escaladas de la entrada para el entrenamiento y, por tanto, capturan información de las características a diferentes escalas que se combinarán después.

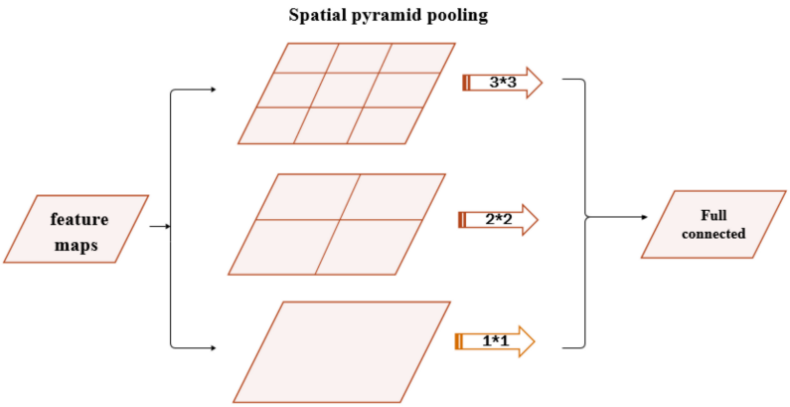


Figura 3.2 Estructura de una red SPP.

Las SPP usas varias copias de la misma arquitectura, lo que incrementa la complejidad operacional y los requerimientos de memoria para el entrenamiento. *DeepLab* introduce el concepto de la convolución *atrous*. Este tipo de convolución requiere un parámetro llamado ratio, que es usado para el control del campo de vista efectivo de la convolución. Su forma generalizada de la convolución es:

$$y_i = \sum_k x_{i+r \cdot k} \cdot w_k$$

siendo la convolución tradicional el caso en el que $r = 1$.

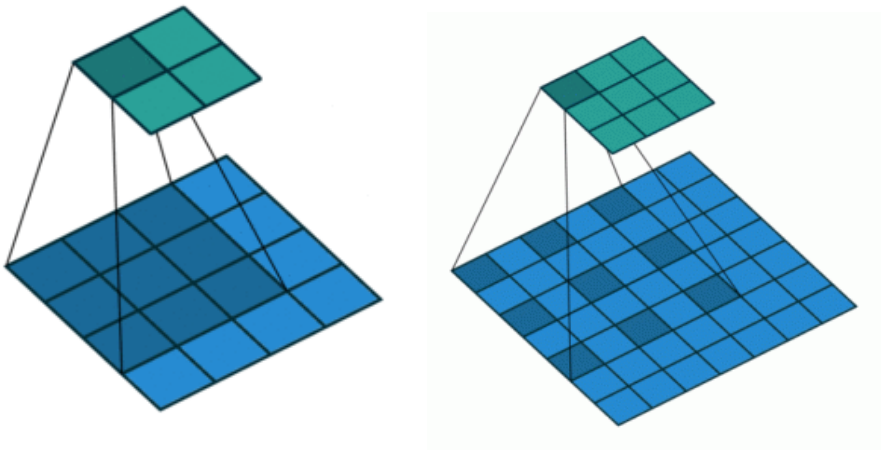


Figura 3.3 Comparación entre la convolución tradicional y la *atrous*.

Las ASPP, o *Atrous Spatial Pyramid Pooling*, nacen de que *DeepLab* combina la estructura de una SPP usando convoluciones *atrous*. La ASPP usada habitualmente consiste de 4 operaciones paralelas: una convolución 1x1 y 3 convoluciones con ratios $r = 6$, $r = 12$ y $r = 18$.

3.2.3 Convoluciones Profundas Separables

Las convoluciones profundas separables, o *depthwise separable convolutions*, son una técnica para realizar convoluciones con un menor número de operaciones que una convolución estándar. Esto implica dividir la operación de convolución en dos pasos:

- Convolución profunda (*depthwise convolution*).
- Convolución de puntos o puntiaguda (*pointwise convolution*).

Para entender este concepto mejor, se usará un ejemplo con una imagen de tamaño 12x12 y 3 canales, es decir, una entrada con dimensiones 12x12x3. El objetivo será aplicar una convolución 5x5, lo que daría una salida de 8x8x1 con una convolución normal. Necesitamos más núcleos para poder aumentar el número de canales.

En primer lugar, aplicamos una convolución con un núcleo de 5x5x1, dándonos una salida de 8x8x3.

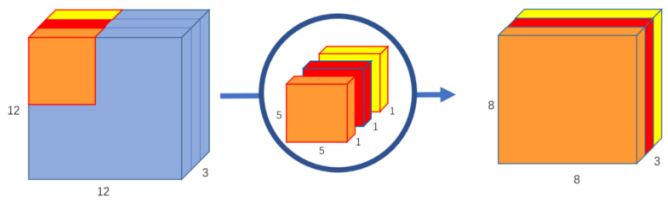


Figura 3.4 Funcionamiento de la convolución profunda..

Después, deseamos aumentar el número de canales. Para ello, usamos núcleos 1x1 con la misma profundidad de la imagen de entrada, 3 en este caso. Está convolución nos da una salida de tamaño 8x8x1, a la cuál podremos aplicar tantas convoluciones 1x1x3 como sea necesarias para aumentar el número de canales.

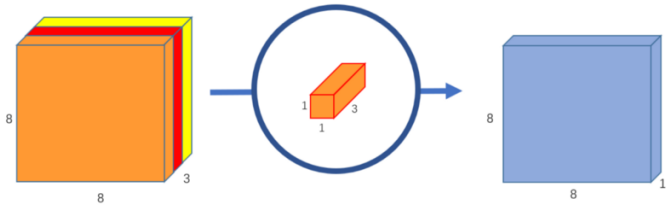


Figura 3.5 Funcionamiento de la convolución puntiaguda..

El uso de estas convoluciones servirá para sustituir las operaciones de agrupación de la red. Además, ambos pasos van seguidos de una normalización por lotes y una función de activación ReLU.

3.2.4 Decodificación

EL codificador esta basado en una zancada de salida (la relación entre el tamaño de la imagen original con el tamaño de las características codificadas finales), o *output stride*, de 16. En lugar de usar el muestreo bilineal con factor de 16, las características codificadas se muestrean primero con un factor de 4 y concatenadas con las características de bajo nivel correspondientes de al modulo codificador que tiene las mismas dimensiones espaciales.

Antes de la concatenación, convoluciones 1x1 son aplicadas sobre las características de bajo nivel para reducir el número de canales. Después de la concatenación, se aplican convoluciones 3x3 y las características son desmontadas con un factor de 4. Este conjunto de operaciones nos da una salida del mismo tamaño que la imagen de entrada.

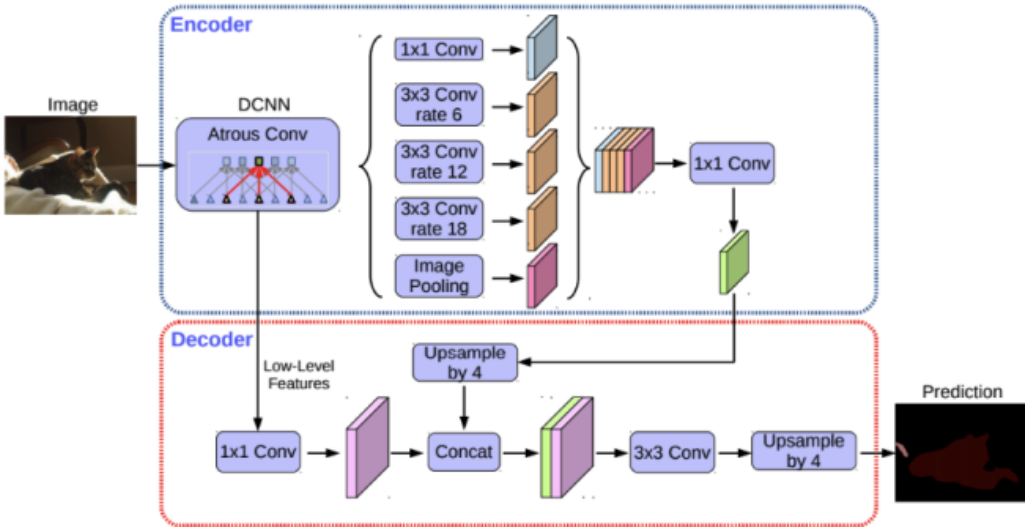


Figura 3.6 Estructura completa del funcionamiento en fases de *Deeplab*.

3.3 MobileNet

MobileNet [16] es una familia de redes neuronales de visión por computador de uso general diseñada pensando en dispositivos móviles para poder admitir clasificación y detección de objetos.

[17] La idea detrás de la primera versión, *MobileNetv1*, era la de sustituir las capas convolucionales, las cuáles son esenciales para tareas de visión por computador pero tienen una alta capacidad computacional, por las convoluciones profundas separables anteriormente explicadas.

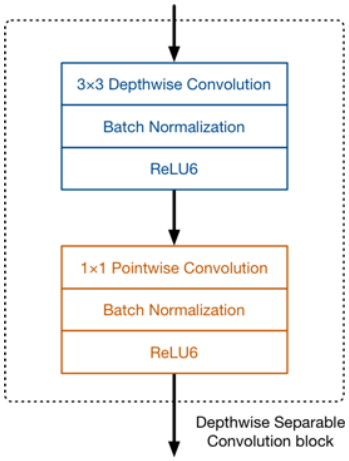


Figura 3.7 Resumen de la estructura de *MobileNetv1*.

Esta sustitución se debe a que se mantiene un funcionamiento similar reduciendo el coste computacional [18]. Por ejemplo, si tuviésemos una imagen de entrada de h_i x w_i x d_i y la transformásemos en una salida de h_i x w_i x d_j mediante un núcleo k x k, el coste computacionales con una capa convolucional estándar sería:

$$h_i \cdot w_i \cdot d_i \cdot d_j \cdot k^2$$

mientras que, con la convolución profunda separable, el coste se reduce a:

$$h_i \cdot w_i \cdot d_i \cdot (k^2 + d_j)$$

Además, cabe destacar que la función de activación escogida por los autores es la ReLU6, la cuál funciona similar a la clásica RelU pero evitando activaciones que puedan ser muy grandes.

La versión actual, *MobileNetv2*, mantiene la convolución profunda separable pero la estructura principal sufre cambios. Ahora, nos encontramos que en el bloque hay 2 capas convolucionales. Las 2 últimas siguen siendo las mismas de la primera versión, siendo la primera capa, la de expansión, la única adición completamente nueva. Además, la convolución de punto tiene un propósito distinto.

En la primera versión, la convolución de punto mantiene el número de canales o los duplica. Sin embargo, en la segunda versión ocurre lo contrario: se reduce el número de canales. Éste es el motivo por el que es conocida ahora como capa de proyección, ya que proyecta datos con un número alto de canales en un tensor con menores dimensiones. Esto provoca un cuello de botella que da nombre a la nueva estructura del bloque (*Bottleneck Residual Block*).

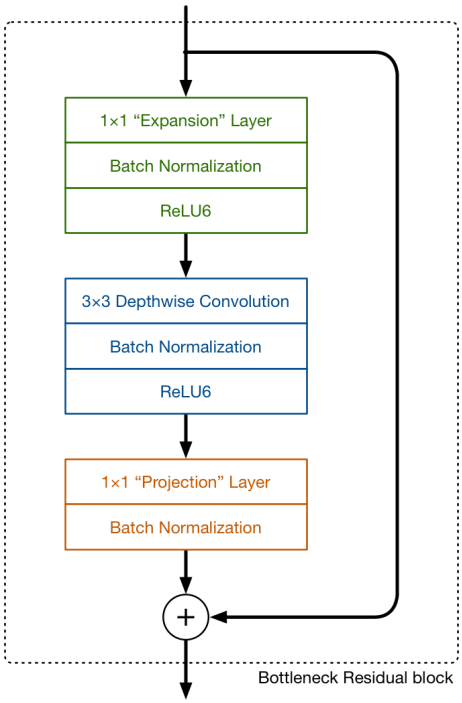


Figura 3.8 Resumen de la estructura de *MobileNetv2*.

La primera capa, que es nueva en este bloque, es también una convolución 1 x 1. Su propósito es expandir el número de canales en los datos antes de que se les aplique la convolución profunda. Por tanto, esta capa de expansión funciona de manera opuesta a la capa de proyección. El parámetro que nos proporciona cuánto se expanden los datos se llama factor de expansión. Su valor predeterminado es 6.

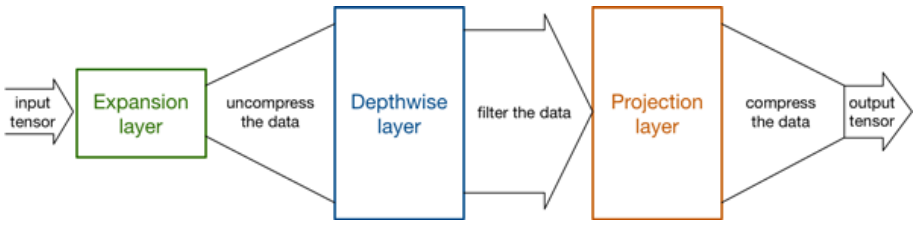


Figura 3.9 Explicación de la estructura del *Bottleneck Residual Block*.

La segunda adición que se realiza en esta versión es la conexión residual. Ésta existe para ayudar con el flujo de los gradientes a través de la red. La conexión residual solo es usada cuando el número de canales a la entrada del bloque es el mismo que a la salida.

3.3.1 MobileNet y Deeplab

En la actual implementación de **DeepLab**, *DeepLabv3+*, se apoya del uso de diversas redes troncales, entre las cuáles se encuentra *MobileNetv2*.

La forma que se combinan ambos es bastante simple, debido a que varios autores trabajan en ambos proyectos. La influencia de a primera versión es obvia en la aparición la convolución profunda separable a partir de *DeepLabv3*. Con el uso de *MobileNetv2*, simplemente hay que aplicar el cambio de ese bloque al bloque residual de cuello de botella.

3.4 Labelme

Labelme [19] es una herramienta de anotación gráfica de imágenes, escrita en python y que usa Qt para su interfaz gráfica. La utilidad de esta herramienta es poder crear, a partir de las imágenes proporcionadas, un conjunto de datos adecuado para el formato de nuestro modelo en *DeepLab*.

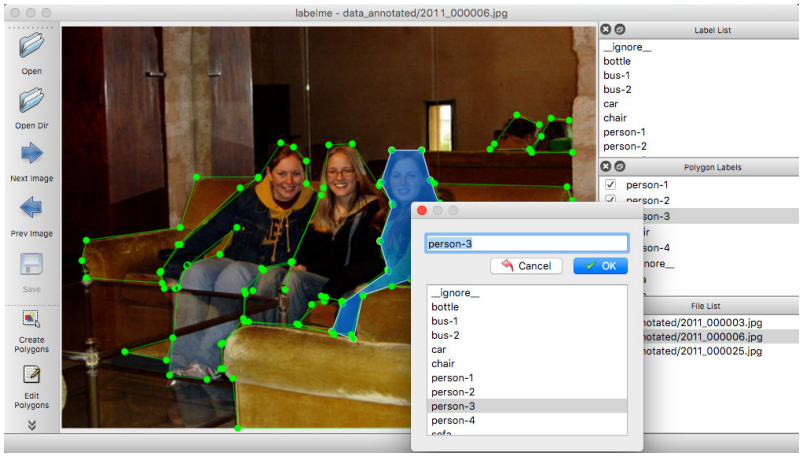


Figura 3.10 Ejemplo del funcionamiento de *labelme*.

3.5 Librerías

Debido a que *TensorFlow* es más sencillo de utilizar en *Python* que en *C/C++*, *DeepLab* se encuentra desarrollado en este lenguaje de programación. Por ello, se ha hecho uso de las siguientes librerías, las cuáles son necesarias para un correcto funcionamiento:

3.5.1 NumPy

NumPy [20] es un paquete para computación científica en *Python*. Entre otras cosas, contiene:

- Grandes vectores N-dimensionales.
- Sofisticadas funciones de transmisión.
- Herramientas para integrar C/C++.
- Capacidades de álgebra lineal, transformada de Fourier y números aleatorios.

Mas allá de sus usos científicos, *NumPy* puede ser usado como un contenedor multi-dimensional eficiente de datos genéricos. Esto permite que se integre rápidamente a una gran variedad de bases de datos.

3.5.2 Pillow

Pillow [21], anteriormente conocida como *Python Imaging Library* (abreviado como *PIL*), es una librería de código abierto que agrega soporte para abrir, manipular y guardar diferentes formatos de archivos de imagen.

3.5.3 Matplotlib

Matplotlib [22] es una librería de código abierto para trazado en *Python* y pensado para usarse junto con la librería *NumPy*. Con ella, se pueden generar histogramas, diagramas y otros tipos de gráficas de forma sencilla.

3.5.4 Sys & OS

Ambas son librerías [23] [24] nativas de *Python* y que permiten acceder a elementos del sistema operativo. Sus mayores utilidades son poder acceder a archivos del sistema y poder leer parámetros introducidos por linea de comandos.

4 Implementación del modelo

En este capítulo vamos a describir todos los puntos necesarios para poder llegar a obtener resultados. Para ello es necesario el etiquetado de las imágenes que se usarán para entrenar las redes, la creación del entorno de trabajo y realizar la configuración necesaria para poder hacer posible el entrenamiento de las redes y la obtención de resultados. [1]

4.1 Etiquetado de imágenes

Para poder realizar diferentes experimentos con las redes neuronales y comprobar los resultados es necesario tener una base de datos funcional. Esta base de datos debe estar conformada por conjunto de imágenes, las cuáles tendrán su versión etiquetada. Dicha versión consiste de una imagen con las siluetas de los objetos que queremos detectar.

Figura 4.1 Imágenes pertenecientes a la base de datos del proyecto.

En nuestro proyecto, la base de datos que hemos creado estará compuesta por un total de 1003 imágenes, las cuáles son en su totalidad de herramientas. No todas las imágenes comparten las

[1] Todos los códigos y directorios necesarios se encuentran en el repositorio: https://github.com/Dani2J/deeplab-own-dataset

mismas dimensiones, aún así se ha limitado el tamaño de estas a menos de 1000 x 1000 píxeles para aligerar el coste computacional de las redes. Las clases que se han utilizado en este proyecto son:

- Alicates (*plier*).
- Llave inglesa (*wrench*).
- Destornillador (*screwdriver*).

Para obtener las imágenes etiquetadas, se usará la herramienta de etiquetado manual *labelme*. Para usarla con nuestras imágenes, se ejecuta mediante la ventana de comandos:

Código 4.1 Comando para ejecutar *labelme*.

```
labelme images --labels labels.txt --nodata
```

dónde:

- **images**: fichero dónde se encuentran las imágenes a etiquetar.
- **labels.txt**: documento de texto dónde se ha escrito el nombre de nuestras clases y de los elementos de fondo (*background*) y separación (*ignore_label*).

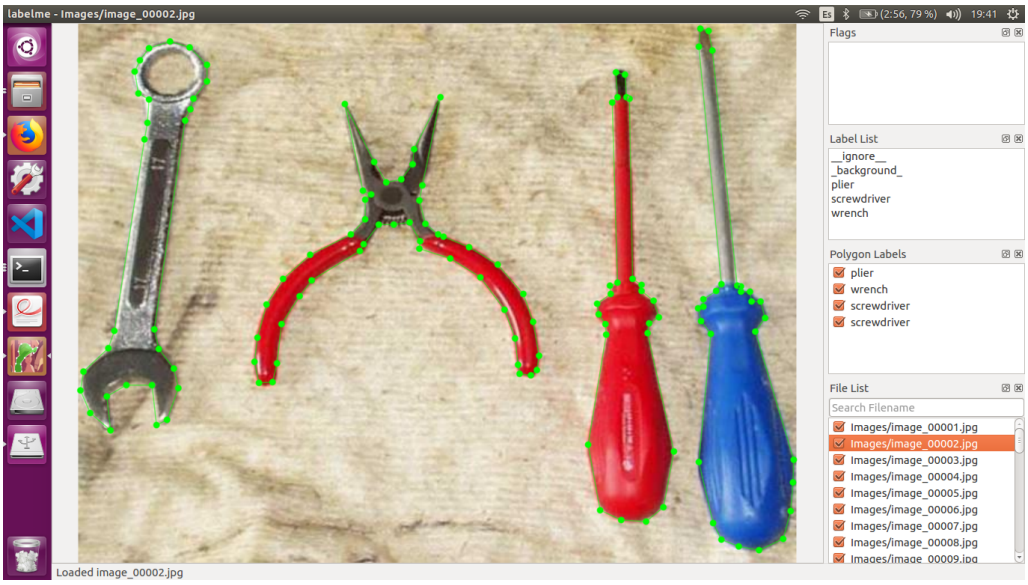


Figura 4.2 Captura del funcionamiento de *labelme*.

Una vez etiquetadas todas las imágenes, se genera un archivo *json* por cada imagen, los cuáles, mediante un código de *python* que se puede descargar desde la página de *github* de la herramienta, darán lugar a las imágenes etiquetadas:

Código 4.2 Comando para obtener las imágenes etiquetadas.

```
python ./labelme2voc.py images results --labels labels.txt
```

siendo **results** el nombre de la carpeta dónde se generarán los resultados.

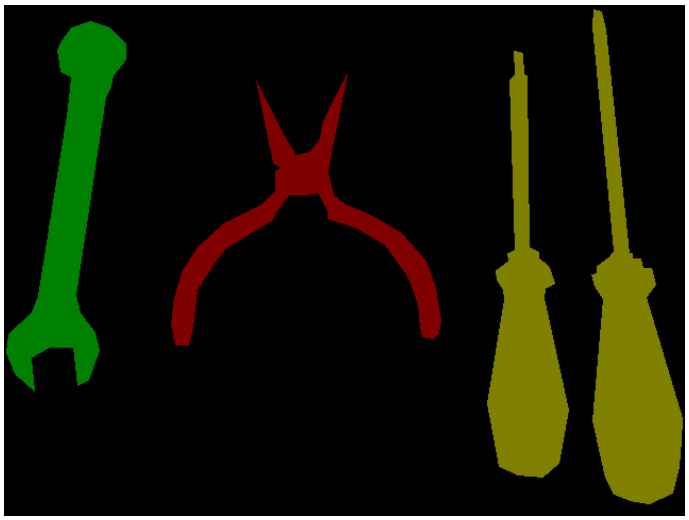


Figura 4.3 Resultados del etiquetado.

Además, las imágenes contienen una etiqueta numérica para cada clase, el cuál es importante cuando se genere el entorno de trabajo. Para nuestras imágenes, dichas etiquetas son:

- 0: *background*
- 1: *plier*
- 2: *wrench*
- 3: *scredriver*

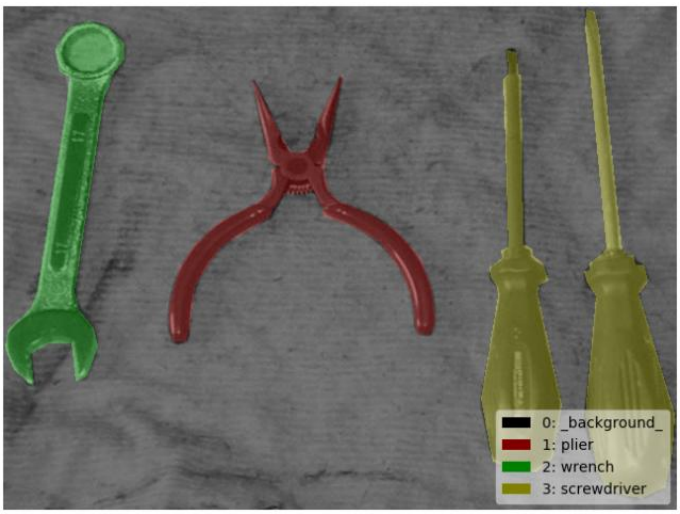


Figura 4.4 Etiquetas de las clases.

4.2 Generación del entorno del trabajo

Una vez obtenemos las imágenes y sus equivalentes etiquetadas, es momento de obtener todos los elementos necesarios. Para ello, primero se ha de generar un directorio (llamado **test** en nuestro caso) con la siguiente estructura:

- dataset
  - ImageSets
  - JPEGImages
  - SegmentationClass
  - SegmentationClassRaw
- exp
- init_model
- tfrecord

dónde:

- **ImageSets**: conformado por ficheros de texto que corresponden con el conjunto de imágenes que usaremos para cada tarea (entrenamiento, evaluación y visualización). Estos documentos incluyen el nombre de todas las imágenes que se incluyen en el conjunto.
- **JPEGImages**: conformado las imágenes que se han usado de base para este proyecto.
- **SegmentationClass**: como su propio nombre puede dar a entender, son las imágenes etiquetadas generadas mediante *labelme*.
- **SegmentationClassRaw**: las verdaderas imágenes que se usarán para el entrenamiento son una versión en escala de grises de las imágenes etiquetadas.
- **exp**: los archivos que se vayan generando en las diversas tareas son guardados en este directorio.
- **init_model**: carpeta dónde se encuentra el modelo pre-entrenado que se usará para poder empezar el proyecto.
- **tfrecord**: dónde se guardarán los archivos de almacenamiento binario que genera *TensorFlow*, cuyo formato es homónimo al nombre de la carpeta.

Con todo el directorio estructurado, necesitamos obtener los datos necesarios paras poder empezar con las tareas. Esto lo podremos realizar con el fichero *convert_test.sh*, en el cuál se invoca a dos programas de *python*.

Código 4.3 Conversión a escala de grises.

```
echo "Removing the color map in ground truth annotations..."
python ./remove_gt_colormap.py \
  --original_gt_folder="${SEG_FOLDER}" \ # imagenes etiquetadas
  --output_dir="${SEMANTIC_SEG_FOLDER}" # imagenes en escala de grises
```

En primer lugar, pasamos a obtener la versión en escala de grises de las imágenes etiquetadas. Estas imágenes son exactamente iguales a su versión a color pero su color en la escala de grises corresponde con la etiqueta numérica. Visualmente, la diferencia no es muy apreciable pero permite a la redes saber perfectamente a que clase corresponde cada objeto.

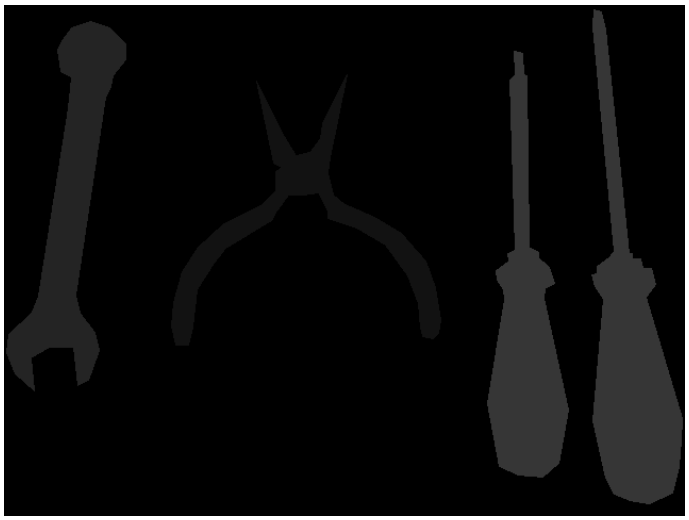


Figura 4.5 Imagen etiquetada en escala de grises.

Después, se generarán los archivos *tfrecord*, los cuáles corresponderán a cada una de las listas de imágenes que hemos creado. Debido a que el proyecto toma de base el ejemplo de los autores de *DeepLab* con el set de datos "*PASCAL_VOC_2012*" [25], se ha optado por usar el programa de *python* utilizado en dicho experimento.

Código 4.4 Obteniendo los archivos *tfrecord*.

```
echo "Converting dataset..."
python ./build_voc2012_data.py \
  --image_folder="${IMAGE_FOLDER}" \ # imagenes jpeg
  --semantic_segmentation_folder="${SEMANTIC_SEG_FOLDER}" \ # imagenes
     en escala de grises
  --list_folder="${LIST_FOLDER}" \ # listas en archivos txt
  --image_format="jpg" \ # formato de las imagenes originales
  --output_dir="${OUTPUT_DIR}" # carpeta tfrecord
```

En este proyecto se ha optado por usar 3 listas de imágenes para poder entender mejor como se organizan las imágenes para esta clase de proyectos, aunque, a efectos prácticos, solo sería realmente necesaria una única lista. Estas listas son:

- **train.txt**: lista de imágenes usadas para el entrenamiento. Normalmente, suelen ser entre un 80 % y un 90 % del total del conjunto. En este caso, 899 imágenes.
- **val.txt**: lista con las imágenes restantes (104 imágenes) y que son usadas para las tareas de evaluación y visualización.
- **trainval.txt**: lista con todas las imágenes. Debido a lo relativamente pequeño que es este set de imágenes, se ha optado por realizar el entrenamiento con la totalidad de las imágenes que tenemos disponibles.

A partir de estas listas, se generan varios (en este caso, 4) archivos de almacenamiento *tfrecord* por cada lista.

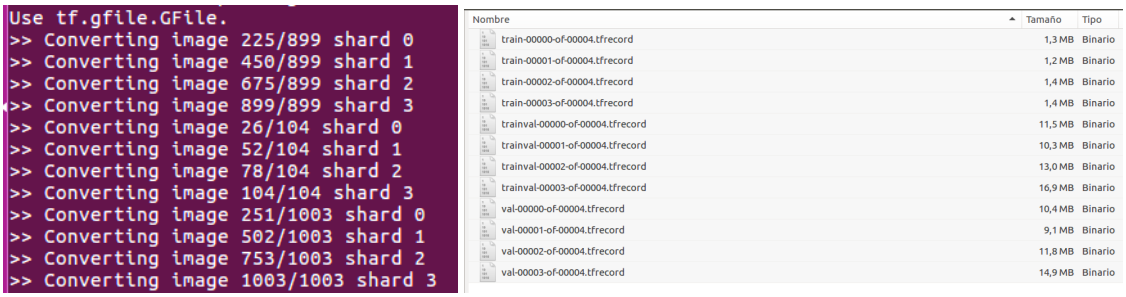


Figura 4.6 Generación de los archivos de almacenamiento *tfrecord*.

4.3 Entrenamiento y seguimiento

Una vez obtenidos estos datos, podemos pasar a poder trabajar con las redes. Para ello, obtenemos un modelo pre-entrenado y se configuran los procesos para poder funcionar de forma acorde a nuestro proyecto.

4.3.1 Modelo pre-entrenado

Para poder iniciar con el entrenamiento, necesitamos de un modelo pre-entrenado. Este tipo de modelos son obtenibles desde la página de *DeepLab* [26], habiendo una gran variedad de modelos según el set de datos y la estructura utilizada.

Debido a que usamos *MobileNetv2* y nuestro set de datos se inspira en el de *PASCAL_VOC_-2012*, se ha optado por el modelo **deeplabv3_mnv2_pascal_train_aug**. Dicho modelo, con sus características y conjunto de datos predefinidos, garantiza un MIOU (*Mean Intersection-Over-Union*) del 75.34 %. Este valor representa el área etiquetada correctamente sobre el total de área etiquetada, tanto por nosotros manualmente como por la red.

4.3.2 Configuración general

Para poder usar nuestro conjunto de imágenes, debemos incluirlo en la lista de *datasets* disponibles y configurar sus características para que sean funcionales con la estructura de la red. Estas modificaciones, se aplican en el programa **segmentation_dataset.py**:

Código 4.5 Declaración del set de datos.

```
_TEST_INFORMATION =DatasetDescriptor(
    splits_to_sizes={
        'train': 899, # numero de imagenes de la lista
        'trainval': 1003,
        'val': 104,
    },
    num_classes=5, # numero de clases + background + ignore_label
    ignore_label=255, # bordes que separan las clases

# se describen los dataset restantes
# ...

_DATASETS_INFORMATION ={
```

```
    'cityscapes': _CITYSCAPES_INFORMATION,
    'pascal_voc_seg': _PASCAL_VOC_SEG_INFORMATION,
    'ade20k': _ADE20K_INFORMATION,
    'test': _TEST_INFORMATION,
}
```

El caso del valor de *ignore_label*, a pesar de que no ha sido necesario generarlo cuando se han obtenido las imágenes etiquetadas, consiste en ponerle el valor más alto posible en la escala de grises para evitar que anule cualquiera de las clases que tenemos.

4.3.3 Configuración de entrenamiento

Una vez tenemos el modelo que usaremos de base y tenemos declarado nuestro set de datos, podríamos ejecutar directamente el código para entrenar las redes (**train.py**) con los pesos predefinidos.

Debido a las diferencias en los conjuntos de imágenes y la diferencia de clases entre nuestro proyecto y el del ejemplo (5 clases frente a 21), se van a modificar los pesos para las redes el programa **train_utils.py**:

Código 4.6 Configurando los pesos para cada clase.

```
irgore_weight =0 # ignore_label
label0_weight =1 # background
label1_weight =12 # plier
label2_weight =10 # wrench
label3_weight =10 # screwdriver
not_ignore_mask =
    tf.to_float(tf.equal(scaled_labels, 0)) *label0_weight +
    tf.to_float(tf.equal(scaled_labels, 1)) *label1_weight +
    tf.to_float(tf.equal(scaled_labels, 2)) *label2_weight +
    tf.to_float(tf.equal(scaled_labels, 3)) *label3_weight +
    tf.to_float(tf.equal(scaled_labels, ignore_label)) *irgore_weight
```

Con los pesos modificados manualmente, hemos habilitar en **train.py** que las redes trabajen con los nuevos pesos en lugar de los predefinidos por el modelo. Para ello, debemos poner los siguientes valores a *False* en el código:

Código 4.7 Habilitando el cambio de pesos y clases.

```
flags.DEFINE_boolean('initialize_last_layer', False, 'Initialize the
                                    last layer.') # Cambiar el numero
                                    de clases

flags.DEFINE_boolean('last_layers_contain_logits_only', False, 'Only
                                    consider logits as last layers or
                                    not.') # Cambiar los pesos de las
                                    clases
```

Por último, debemos habilitar el uso de de nuestro set de datos, tambien en **train.py**:

Código 4.8 Habilitando nuestro set de datos.

```
# Dataset settings.
flags.DEFINE_string('dataset', 'test', 'Name of the segmentation dataset
                                        .')
```

Con todos los parámetros ajustados y todas las modificaciones habilitadas, podemos ejecutar finalmente el entrenamiento, a través de **train.py**:

Código 4.9 Ejecutando el entrenamiento.

```
# Train 3000 iterations.
NUM_ITERATIONS=3000
python "${WORK_DIR}"/train.py \
  --logtostderr \
  --train_split="trainval" \ # imagenes que usaremos para entrenar
  --model_variant="mobilenet_v2" \
  --output_stride=16 \
  --train_crop_size=513 \
  --train_crop_size=513 \
  --train_batch_size=4 \
  --training_number_of_steps="${NUM_ITERATIONS}" \
  --fine_tune_batch_norm=true \
  --tf_initial_checkpoint="${INIT_FOLDER}/${CKPT_NAME}/model.ckpt-30000"
      \ # modelo pre-entrenado
  --train_logdir="${TRAIN_LOGDIR}" \ # directorio dónde se guardan los
     datos del entrenamiento
  --dataset_dir="${PASCAL_DATASET}" # carpeta tfrecord
```

dónde:

- *output_stride*: explicado cuando hablamos de decodificación en la arquitectura de *DeepLab* (véase *Capítulo 3*). Mantiene el valor estándar de 16.
- *train_crop_size*: tamaño al que se ajustan las imágenes para poder ser entrenadas. No es necesario que se ajuste al tamaño de las imágenes, ya que esto aumentaría el coste computacional. Hay dos entradas para ajustarse a cada una de las dimensiones, aunque puede realizarse con una entrada usando un vector o un único valor, si ambas dimensiones van a ser iguales.
- *train_batch_size*: las entradas se dividen en lotes o *batch* en vez de ser mandadas todas a la vez.
- *fine_tune_batch_norm*: ajusta el tamaño de los lotes. En caso de usarse lotes grandes (> 12), es recomendable ponerlo en *False*.

Mientras se ejecuta el entrenamiento, se puede hacer seguimiento de este viendo las perdidas que se producen aproximadamente cada 10 iteraciones. Además, podemos ver cada cuantas iteraciones se realiza un guardado del *checkpoint*, que se ubica en la carpeta donde se guardan los datos de entrenamiento.

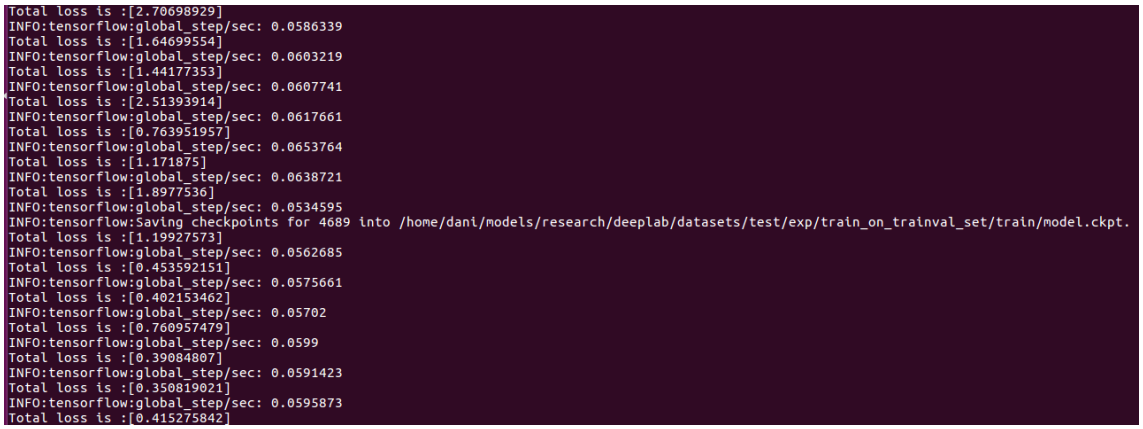


Figura 4.7 Seguimiento del entrenamiento.

4.3.4 Configuración de evaluación

A diferencia de con el entrenamiento, con la evaluación de los resultados no se ha de hacer ninguna modificación previa de datos ni habilitar ningún tipo de características. Solamente, se ha de ejecutar el código **eval.py**:

Código 4.10 Ejecutar evaluación.

```
python "${WORK_DIR}"/eval.py \
  --logtostderr \
  --eval_split="val" \
  --model_variant="mobilenet_v2" \
  --eval_crop_size=1201 \
  --eval_crop_size=1201 \
  --checkpoint_dir="${TRAIN_LOGDIR}" \ # carpeta donde se guardan los
     datos del entrenamiento
  --eval_logdir="${EVAL_LOGDIR}" \ # entrada donde se guardan los datos
     de evaluación
  --dataset_dir="${PASCAL_DATASET}" \
  --max_number_of_evaluations=1
```

Como se puede apreciar, las entradas de datos *eval_crop_size* no coinciden con sus equivalentes usadas en el entrenamiento. Esto se debe a que, al tratarse de una evaluación de los valores de salida, trabajamos con las dimensiones originales de las imágenes. Por tanto, se ha optado por un número superior al de las dimensiones de las imágenes más grandes.

La función de este código, además de permitirnos visualizar posteriormente los resultados, es la de informar de la calidad de nuestras redes a través del MIOU previamente nombrado. Para poder ver y seguir la evolución de este valor con el paso de las evaluaciones, se ha usado *TensorBoard*, una *suite* de visualización que incluye *TensorFlow*. Ésta se ejecuta por ventana de comandos y nos genera una dirección aplicable en nuestro navegador, desde dónde seguir nuestros resultados.

Código 4.11 Ejecutar *TensorBoard*.

```
# Desde dentro del directorio exp
tensorboard --logdir .
```

4.3.5 Configuración de visualización

Para poder representar cada una de clases debemos añadir la paleta de colores que queremos usar en **get_dataset_colormap.py**:

Código 4.12 Añadir paleta de colores para las clases.

```
def create_test_label_colormap():
 return np.asarray([
      [ 0, 0, 0], #background
      [254, 0, 0], #plier
      [0, 254, 0], #wrench
      [ 0, 0, 254], #screwdriver
      ])
# Se crean el resto de paletas de colores
# Se asigna la paleta al nombre de nuestro dataset
def get_test_name():
  return _TEST

def create_label_colormap(dataset=_TEST):
  if dataset == _ADE20K:
    return create_ade20k_label_colormap()
  elif dataset == _CITYSCAPES:
    return create_cityscapes_label_colormap()
  elif dataset == _MAPILLARY_VISTAS:
    return create_mapillary_vistas_label_colormap()
  elif dataset == _PASCAL:
    return create_pascal_label_colormap()
  elif dataset == _TEST:
    return create_test_label_colormap()
  else:
    raise ValueError('Unsupported dataset.')
```

Además, hay que habilitar el uso de la paleta en **vis.py** para que así pueda ser capaz de usarla:

Código 4.13 Habilitación de la paleta de colores.

```
flags.DEFINE_enum('colormap_type', 'test', ['pascal', 'cityscapes', '
                                    test'], 'Visualization colormap
                                    type.')
```

Por último, ejecutamos la visualización mediante el programa **vis.py**. Éste recibe exactamente los mismos parámetros que **eval.py** a la hora de iniciarlo mediante ventana de comandos. Además, es posible ejecutar simultáneamente la evaluación y la visualización ya que no depende una acción de la otra.

Código 4.14 Ejecutar visualización.

```
# Visualize the results.
python "${WORK_DIR}"/vis.py \
```

```
--logtostderr \
--vis_split="val" \
--model_variant="mobilenet_v2" \
--vis_crop_size=1201 \
--vis_crop_size=1201 \
--checkpoint_dir="${TRAIN_LOGDIR}" \
--vis_logdir="${VIS_LOGDIR}" \ # Carpeta donde guardar los resultados
   finales
--dataset_dir="${PASCAL_DATASET}" \
--max_number_of_iterations=1
```

Desde la ventana de comandos, podemos seguir el ritmo al que se generan las imágenes resultantes, las cuales se encontrarán en el directorio que le hayamos indicado (en nuestro caso, una carpeta propia dentro del directorio **exp**)

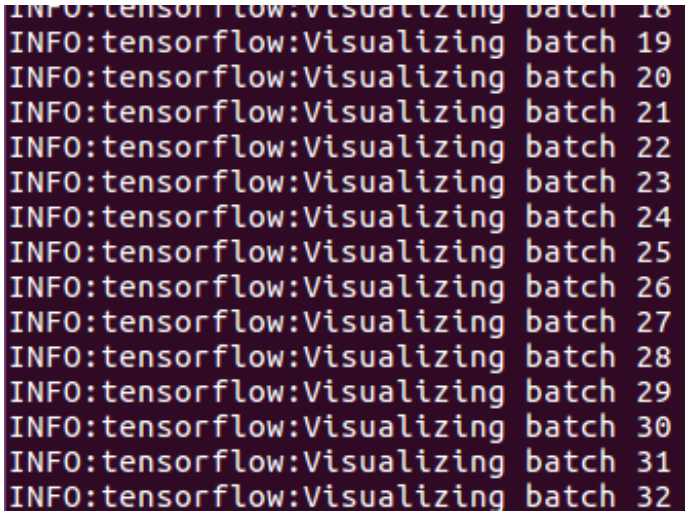


Figura 4.8 Seguimiento de la visualización.

5 Resultados

En este capítulo se realizará un seguimiento de como se ha ido desarrollando todo el proceso para obtener los resultados. Además, se añadirá un comentario con los resultados finales y como se han llegado hasta ellos.

5.1 Procedimientos de entrenamiento

Inicialmente, se ha optado por iniciar el entrenamiento con una cierta metodología: asignar unos valores aleatorios a los pesos (los vistos en el **Código 4.6**) e ir modificando estos en base a los resultados que se han ido obteniendo mientras se entrena con imágenes de las diversas clases.

Figura 5.1 Resultados del primer método.

Esta metodología sería la más apropiada en múltiples tipos de redes, ya que evita el entrenamiento fallido que se produce si este se centra en una sola clase. En nuestro caso, esta forma de trabajar no surte el efecto deseado debido a la complejidad de datos con la que trabaja la red y, por tanto, las pruebas modificando los pesos no sean del todo concluyentes. Además, cabe sumar a este incidente un cierto desbalance en el conjunto de imágenes entre una de las clases (alicates o *plier*) sobre las demás. Todo esto provoca que, más allá de esta clase, no se pueda apreciar unos resultados plenamente funcionales (MIOU alrededor del 20 %).

A raíz de esto, se ha optado por un método diferente. Este consistirá en hacer entrenamientos a pequeña escala con cada una de las clases individualmente para así poder obtener de una manera relativamente más rápida el valor correspondiente a los pesos de cada clase y poder realizar, posteriormente, el entrenamiento completo.

Para realizar este tipo de entrenamiento, se han creado 2 listas más de imágenes para poder entrenar, correspondientes a la segunda (*wrench*) y tercera clase (*screwdriver*), y se ha optado por no realizar esta práctica con la primera clase debido al ser la única que tiene unos resultado aceptables con el primer método, por lo que se puede llegar a suponer que su peso está bien ajustado.

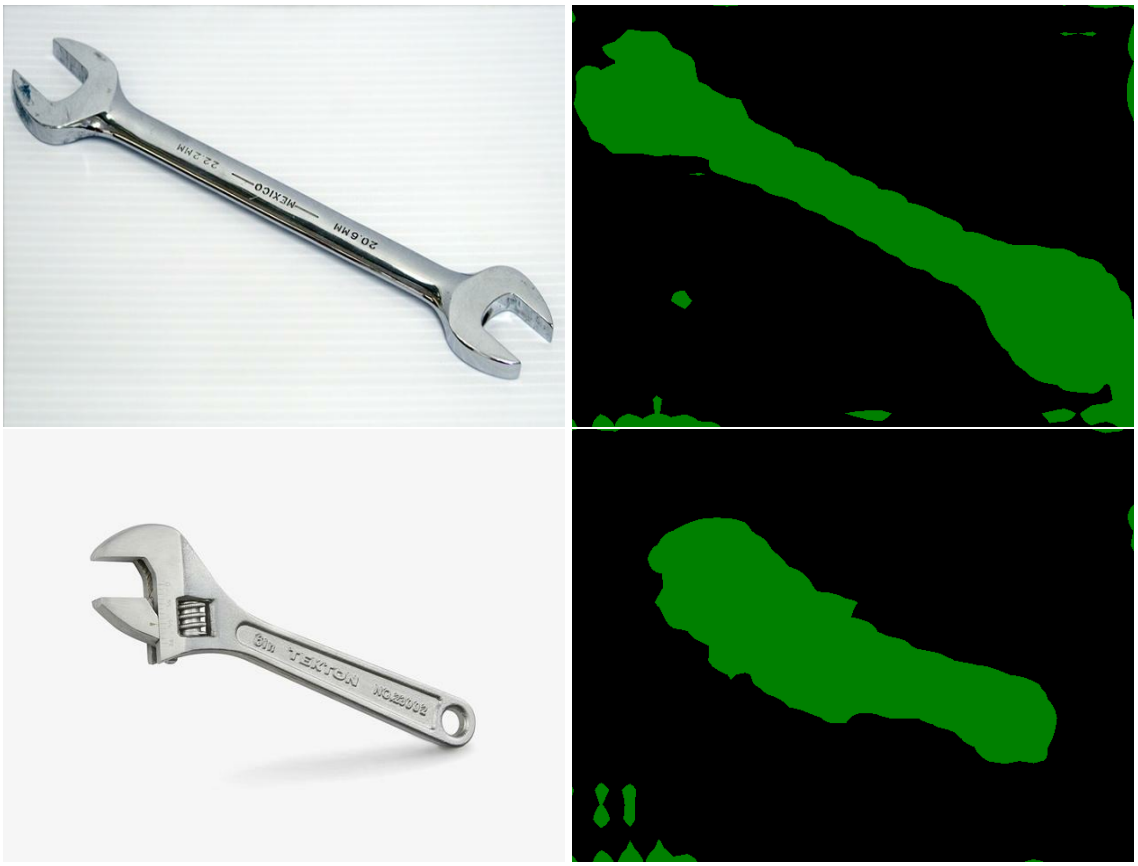


Figura 5.2 Resultados del entrenamiento individual para la segunda clase.

Cabe destacar que, a pesar de ser funcionales los pesos sacados de estos entrenamiento individuales, se deberá tener en cuenta que estos valores puedan estar sujetos a variaciones para ajustarse completamente al entrenamiento general.

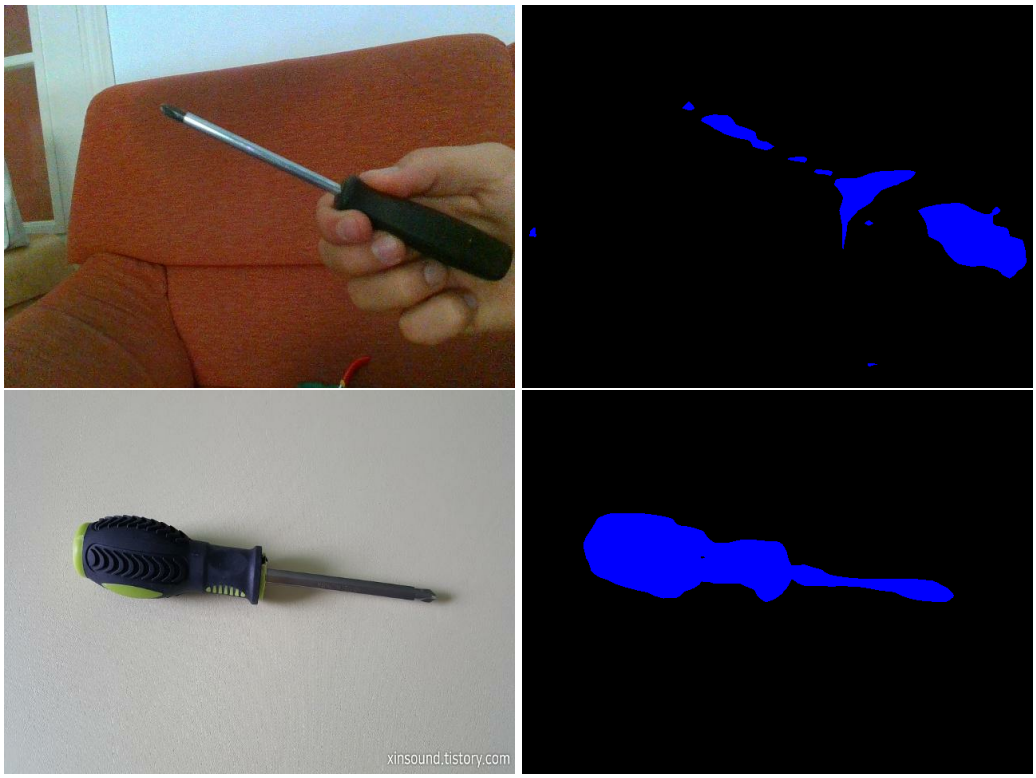


Figura 5.3 Resultados del entrenamiento individual para la tercera clase.

Por cuestiones de tiempo, no se han podido extender mucho los entrenamientos individuales pero no se escogieron los pesos hasta que no se obtuviesen unos resultados medianamente funcionales. A su vez, esto nos permite afirmar que no hay ningún inconveniente a la hora de detectar cada clase.

Finalmente, los valores utilizados para los pesos asignados a cada clase tras los entrenamientos individuales son los siguientes:

Tabla 5.1 Pesos asignados a cada clase tras los entrenamientos individuales.

Pesos asignados a cada clase	
Clase	Rango de pesos
ignore_label	0
background	1
plier	10-12
wrench	2-5
screwdriver	6-8

Debido a la inexistencia de bordes para la separación de clases, la clase *ignore_label* es innecesaria en nuestro proyecto, tal como ha sido descrito anteriormente. Por tanto, se le ha asignado un valor de 0 para evitar conflictos, ya que *DeepLab* la tiene en cuenta.

5.2 Evaluación final

Una vez obtenidos unos pesos, en teoría, adecuados para nuestra red neuronal, se pasa al entrenamiento final. Este se extenderá todo el tiempo posible o hasta que se compruebe que el funcionamiento acaba siendo limitado.

Este entrenamiento final ha consistido de un total aproximado de 7500 iteraciones de entrenamiento usando la lista de imágenes **trainval.txt** como datos de entrada y asignando los pesos mostrados en la **Tabla 5.1**. La evolución de los resultados de la evaluación del modelo con la lista **val.txt** son los siguientes:

Tabla 5.2 Evolución de la calidad del modelo durante el entrenamiento.

Precisión del modelo	
nº de iteraciones	MIOU
3500 (primer método)	20.85 %
500	21.04 %
1500	24.15 %
3500	24.98 %
5000	25.24 %
7546	25.22 %

Los resultados muestran una obvia mejora con respecto al modelo previo, viendo que con unas pocas iteraciones del modelo final se ha conseguido mejorar los resultados.

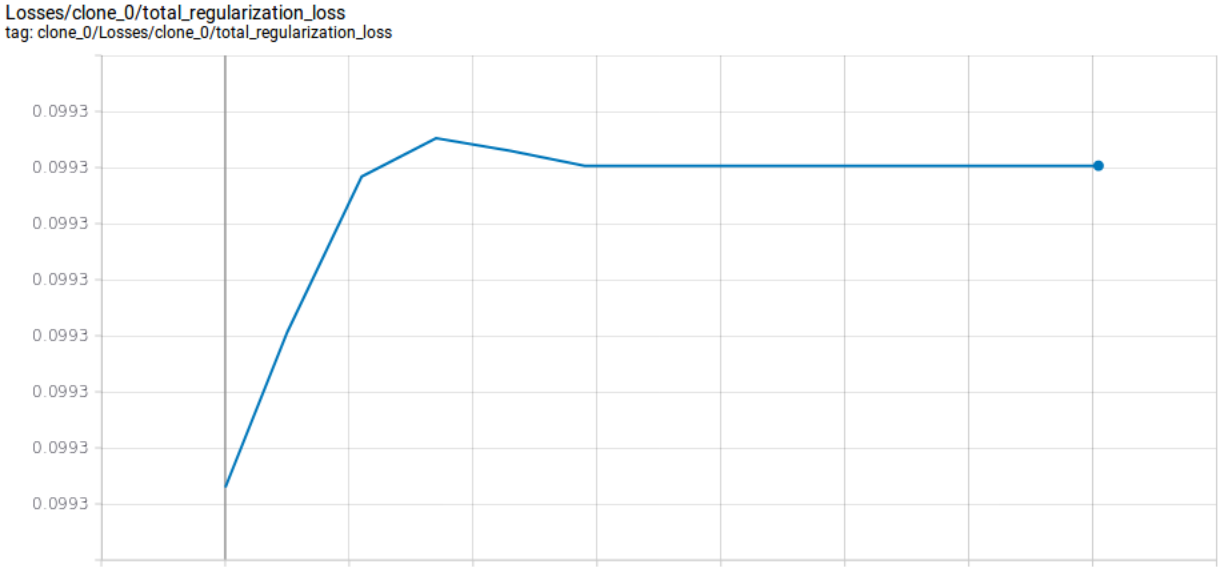


Figura 5.4 Gráfica de perdidas de regularización en *TensorBoard*.

Por otro lado, también se muestra un estancamiento de los resultados a partir de las 3500 iteraciones. Esto no solo es claramente apreciable con el MIOU, si no que en la representaciones obtenidas vía *tensorboard* de las perdidas y del ratio de aprendizaje se puede ver este casi nulo avance del modelo.

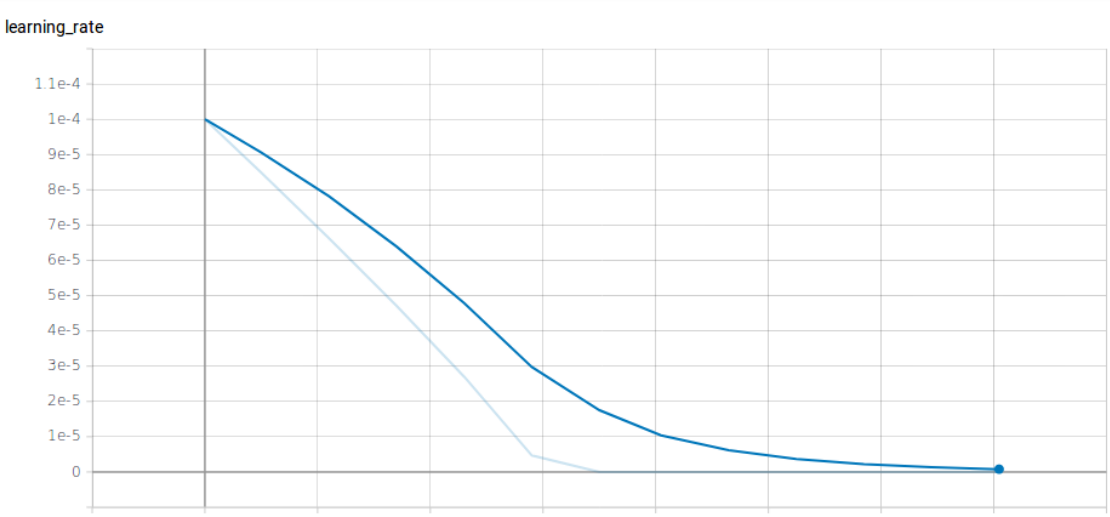


Figura 5.5 Gráfica del ratio de aprendizaje en *TensorBoard*.

La más probable causa de estos resultados es la relativamente baja cantidad de imágenes de nuestro set de datos. Los conjuntos de imágenes más comunes para este tipo de proyectos, como *PASCAL_VOC_2012* [25], *Cityscapes* [27] o *ADE20K* [28], suelen tener una cantidad de imágenes dedicadas a entrenamiento alrededor de 10 veces las que se usan en este proyecto en total. Esto provoca una comprensible falta de recursos para la red y su posterior estancamiento durante el proceso.

5.3 Visualización final

Una vez analizados los resultados de la evaluación, los cuáles son más teóricos, se pasa a observar los resultados a nivel visual.

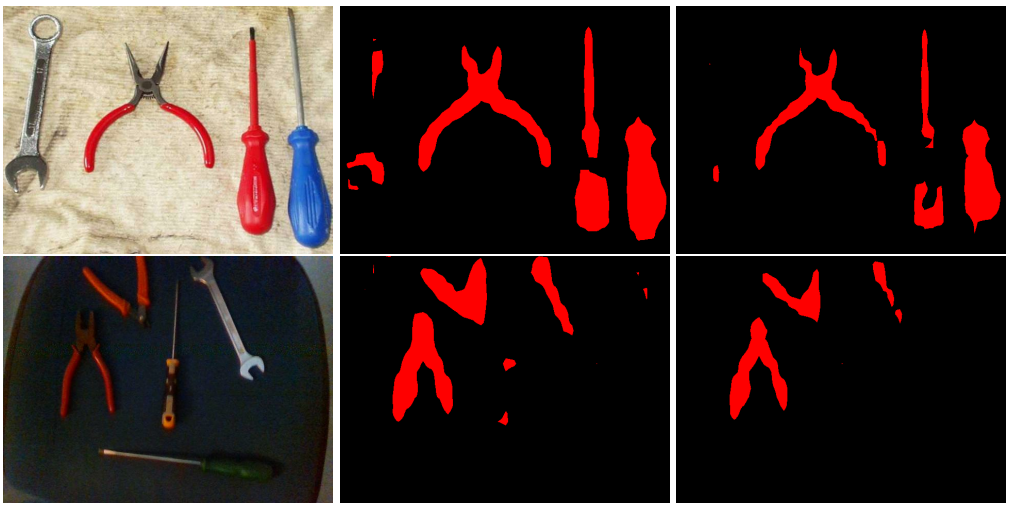


Figura 5.6 Resultados del proyecto en las iteraciones 3500 y 7546.

Estos resultados parecen sufrir de los mismos inconvenientes que en el caso de la primera metodología que aplicamos para el entrenamiento, viéndose una predisposición de la red neuronal

a clasificar la amplia mayoría de los objetos como parte de la primera clase aunque esto se va corrigiendo conforme avanzan los entrenamientos.

Por otro lado, cabe destacar que, a pesar de estos errores en clasificación, la precisión con la que la red detecta los objetos es bastante óptima. Esto podría ser en parte el origen de que, con el paso de las iteraciones en el entrenamiento, objetos muy bien definidos por la red pero mal clasificados en las primeros resultados de este entrenamiento pasen a no ser clasificados. Se podría también asumir que, al ser la primera clase la mayoritaria en el conjunto de imágenes, la red clasifica por defecto a todos los objetos que detecta como dicha clase a espera de una posterior corrección hacia las otras clases con menos presencia.

Con estos resultados y su posterior análisis, se puede asumir que el entrenamiento ha sido fallido y cuáles son los posibles errores del modelo:

- El número de imágenes de entrenamiento. Como ya ha sido comentado en el apartado anterior con los resultados de la evaluación, el número de imágenes usado es bastante pequeño en comparación al de los experimentos realizados por los desarrolladores de *DeepLab*, de un tamaño al menos 10 veces superior al usado en este proyecto. No haber creado un set de datos mayor puede ser en gran parte el resultante del estancamiento del modelo durante el entrenamiento.

- Descompensación de una de las clases. El hecho de que la primera clase (*plier*) sea, hasta cierto punto, más común en las imágenes que las otras clases puede ser responsable de la predisposición de la red a clasificar los objetos como pertenecientes a dicha clase.

- Mala regulación de los pesos. Como se había advertido al tomar la decisión de realizar entrenamientos individuales para obtener los pesos, la posibilidad de que estos no fueran adecuados para el entrenamiento conjunto era posible. Este puede ser otro de los motivos de la predisposición de la red hacia la primera clase.

6 Conclusiones

En este proyecto, se ha intentado mostrar como usar redes neuronales para la identificación de objetos mediante el reconocimiento de sus formas. El principal reto que tenía este proyecto ha sido el entendimiento de *Deeplab* para la creación de redes de una estructura compleja. La segmentación semántica involucra una cantidad amplia de variantes y una considerable complejidad computacional. Todo esto muestra el reto que supone crear un proyecto capa de realizar dicha tarea.

6.1 Reflexión

En conclusión, es posible crear una estructura basada en redes neuronales para poder identificar objetos y, probablemente, seres vivos mediante sus siluetas a partir de imágenes de entrada. Todo esto a pesar de no haber obtenido los resultados más adecuados.

Respecto al entrenamiento, se ha de destacar que, con una cantidad relativamente baja de imágenes de muestra para los números con los que se suele trabajar en esta clase de proyectos, se puede entrenar de manera eficiente las redes para la detección de un tipo de objeto. Con alrededor de 1000 iteraciones en el entrenamiento, se pueden conseguir unas formas medianamente definidas, como muestran los resultados rápidos que hemos tenidos con la clase correspondiente a la primera clase (véase **Figura 5.1**) o los resultados correspondientes a las otras clases en lo entrenamientos individuales (véase **Figura 5.2** y **Figura 5.3**). Es por esto que se decide tomar el enfoque en el entrenamiento para conseguir los pesos adecuados para cada una de las clases.

Todo lo anteriormente dicho se ve, hasta cierto punto, obstaculizado por el desequilibrio entre las clases en las muestras de entrenamiento y por la ralentización en las tareas debido a la falta de procesador más competente.

6.2 Posibles mejoras a realizar

Este proyecto ha sido una primera aproximación a la identificación de objetos mediante el uso de redes neuronales de aprendizaje profundo, La amplitud de los conocimientos que hay en este campo permite que se pudiesen aplicar varias mejoras al proyecto.

Al realizar este proyecto usando redes neuronales es necesario hacer diversos experimentos para poder configurar el comportamiento que han de imitar la red. Por tanto, todo objeto que se quiera detectar deberá ser previamente configurado para después ser generalizado en distintos casos. Todo esto quiere decir que todo objeto que quiera ser detectado deberá ser tenido en cuenta a la hora de entrenar la red.

La aproximación tomada en este proyecto no ha tenido en cuenta diversos factores:

- Un conjunto e imágenes más amplio. Muchos de los ejemplos realizados por los desarrolladores de las diferentes versiones de *DeepLab* y *MobileNet* usan una base de imágenes mucho más amplia para evitar para poder entrenar con una mayor eficiencia y evitar que las redes acaben mal entrenadas, pudiendo ser únicamente funcionales con estas imágenes.

- Mayor número de clases. La complejidad de configurar los pesos de las clases cuando el modelo de base ya tiene una configuración diferente tanto de clases como sus pesos hacen que se haya optado por un número bajo de clases. Con un tiempo mayor para poder entrenar las redes y una mejor capacidad de procesamiento, sería posible crear una red con una capacidad de reconocimiento mucho mayor.

- La diversidad de los elementos de fondo. En este proyecto, se ha tenido en cuenta a todos los elementos externos de las imágenes que no corresponden con ninguna clase como una clase genérica (*background*). Está decisión permite un comportamiento funcional pero puede producir confusiones debido a que estos elementos pueden cambiar de una imagen a otra y hacer que se confundan con alguna de las otras clases.

- El uso de otras redes troncales. El enfoque tomado en el proyecto de querer reducir el coste computacional dio lugar a la decisión de utilizar *MobileNet* pero este no es el único modelo que puede ser usado por *DeepLab*. Otros, como pueden ser *Xception* [29] o *ResNet* [30], han demostrado una gran funcionalidad aunque con un coste computacional mayor al usado en este proyecto.

6.3 Trabajo futuro

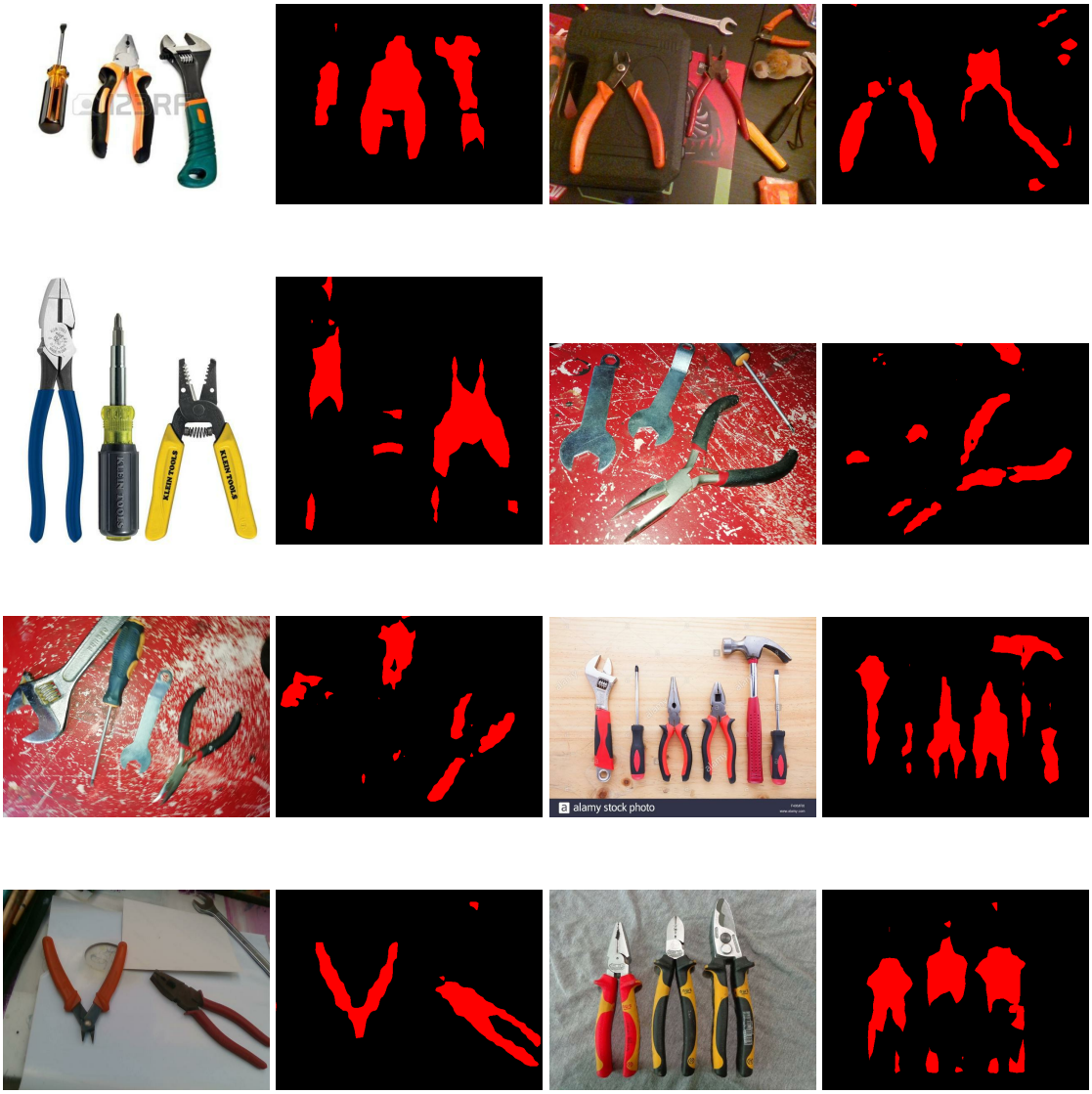
Durante la realización de este proyecto, se han ido produciendo avances tanto en *TensorFlow* como en *MobileNet*, teniendo actualmente el primero una nueva versión en fase beta [31] y el segundo una versión en desarrollo [32]. Estos avances por si solos ya nos permitirían una mejora en la optimización de los recursos y, probablemente, una mejora en los resultados.

Otro punto posible a desarrollar a partir de este proyecto sería una implementación de esta clase de redes para su funcionamiento en tiempo real. Esto tipo de implementación enlazaría con lo desarrollado en la introducción de este documento, cuando se hablaba del posible uso de la segmentación semántica en vehículos autónomos para tareas de SLAM o en las líneas de montaje para la supervisión del proceso. Habiendo ya intentos en esta línea de desarrollo [33], sería una aplicación viable pero dependiente de capacidad de procesamiento que permita un rápido funcionamiento para que sea funcional a efectos reales.

Recientemente, trabajadores de *Google* han estado desarrollando software que permite realizar el seguimiento de los movimientos de una mano y el tipo de gestos que realiza en tiempo real [34]. Aunque este proyecto este centrado en un enfoque diferente al nuestro, sus resultados podrían resultar a futuro en una combinación de funcionalidades que permitiría una mayor precisión en la segmentación semántica que incluyesen algún tipo de ser vivo entre sus clases. Además, el reconocimiento de posiciones especificas podría ser de utilidad para detectar elementos posicionados incorrectamente en una linea de montaje que pudiesen provocar inconvenientes en el funcionamiento de esta.

Otro avance por parte de trabajadores de *Google* es el desarrollo de un entorno para el funcionamiento de redes neuronales que no necesitarían de la introducción de parámetros como los pesos o los umbrales [35]. Estas redes neuronales se encargarían por si misma de encontrar unos valores adecuados para estos parámetros. Este proyecto sería beneficioso a la hora de obtener los resultado, ya que permitiría una mayor precisión de las redes a la hora de trabajar con las clases. Además, también es beneficioso para el desarrollador agilizando el trabajo debido a que al trabajar con reconocimiento con imágenes se necesita ajustar una diversa cantidad de parámetros, los cuáles se verían reducidos en número. El único inconveniente a corto plazo es que demandarían una mayor capacidad de procesamiento que unas redes neuronales más comunes.

Apéndice A
Galería



Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] https://www.elconfidencial.com/tecnologia/2016-05-20/vehiculo-autonomo-empresas-prueban-coche-futuro-google-uber_1203221/.

[2] https://cincodias.elpais.com/cincodias/2019/06/05/gadgets/1559768222_878416.html.

[3] https://pytorch.org/.

[4] https://aws.amazon.com/es/machine-learning/.

[5] https://en.wikipedia.org/wiki/simultaneous_localization_and_mapping.

[6] https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report.

[7] Mauricio Roberto Veronez, Sérgio Florêncio de Souza, Marcelo Tomio Matsuoka, Alessandro Reinhardt, and Reginaldo Macedônio da Silva. Regional mapping of the geoid using gnss (gps) measurements and an artificial neural network.

[8] https://en.wikipedia.org/wiki/multilayer_perceptron.

[9] https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/.

[10] https://en.wikipedia.org/wiki/convolutional_neural_network.

[11] https://brohrer.github.io/how_convolutional_neural_networks_work.html.

[12] https://www.tensorflow.org/.

[13] https://keras.io/.

[14] https://github.com/tensorflow/models/tree/master/research/deeplab.

[15] https://www.analyticsvidhya.com/blog/2019/02/tutorial-semantic-segmentation-google-deeplab/.

[16] https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet.

[17] https://machinethink.net/blog/mobilenet-v2/.

[18] Mark Sandler, Andrew Howard, Andrey Zhmoginov Menglong Zhu, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks.

[19] https://github.com/wkentaro/labelme.

[20] https://www.numpy.org/.

[21] https://python-pillow.org/.

[22] https://matplotlib.org/.

[23] https://docs.python.org/2/library/sys.html.

[24] https://docs.python.org/2/library/os.html.

[25] http://host.robots.ox.ac.uk/pascal/voc/voc2012/l.

[26] https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/model_zoo.md.

[27] https://www.cityscapes-dataset.com/.

[28] https://groups.csail.mit.edu/vision/datasets/ade20k/.

[29] François Chollet. Xception: Deep learning with depthwise separable convolutions.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[31] https://medium.com/tensorflow/announcing-tensorflow-2-0-beta-abb24bbfbe3d.

[32] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3.

[33] Eduardo Romera, José M. Álvarez, Luis M. Bergasa, and Roberto Arroyo. Efficient convnet for real-time semantic segmentation.

[34] https://ai.googleblog.com/2019/08/on-device-real-time-hand-tracking-with.html?m=1.

[35] https://ai.googleblog.com/2019/08/exploring-weight-agnostic-neural.html.