



**Universidad de Huelva**  
Escuela Técnica Superior de Ingeniería

**TRABAJO FIN DE GRADO**

**Grado en Ingeniería Informática**

---

# **Deep Learning para clasificación y detección de personas**

---

**Autor: José María Baeza-Herrazti Vázquez**

**Tutor: Diego Marín Santos**

**Cotutor: Manuel Emilio Gegúndez Arias**

Escuela Técnica Superior de Ingeniería de la Universidad  
de Huelva

Octubre 2018, Huelva



# ÍNDICE

<b>RESUMEN .....</b>	<b>3</b>
<b>ABSTRACT .....</b>	<b>4</b>
<b>ÍNDICE DE FIGURAS .....</b>	<b>5</b>
<b>1. PROPUESTA DEL PROYECTO.....</b>	<b>7</b>
1.1. MOTIVACIÓN.....	7
1.2. OBJETIVOS .....	8
• GENERALES.....	8
• ESPECÍFICOS .....	8
1.3. ORGANIZACIÓN DEL PROYECTO.....	9
<b>2. INTRODUCCIÓN AL DEEP LEARNING .....</b>	<b>10</b>
2.1. INTRODUCCIÓN .....	10
2.2. HISTORIA Y EVOLUCIÓN .....	10
2.3. IMPACTO DE DL EN LA SOCIEDAD .....	16
<b>3. FUNDAMENTOS TEÓRICOS .....</b>	<b>19</b>
3.1. REDES NEURONALES CONVOLUCIONALES.....	19
3.1.1. PRINCIPALES TIPOS DE CAPAS .....	20
3.2. TENSORFLOW .....	29
3.3. CLASIFICACIÓN .....	32
3.4. DETECCIÓN .....	37
3.4.1. DETECCIÓN CON FASTER R-CNN.....	39
<b>4. EXPERIMENTACIÓN .....</b>	<b>45</b>
4.1. INTRODUCCIÓN .....	45
4.2. CLASIFICACIÓN .....	45
4.2.1. BASE DE DATOS.....	45
4.2.2. ARQUITECTURA .....	46
4.2.3. ENTRENAMIENTO Y RESULTADOS .....	48
4.2.4. ANÁLISIS .....	50
4.3. DETECCIÓN.....	51
4.3.1. BASE DE DATOS.....	51
4.3.2. ARQUITECTURA .....	52
4.3.3. ENTRENAMIENTO Y RESULTADOS .....	55
4.3.4. ANÁLISIS .....	58

<b>5. CONCLUSIONES .....</b>	<b>59</b>
5.1. PRINCIPALES APORTACIONES .....	59
5.2. LÍNEAS FUTURAS .....	60
<b>6. REFERENCIAS.....</b>	<b>64</b>
<b>7. BIBLIOGRAFÍA .....</b>	<b>65</b>

# RESUMEN

Deep Learning (DL) proporciona una mejora considerable en el ámbito de la visión artificial, en comparación con los algoritmos más tradicionales, logrando niveles de precisión sin precedentes. Permite obtener conocimiento de enormes volúmenes de datos, como por ejemplo, de repositorios de imágenes, videos, textos o conversaciones.

El principal objetivo de este TFG es el aprendizaje de DL desde un punto de vista serio, profundizando en uno de los tipos de Redes Neuronales profundas más utilizadas para la visión artificial en la actualidad, las denominadas Redes Neuronales Convolucionales (CNNs).

En nuestro caso, se ha abordado la aplicación de estas CNNs para clasificar imágenes y detección de objetos de interés. De forma específica, se ha desarrollado un primer sistema de clasificación de imágenes con personas, para, posteriormente, proceder al diseño de un sistema que permite su detección. Para ello, se ha hecho uso del lenguaje de programación Python, de la librería TensorFlow (librería de código abierto para aprendizaje automático, desarrollada por Google) y el entorno de trabajo Jupyter Notebook.

Para realizar la clasificación y detección de personas mediante esta técnica de aprendizaje automático, ha sido necesario recopilar una gran cantidad de imágenes. En el caso del aprendizaje para la detección, además, se ha tenido que etiquetar estas imágenes, señalando mediante rectángulos las personas presentes en cada una de estas.

En cuanto a los resultados obtenidos, en la clasificación se ha conseguido obtener una precisión del 81% de acierto en el conjunto de imágenes de test, y en la detección la precisión no se ha cuantificado, pero si ha sido analizada visualmente y los resultados han sido aceptables.

El objetivo final de este proyecto era el estudio teórico y comprensión de los fundamentos de DL y de las CNNs para posteriormente realizar dos implementaciones de CNNs, una para clasificación de personas en imágenes y otra para detección de personas en imágenes. Dicho objetivo ha sido alcanzado satisfactoriamente.

## **Palabras claves:**

Deep Learning, Machine Learning, Redes Neuronales, Convoluciones, Clasificación, Detección, TensorFlow, Transferencia del aprendizaje, Dataset, Sobreaprendizaje, Inteligencia artificial.

# ABSTRACT

Deep Learning (DL) provides a considerable improvement in the field of machine vision, compared to more traditional algorithms, achieving unprecedented levels of accuracy. It allows to obtain knowledge of huge volumes of data, such as repositories of images, videos, texts or conversations.

The main objective of this TFG is the learning of DL from a serious point of view, deepening in one of the types of deep Neural Networks most used for artificial vision today, the so-called Convolutional Neural Networks (CNNs).

In our case, we have addressed the application of these CNNs to classify images and detect objects of interest. Specifically, a first system for classifying images with people has been developed, in order to subsequently design a system that allows their detection. To this end, the Python programming language, the TensorFlow library (open source library for machine learning, developed by Google) and the Jupyter Notebook environment have been used.

In order to classify and detect people using this machine learning technique, it has been necessary to collect a large number of images. In the case of learning for detection, it has also been necessary to label these images, indicating by rectangles the people present in each of them.

As for the results obtained, in the classification it has been possible to obtain a precision of 81% of success in the set of test images, and in the detection the precision has not been quantified, but it has been analyzed visually and the results have been acceptable.

The final objective of this project was the theoretical study and understanding of the fundamentals of DL and CNNs to later carry out two implementations of CNNs, one for classification of people in images and another for detection of people in images. This objective has been satisfactorily achieved.

## **Key words:**

Deep Learning, Machine Learning, Neural Networks, Convolutions, Classification, Detection, TensorFlow, Learning Transfer, Dataset, Overfitting, Artificial Intelligence.

# ÍNDICE DE FIGURAS

**Figura 1.** Gráfico de las librerías más utilizadas para aplicar DL. (Roger Huang, 2017)

Fuente: <http://code-love.com/2017/06/26/deep-learning-library/>

**Figura 2.** Gráfico del interés de los usuarios sobre DL, basado en las búsquedas en Google. (Elaboración propia, con la ayuda de herramientas de Google) Fuente:

<https://trends.google.com/trends/explore?date=all&q=deep%20learning>

**Figura 3.** Estructura de la red VGGNet. (Naveen Honest Raj, 2017) Fuente:

<https://www.skcript.com/svr/writing-cnn-from-scratch/>

**Figura 4.** Estructura de la red GoogleNet. (Prabhu, 2015) Fuente:

<https://medium.com/@RaghavPrabhu/cnn-architectures-lenet-alexnet-vgg-googlenet-and-resnet-7c81c017b848>

**Figura 5.** Resumen gráfico de los modelos del ILSVRC. (Vieira, 2016) Fuente:

<https://medium.com/@Lidinwise/the-revolution-of-depth-facf174924f5>

**Figura 6.** Ejemplo tratamiento con red neuronal clásica. Fuente: Elaboración propia

**Figura 7.** Explicación de una operación de convolución. (Gabriel Kapellmann-Zafra,

2013) Fuente: [https://www.researchgate.net/figure/Matrix-convolution-filter-algorithm-as-a-graphic-diagram-12\\_fig18\\_304526867](https://www.researchgate.net/figure/Matrix-convolution-filter-algorithm-as-a-graphic-diagram-12_fig18_304526867)

**Figura 8.** Ejemplo Mapa de Características. (Larry Brown, 2014) Fuente:

<https://devblogs.nvidia.com/accelerate-machine-learning-cudnn-deep-neural-network-library/>

**Figura 9.** Ejemplo Stride y Padding. (Mohamed Loey, 2017) Fuente:

<https://www.slideshare.net/mohamedloey/convolutional-neural-network-models-deep-learning>

**Figura 10.** Funciones de activación. (Adil Moujahid, 2016) Fuente:

<https://carchenilla.wordpress.com/2016/06/27/80/>

**Figura 11.** Funciones de agrupación. (Long, 2017) Fuente:

<https://medium.com/@Aj.Cheng/convolutional-neural-network-d9f69e473feb>

**Figura 12.** Capa FC. (Universitat Politècnica de Catalunya, 2017) Fuente:

<https://www.slideshare.net/xavigiro/skin-lesion-detection-from-dermoscopic-images-using-convolutional-neural-networks>

**Figura 13.** Grafo computacional TensorFlow. (Liu Songxiang, 2017) Fuente:

<https://becominghuman.ai/an-introduction-to-tensorflow-f4f31e3ea1c0>

**Figura 14.** Ejemplo sencillo con TensorFlow. (David Rios Y Valles, 2016) Fuente: <http://riosyvalles.com/wp/2016/10/26/small-introduction-to-tensorflow/>

**Figura 15.** Ejemplo clasificación, localización, detección y segmentación por instancia. (Leonardo Araujo Santos) Fuente: [https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/object\\_localization\\_and\\_detection.html](https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/object_localization_and_detection.html).

**Figura 16.** Ejemplo CNN. (Mark Scully, 2017) Fuente: <https://www.slideshare.net/MarkScully5/intro-to-convolutional-neural-networks>

**Figura 17.** Ejemplo de sobreaprendizaje detectado mediante el análisis del error del modelo. (GLUON, 2017) Fuente: [https://gluon.mxnet.io/chapter02\\_supervised-learning/regularization-scratch.html](https://gluon.mxnet.io/chapter02_supervised-learning/regularization-scratch.html)

**Figura 18.** Esquema Faster R-CNN. (Javier Rey, 2018) Fuente: <https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>

**Figura 19.** Ejemplo anclas RPN. (Javier Rey, 2018) Fuente: <https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>

**Figura 20.** Esquema RPN. (Javier Rey, 2018) Fuente: <https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>

**Figura 21.** Parametrizaciones Faster R-CNN. (Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, 2016) Fuente: <https://arxiv.org/pdf/1506.01497.pdf>

**Figura 22.** Modelo de clasificación. Fuente: Elaboración propia

**Figura 23.** Error clasificación. Fuente: Elaboración propia

**Figura 24.** Precisión clasificación. Fuente: Elaboración propia

**Figura 25.** Modelo de detección. Fuente: Elaboración propia

**Figura 26.** Error detección. Fuente: Elaboración propia

**Figura 27.** Pruebas detector Train. Fuente: Elaboración propia

**Figura 28.** Pruebas detector Test. Fuente: Elaboración propia

**Figura 29.** Modelos API TensorFlow. (Repositorio github de TensorFlow, 2017) Fuente: [https://github.com/tensorflow/models/blob/master/research/object\\_detection/g3doc/detection\\_model\\_zoo.md](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md)



# 1. PROPUESTA DEL PROYECTO

## 1.1. MOTIVACIÓN

En esta última década, la Inteligencia Artificial (IA) ha tenido unos avances grandiosos, que han permitido elaborar nuevos sistemas más sofisticados y con mayor precisión que los que existían con anterioridad. Esta evolución en la IA, viene acompañado a su vez de una gran evolución en DL, que posiblemente sea una de las técnicas de IA que más haya impactado a la sociedad en los últimos años, por los resultados obtenidos en algunos proyectos. Se ha llegado a superar incluso el nivel humano para la detección de objetos en imágenes.

La motivación por la que he llevado a cabo este trabajo ha sido por:

- Aprender los fundamentos de DL y realizar una implementación práctica de esta técnica, que en mi opinión, antes de realizar este TFG, pensaba que era fundamental para el avance que ha tenido la IA en los últimos años, y tras finalizarlo, lo sigo pensando, pero aún con más convicción.
- La rama que más me interesa de mi grado es el de la IA, y también pienso que la visión artificial está presente en la mayoría de los grandes problemas actuales a resolver, como pueden ser la conducción autónoma, robótica, automatización, seguridad, etc.
- No existe ninguna asignatura en mi grado en la que se explique algo sobre el DL, y quería que en mi trabajo fin de grado aprendiese algo nuevo que no hubiese visto anteriormente y además hiciese un ejemplo práctico sobre ello, como es el que se ha elegido, que consiste en realizar un clasificador y detector de imágenes con personas.
- Con la elaboración de este proyecto, sabía que adquiriría muchos nuevos conocimientos y además de aprender cómo funciona DL aplicado a un problema de visión artificial, como es la detección de objetos, también aprendería, de forma teórica, cómo aplicarlo a otros tipos de problemas.
- Nunca antes de este proyecto había programado con Python, y la implementación de las CNNs con TensorFlow se tenía que realizar con este lenguaje de programación, de forma que iba aprender otro lenguaje de programación que no enseñan en ninguna de las asignaturas del grado.

## 1.2. OBJETIVOS

### • GENERALES

El objetivo principal es el aprendizaje de los fundamentos, a nivel teórico y práctico, de las CNNs, utilizadas en el ámbito de la clasificación de imágenes y detección de objetos. Para ello se plantea el diseño e implementación de un sistema de clasificación de imágenes con personas y un sistema de detección. Aunque ya existen detectores de imágenes con personas con un alto porcentaje de acierto, la prioridad no es realizar un detector con un alto nivel de precisión, sino aprender las bases y la metodología de cómo hacerlo, sin invertir mucho tiempo del trabajo en la eficiencia y mejora de la precisión del sistema. Cumpliendo con este objetivo, en el futuro, cuando se presente cualquier otro problema que se pueda resolver mediante este tipo de redes, se sabrá cómo resolverlo mediante lo aprendido en este trabajo.

### • ESPECÍFICOS

Para poder alcanzar el objetivo general reseñado, el desarrollo del TFG se ha organizado con los siguientes objetivos específicos:

- Comprender la historia y evolución hasta el día de hoy del DL.
- Aprender los fundamentos del DL y las principales arquitecturas y capas de las CNNs.
- Comprensión y toma de contacto mediante ejemplos sencillos de Machine Learning (ML) y DL, con la librería TensorFlow y Python.
- Estudio y aplicación de técnicas de aumento de datos para imágenes.
- Elaboración de una base de datos con imágenes donde aparezcan personas e imágenes donde no aparezcan personas y etiquetado de dichas imágenes para la utilización de estas durante el entrenamiento de la CNN de clasificación y la CNN de detección.
- Implementación de un sistema clasificador de personas mediante CNN.
- Estudio de los distintos modelos existentes para elaborar un detector mediante CNN.
- Implementación de un sistema detector de personas en imágenes con el modelo Faster R-CNN.

## 1.3. ORGANIZACIÓN DEL PROYECTO

La memoria de este TFG se ha organizado en 5 capítulos de la siguiente forma:

- **CAPÍTULO 1: PROPUESTA DEL PROYECTO:**

En este capítulo se han comentado las razones por las que se ha llevado a cabo este proyecto y se han expuesto los distintos objetivos que han sido marcados para la elaboración de este TFG.

- **CAPÍTULO 2: INTRODUCCIÓN AL DEEP LEARNING:**

En este capítulo se hará una introducción a DL, se verá su historia y su evolución en los últimos años y, por último, se explicará cómo se encuentra actualmente el estado de esta técnica de aprendizaje automático, viendo algunos campos donde se está aplicando en la actualidad.

- **CAPÍTULO 3: FUNDAMENTOS TEÓRICOS:**

En este capítulo se explicarán los fundamentos teóricos necesarios de aprender para cumplir con los objetivos propuestos y se hará especial hincapié en explicar detalladamente cómo funciona cada una de las capas existentes en una CNN.

- **CAPÍTULO 4: EXPERIMENTACIÓN:**

En este capítulo se detallará la experimentación, donde se introducirá y se explicará cómo se ha elaborado el sistema de clasificación y el sistema de detección de imágenes con personas, y, a su vez, se mostrará los distintos resultados obtenidos por ambos sistemas.

- **CAPÍTULO 5: CONCLUSIONES:**

En este último capítulo se verán las conclusiones obtenidas por la realización de este trabajo, y se comentará algunas futuras mejoras y aplicaciones del sistema de detección de personas realizado.

## 2. INTRODUCCIÓN AL DEEP LEARNING

### 2.1. INTRODUCCIÓN

En este capítulo se aborda la historia que ha tenido DL, desde que nacieron las redes neuronales, hasta su situación actual. Aunque parezca que es muy reciente, no es así, como veremos a continuación. De esta forma, se tratará su evolución justificando el avance tan grande que ha tenido en los últimos años y porque no había ocurrido antes.

Además se presentará el estado actual de esta técnica de aprendizaje automático, reseñando las investigaciones relevantes y los últimos logros conseguidos.

### 2.2. HISTORIA Y EVOLUCIÓN

Se va a comenzar citando los hitos y comentando un poco la historia de las redes neuronales, de forma resumida, y luego se verá más en profundidad cómo surgió el término DL a partir de las redes neuronales artificiales y su evolución.

Todo comenzó con McCulloch y Pitts en 1943 (<http://www.mind.ilstu.edu/curriculum/modOverview.php?modGUI=212>), definiendo formalmente la neurona como una maquina binaria con varias entradas y salidas.

En 1949, Hebb, definió dos conceptos de mucha importancia para el campo de las redes neuronales ([1]):

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

Frank Rosenblatt en 1958 introdujo el Mark I Perceptron ([2]), consistente en simular un proceso de decisión basado en la suma ponderada de las entradas al sistema, y ofrece un 0 si el resultado no supera un cierto umbral, y un 1 en caso contrario. En esta máquina, los pesos de la red se ajustaban mediante potenciómetros, en función del error obtenido en la salida, ante una entrada conocida.

En 1959, la universidad de Stanford creó las redes neuronales conocidas como ADALINE y MADALINE ([3]). La red MADALINE sirve para eliminar el ruido existente en las comunicaciones telefónicas.

En 1969, el MIT publicó un libro titulado “Perceptrons” ([3]), en el que argumentaban, entre otras cosas, que el concepto del Perceptron no podía ser aplicado a redes

multicapa, dado que el cálculo de los valores correctos de los pesos requeriría de un número demasiado grande de operaciones, que no eran posibles de calcular para la capacidad de los ordenadores de la época. Esto hizo que hubiera un gran desinterés para realizar investigaciones sobre las redes neuronales, durante unos 14 años.

Hasta 1985, no surgió el algoritmo denominado Descenso por Gradiente, publicado por el MIT. Este algoritmo, tiene la función de calcular el ajuste que hay que hacerle a cada peso de la red, para que la salida de esta, sea la deseada, en base a una entrada conocida.

No es hasta 1990, cuando de verdad toman un gran impulso las redes neuronales, debido a los avances en el hardware de los ordenadores, lo que permitiría el entrenamiento de las redes en un periodo razonable de tiempo.

Es entonces cuando nace el término de DL, que consiste en añadir capas de neuronas intermedias a la red, permitiendo realizar clasificaciones no lineales complejas. Las redes neuronales tradicionales solo contienen dos o tres capas ocultas, mientras que las redes profundas pueden llegar a tener muchísimas más (en la actualidad se ha llegado hasta las 150 capas ocultas). Estas redes, al tener más neuronas, también tenían más pesos que se debían ajustar en la fase de entreno y por tanto, eran necesario ordenadores más potentes que los que existían en aquella época. Además, era necesario una gran cantidad de datos para entrenar estas redes, e internet, que hoy en día es nuestra principal fuente de información y obtención de datos, acababa de nacer.

A continuación, vamos a explicar la historia de las CNNs, que es el tipo de red profunda que se ha tratado en este TFG.

Fue en 1980 ([4]), cuando Kunihiko Fukushima, desarrolló un trabajo que condujo al desarrollo de las primeras CNNs, que se basan en la organización de la corteza visual que se encuentra en los animales.

A inicios de los 90, LeCun ([5]), diseñó e implementó LeNet-5, una CNN, con 5 capas ocultas que clasifica dígitos escritos a mano. Con esta red, se obtuvo una precisión del 99.5%, superando a cualquier otra técnica de ML para un problema de visión por computador. Es entonces cuando se demuestra el gran potencial de estas redes neuronales. Debido a las limitaciones existentes en relación al hardware, LeCun tuvo que utilizar imágenes de 32x32 píxeles (imágenes bastante pequeñas, con muy poca resolución), para que el entrenamiento se pudiese realizar en un periodo de tiempo razonable.

No fue entonces hasta 2009, con el lanzamiento de ImageNet, cuando llegó la revolución de DL en los últimos años. ImageNet es una base de datos de imágenes, muy extensa y gratuita. Esto permitió, gracias también al aumento de la potencia del hardware de los ordenadores, la posibilidad de que cualquier persona que se interesara por estas técnicas de aprendizaje automático y pudieran realizar pruebas en sus propios ordenadores. Aún, el hardware era una limitación y aunque ya existían ordenadores con gran poder computacional, eran demasiado caros.

Allá por 2012, AlexNet, una CNN, ganó las olimpiadas de visión por computador (ILSVRC), consiguiendo una muy superior a la conseguida por el segundo clasificado de la competición, que utilizó otras técnicas de ML. Además, superó por mucho la precisión que se obtuvo en ediciones anteriores del campeonato. Esto provocó un asombro en la comunidad de visión por computador.

En Noviembre de 2015, Google liberó la librería que había desarrollado y utilizado en los anteriores años para aprendizaje automático: TensorFlow. Hoy en día, esta librería es la más utilizada para la creación de CNNs y el uso de DL.

Se puede observar en la figura 1 un gráfico donde se visualizan las librerías más utilizadas en la actualidad según un estudio realizado sobre los repositorios de DL existentes en GitHub:

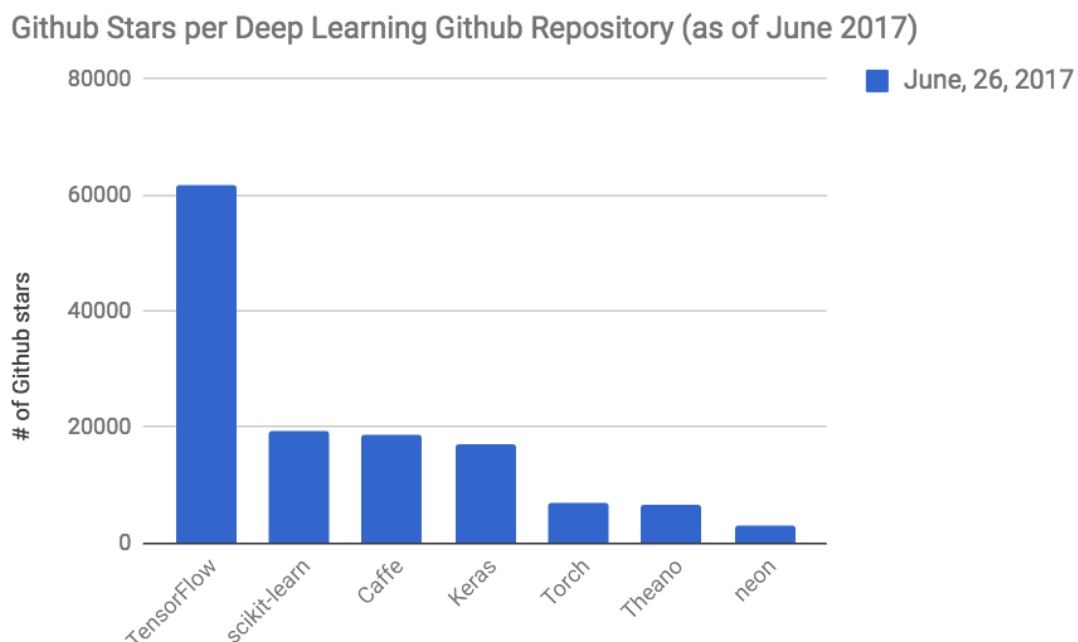


Figura 1. Gráfico de las librerías más utilizadas para aplicar DL. Fuente: <http://code-love.com/2017/06/26/deep-learning-library/>

Como se ha dicho anteriormente, TensorFlow es la librería más utilizada con bastante diferencia sobre otras importantes librerías, como Scikit-Learn, Caffe o Keras.

A continuación se mostrará el interés que ha tenido el DL, en base a las búsquedas realizadas en Google por los usuarios de todo el mundo, mediante la figura 2, donde los números reflejan el interés de búsqueda en relación con el valor máximo del gráfico. Un valor de 100 indica la popularidad máxima de un término, mientras que 50 y 0 indican

que un término es la mitad de popular en relación con el valor máximo o que no había suficientes datos del término, respectivamente:



Figura 2. Gráfico del interés de los usuarios sobre DL, basado en las búsquedas en Google. Fuente: <https://trends.google.com/trends/explore?date=all&q=deep%20learning>

Se observa como a partir del 2012, con el comentado hito de AlexNet, el interés de las personas por el DL asciende considerablemente y cómo hoy en día sigue aumentando el interés por ello.

Este aumento de interés en los últimos años por el DL es debido principalmente a 3 causas:

- Aumento exponencial de la cantidad de datos existentes en internet, libres de descarga para el público, que hace posible el entrenamiento de estas técnicas de ML.
- Aumento de la potencia del hardware de los ordenadores, y a su vez, disminución de los costes de estos, lo que permite su compra a más personas del mundo.
- Asombrosos resultados del DL aplicados a distintos problemas como la visión artificial, aprendizaje por refuerzo, conducción autónoma y otros problemas resolubles mediante aprendizaje automático.

A continuación se verá cómo ha sido la evolución de las CNNs aplicadas a la visión artificial, mediante la comparativa entre las redes que han ganado los últimos años la competición ILSVRC.

Como ya se ha comentado, en el año 2012, por primera vez, se utilizó una CNN para la competición ILSVRC. Esta red la apodaron AlexNet, y tenía hasta 8 capas ocultas. Su tasa de error con respecto al top 5 fue de 15.4% (el error top 5 es la tasa a la que, dada una imagen, el modelo no da la etiqueta correcta en sus 5 mejores predicciones). El siguiente mejor modelo logró un 26.2% de tasa de error, que era lo que venían consiguiendo los modelos ganadores de años anteriores. Esto provocó un gran asombro y fue el inicio del gran uso de este tipo de redes para tratar los problemas de visión artificial en las posteriores ediciones del campeonato.

En 2013, volvió a ganar otra CNN, a la cual apodaron ZF Net. Alcanzó una tasa de error del 11.2% con respecto al top 5, y tenía 8 capas ocultas, al igual que AlexNet. El equipo desarrollador de esta red, reconoció que el gran interés actual por las CNNs era debido a la actual gran accesibilidad a muchos datos y al aumento de la potencia computacional con el uso de las GPU's.

En 2014, se crearon dos CNNs que tuvieron muy baja tasa de error. Estas fueron VGG Net y GoogleNet, con una tasa de error del 7.3% y 6.7%, respectivamente. VGG Net se implementó con 19 capas ocultas, mientras que Google Net, con 22. GoogleNet cambió un poco la secuencia que se venía siguiendo en las capas de las CNNs durante años anteriores. Implementó en su red, una nueva capa, denominada *Inception Layer*; además, introdujo una estructura paralela en ciertas partes, en lugar de la secuencial utilizada en el resto de redes. En las figuras 3 y 4 se muestran las estructuras de los modelos anteriores y la estructura de GoogleNet, para ilustrar visualmente la diferencia entre ambas.

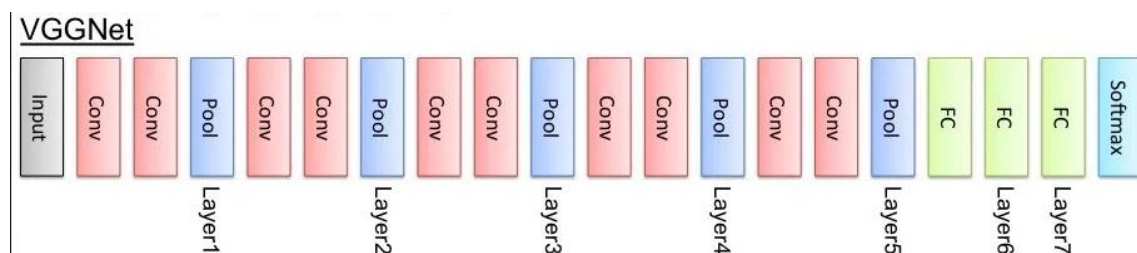


Figura 3. Estructura de la red VGGNet. Fuente: <https://www.skcript.com/svr/writing-cnn-from-scratch/>



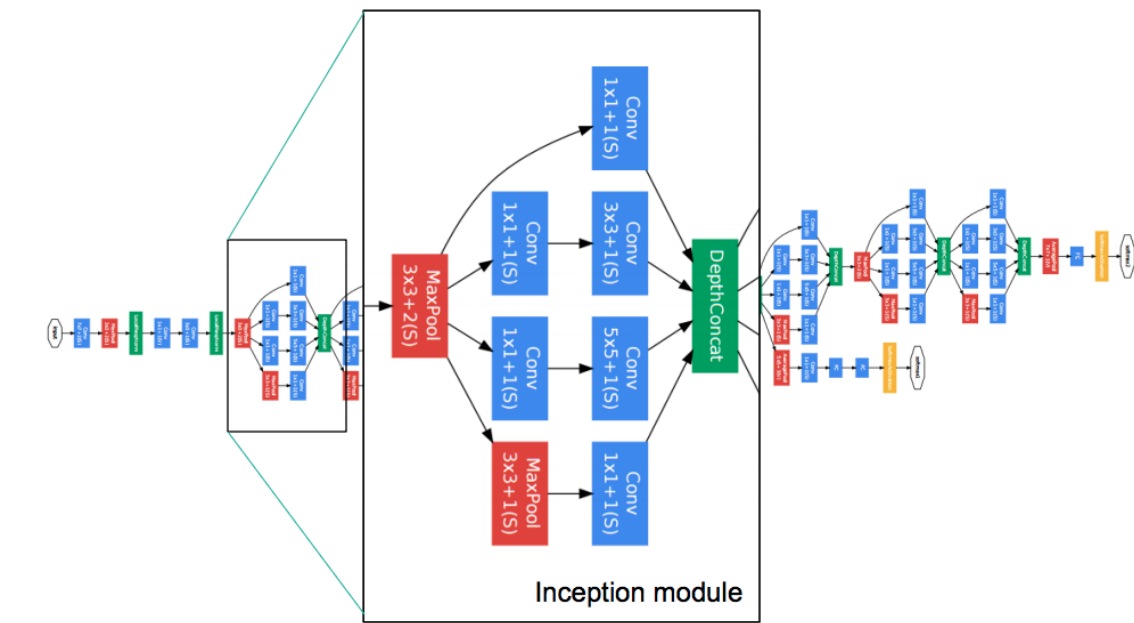


Figura 4. Estructura de la red GoogleNet. Fuente:

<https://medium.com/@RaghavPrabhu/cnn-architectures-lenet-alexnet-vgg-googlenet-and-resnet-7c81c017b848>

Fue en el año 2015 cuando el modelo ResNet ganó la competición con una tasa de error del 3.6% con respecto al top 5, superando así la tasa de error de los humanos, que está situada en torno a un 5%. Los desarrolladores de este modelo, al igual que los de GoogleNet, innovaron e implementaron una nueva capa en su red, denominada capa *Residual*. Esta red fue la más profunda hasta el momento, con 152 capas ocultas.

A continuación, mostraremos un gráfico (figura 5), que hará de resumen de lo que se ha visto sobre el campeonato de visión artificial, ILSVRC. En el gráfico se aprecia como a partir de la aparición de la primera CNN, la tasa de error tiene una bajada importante con respecto a la tasa de error del año anterior a dicha aparición. A su vez, se observa cómo desde 2010, cada año se consiguen mejores modelos, con menor tasa de fallo. En la figura también se visualiza el número de capas que utiliza cada red, que como se muestra, cada año van aumentando cada vez más el número de capas ocultas utilizadas por cada modelo.

## Revolution of Depth

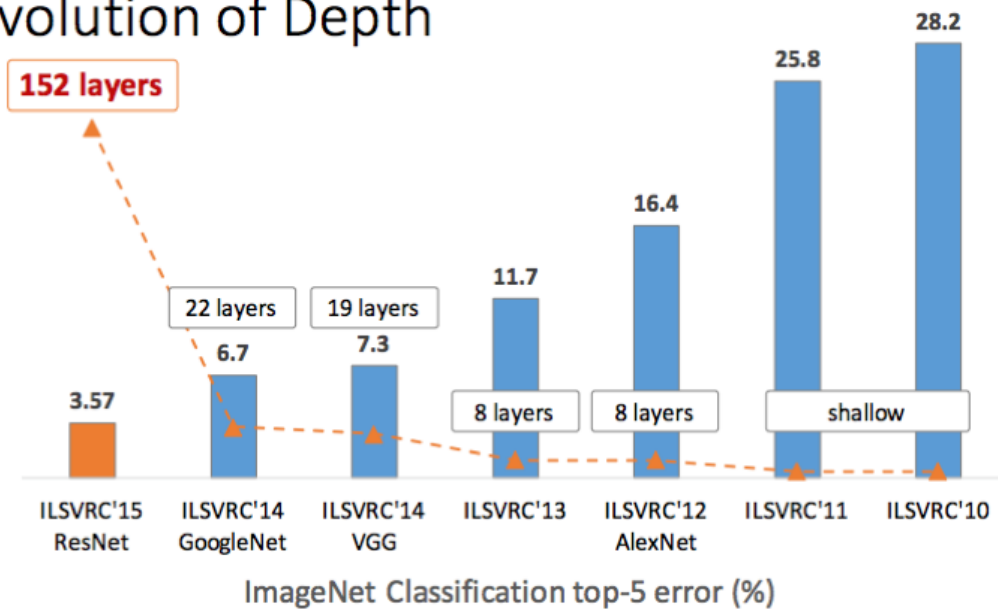


Figura 5. Resumen gráfico de los modelos del ILSVRC. Fuente: <https://medium.com/@Lidinwise/the-revolution-of-depth-fac174924f5>

### 2.3. IMPACTO DE DL EN LA SOCIEDAD

En la actualidad, DL ha ganado una gran popularidad en el sector de la IA y se han conseguido desarrollar sistemas que hace unos pocos de años se pensaba que serían intratables en corto plazo. Pero aún, esta técnica sigue evolucionando y se utiliza cada vez más a la hora de resolver un problema de ML. Dentro de las distintas técnicas existentes en ML, en la actualidad, puede decirse que DL es la técnica más utilizada en la industria. Se puede ver implementado en multitud de problemas, como la conducción autónoma, traductores en tiempo real, procesadores del lenguaje natural, etc.

A continuación se verán los campos que han conseguido un gran avance con la ayuda de técnicas de DL.

- **Conducción autónoma:**

Desde hace unos 5 años, la conducción autónoma ha tenido unos avances grandiosos, empezando incluso a comercializarse coches que ya conducen de forma autónoma, en determinadas calzadas. Eso es debido, en parte, a la llegada del DL, ya que gracias a este, el problema principal que existía en los vehículos autónomos, de que no sabían cómo detectar lo que está a su alrededor, como pueden ser peatones, señales de tráfico, semáforos, otros vehículos, etc., se han resuelto, mediante la visión artificial, con técnicas de DL. Además, la precisión de estos sistemas de detección existentes en los vehículos autónomos, han llegado a niveles de precisión que hace años era impensable llegar, teniendo en cuenta la multitud de objetos que detecta a la vez.

El sistema desarrollado en este trabajo, podría tener cómo aplicación la detección de personas mediante imágenes para un vehículo autónomo.

- **Traductores:**

En 2017, se creó DeepL, un traductor online que logró superar la precisión del traductor de Google, Google Translate, que era el referente hasta esa fecha. Este sistema utiliza DL para lograr su gran eficacia en la traducción de idiomas.

Google, también ha implementado estas técnicas en su traductor para resolver el problema de la traducción de palabras existentes en imágenes. Ahora, con Google Translate, es posible hacerle una foto a un cartel donde haya palabras, y automáticamente se traduce al idioma que se elija.

Las últimas innovaciones vienen para resolver el problema de traducción en tiempo real y sin necesidad de conexión a internet, para lo que la mayoría de proyectos hacen uso de redes neuronales profundas.

- **Procesadores del lenguaje natural:**

El procesamiento del lenguaje natural, desde hace varios años, es un problema que se da con bastante frecuencia en los asistentes virtuales, por ejemplo. Consiste en la comprensión del habla humana. Esto tiene muchísimas aplicaciones, cómo en los traductores, citado anteriormente, para que en caso de desarrollar un sistema que traduce lo que una persona habla, el sistema pueda saber lo que está diciendo dicha persona. También es muy utilizado, por ejemplo, como asistente virtual en los teléfonos móviles.

Además, se espera que en los próximos años, la necesidad de desarrollar sistemas que necesiten procesar el lenguaje natural, seguirán aumentando como lo ha hecho hasta ahora.

Los últimos avances que se han conseguido en este campo han sido protagonizados por redes neuronales profundas, con las que se han obtenido unos niveles de precisión en los sistemas desarrollados que nunca antes se habían conseguido con otras técnicas.

- **Predicción de precios:**

Con la llegada de las criptomonedas, muchos han sido los proyectos que han surgido sobre Trading automático. Muchos de estos proyectos utilizan DL para la predicción de precios, entre otras cosas. También se utiliza mucho para la detección y análisis de las emociones del mercado, mediante la extracción de la información que producen los usuarios en diferentes plataformas, como Twitter, u otras redes sociales, obteniendo así conocimiento mediante esta técnica de ML.

Por último, se van a comentar dos noticias recientes, en las que se indican los últimos proyectos en los que se ha enfocado la empresa Nvidia, en los cuales, han hecho uso de técnicas de DL. Uno de sus últimos desarrollos, ha sido la creación de un sistema capaz de crear un video a cámara lenta, a partir del video original, grabado a una velocidad normal, mediante el uso del DL. También, en otra noticia, informan de que han logrado crear un sistema que elimina el ruido presente en imágenes, mediante el uso de CNN.

A continuación se ofrecen los links a ambas noticias:

- Videos cámara lenta: <https://www.xataka.com/investigacion/nvidia-capaz-crear-videos-casi-perfectos-camara-lenta-cualquier-video-velocidad-normal>
- Eliminación del ruido: <https://www.albedomedia.com/tecnologia/la-ia-de-nvidia-que-elimina-el-ruido-de-las-imagenes-usando-solo-el-ruido/>

### 3. FUNDAMENTOS TEÓRICOS

En este capítulo, se va a explicar los conceptos teóricos que se han aprendido sobre las CNNs y se verán las distintas capas que pueden encontrarse en una red de este tipo y sus funcionalidades.

También se explicará cómo funciona la librería TensorFlow, la cual se ha utilizado para desarrollar tanto el sistema de clasificación como el de detección. Esta librería tiene unas particularidades que se ha pensado que son necesarias de explicar para comprender correctamente su funcionamiento.

Por último se incluyen dos apartados, uno dedicado a la parte de clasificación y otro a la detección, donde se verá la teoría necesaria para poder elaborar ambos sistemas mediante CNNs.

#### 3.1. REDES NEURONALES CONVOLUCIONALES

Las CNNs son muy similares a las redes neuronales clásicas. Están formadas por neuronas que tienen pesos y sesgos que se pueden ajustarse. La diferencia entre las CNNs y las redes neuronales clásicas es que las primeras suponen explícitamente que la entrada se trata de un conjunto de datos invariante en cuanto a que no se pueden intercambiar las columnas y/o filas entre sí, como es el caso de las imágenes o los espectrogramas, que al hacer dichos tipos de intercambios dejarían de ser datos útiles. Esto permite a las CNNs codificar ciertas propiedades en la arquitectura y hace que el entrenamiento sea muchísimo más eficiente y el número de parámetros a ajustar por la red se reduzca considerablemente.

Se comprenderá mejor con un ejemplo. Al tratar de introducir una imagen en una red neuronal clásica, si la imagen es medianamente grande, como por ejemplo, el tamaño de imagen que se ha utilizado para este problema (300x300), se tendría un total de 90.000 píxeles por cada imagen, en ese caso, y cada pixel tendría una conexión a cada neurona de la primera capa. Asumiendo una red con 1000 neuronas en la primera capa (figura 6), se tendrían que ajustar 90.000.000 de pesos.

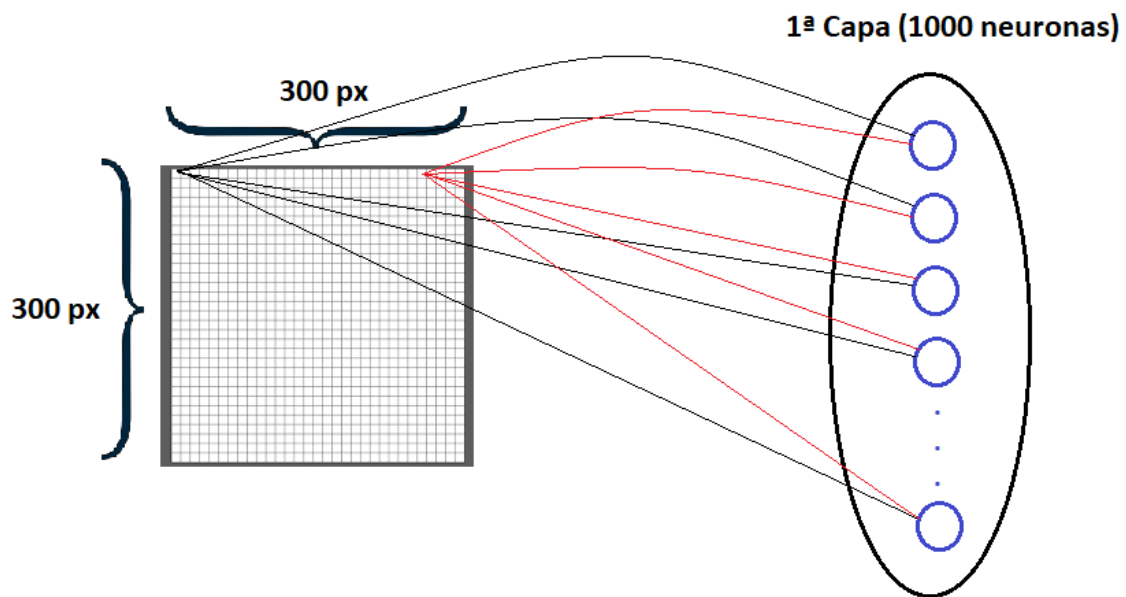


Figura 6. Ejemplo tratamiento con red neuronal clásica. Fuente: Elaboración propia

Sin embargo, las CNNs trabajan de forma distinta. Existen capas convolucionales, que consisten en uno o varios filtros que se van deslizando sobre la imagen, realizando operaciones de convolución. Cada uno de estos filtros están compuestos de neuronas. Si por ejemplo, en una primera capa oculta se tiene una capa convolucional de  $7 \times 7 \times 128$ , esto significa que tiene 128 filtros de  $7 \times 7$  neuronas, que se van a ir deslizando por la imagen, como se explicará más adelante, realizando operaciones de convolución. En ese ejemplo, la red tendría que aprender  $7 \times 7 \times 128 = 6.272$  pesos para la primera capa. Como se ve, la diferencia con respecto al ejemplo de la red neuronal clásica en cuanto al número de pesos a aprender es muy grande.

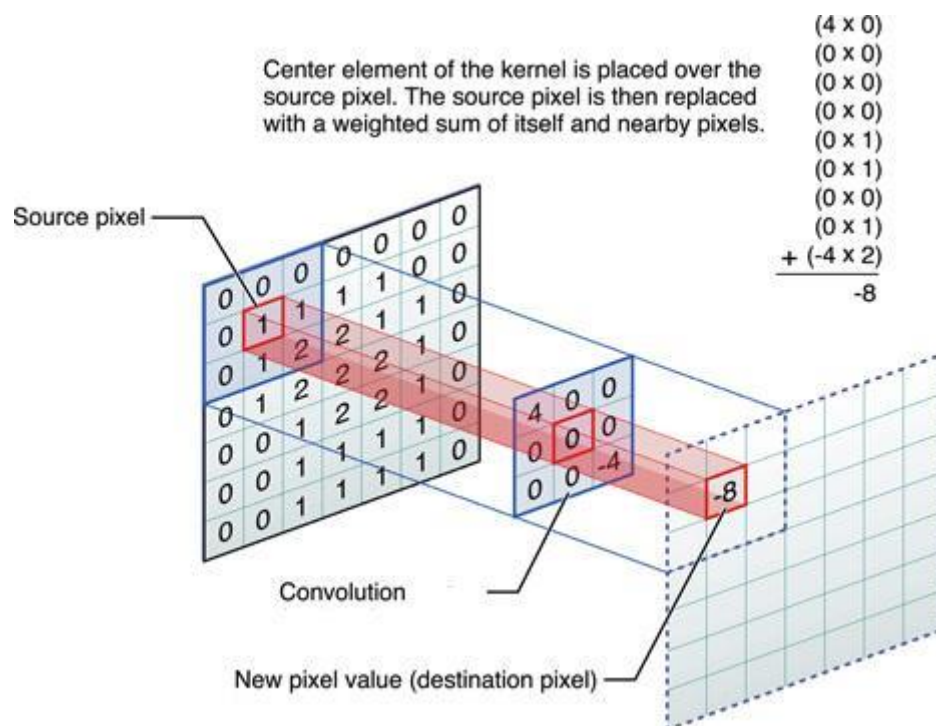
### 3.1.1. PRINCIPALES TIPOS DE CAPAS

A continuación, se va a explicar detalladamente los distintos tipos de capas que pueden encontrarse en una CNN.

- **Capa Convolucional:**

Estas capas son las encargadas de hacer la operación más característica de las CNNs, las convoluciones. Dada la importancia de esta operación a continuación se va a realizar una breve explicación del fundamento de esta operación.

- **Operación de convolución:** La operación de convolución consiste en el sumatorio de las multiplicaciones elemento a elemento entre dos matrices del mismo tamaño. A continuación, se va a mostrar un ejemplo de una convolución (figura 7). Podréis observar que el resultado de la convolución se sitúa en la posición central con respecto a la porción de la imagen en la que se ha situado el filtro, en una nueva matriz de las mismas dimensiones que la imagen.



- Figura 7. Explicación de una operación de convolución. Fuente: [https://www.researchgate.net/figure/Matrix-convolution-filter-algorithm-as-a-graphic-diagram-12\\_fig18\\_304526867](https://www.researchgate.net/figure/Matrix-convolution-filter-algorithm-as-a-graphic-diagram-12_fig18_304526867)

Esta capa recibe como entrada una matriz y se realizan convoluciones sobre ella mediante un filtro que se irá deslizando por dicha matriz de entrada. Una vez se le han hecho las convoluciones, se obtiene una nueva matriz, que será la salida de esta capa. Esta matriz de salida, luego se introduce a una capa de regulación, de las que se hablará luego, y a la salida de esa capa de regulación se le suele llamar mapa de características, debido a que al aplicarle filtros mediante convoluciones a la matriz de entrada, la matriz de salida contiene características de dicha entrada.



Las CNNs suelen tener varias capas convolucionales seguidas de regularizaciones. La primera de ellas, tiene como entrada, en caso de que estemos ante un problema de visión artificial, la imagen a tratar. Tras la primera capa convolucional, se obtendrá un mapa de características de bajo nivel, en las que se podrían apreciar bordes horizontales, bordes verticales, etc. Ese mapa de características irá pasando por varias capas convolucionales más, de forma que cada vez se va a ir obteniendo mapas de características con características de más alto nivel, llegando a detectar, por ejemplo orejas, caras, pelos, patas, etc.

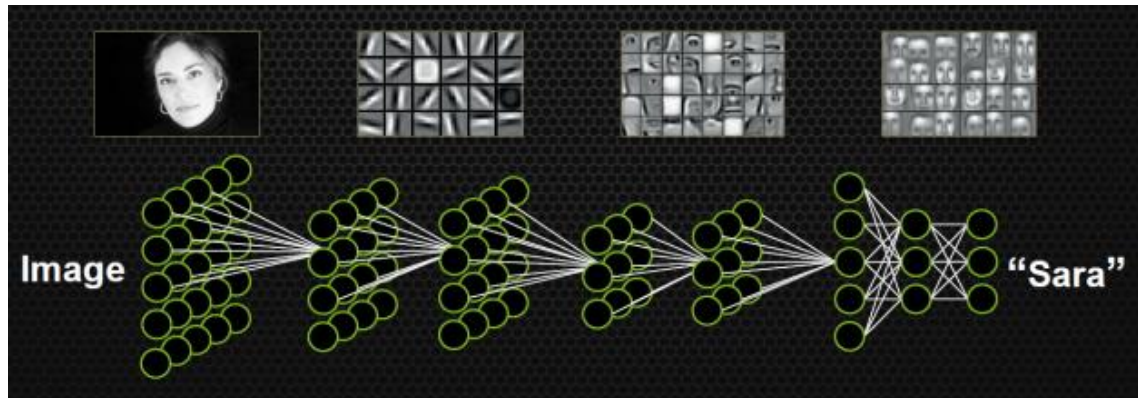


Figura 8. Ejemplo Mapa de Características. Fuente:  
<https://devblogs.nvidia.com/accelerate-machine-learning-cudnn-deep-neural-network-library/>

En la figura 8, se puede observar a modo de resumen, la función de las capas convolucionales con regularización. Podéis ver que la entrada de la CNN es una imagen, y que tras la primera convolución, se obtiene un mapa de características de bajo nivel. Conforme esos mapas de características van pasando por más capas convolucionales, se va obteniendo características de más alto nivel, hasta llegar incluso a detectar el objeto deseado, ignorando los demás objetos o fondo.

Se va a hablar ahora de cómo funciona esta capa y se comentará los parámetros que tiene que determinar el programador.

Esta capa consiste en varios filtros de  $N \times N$  dimensiones, en la que cada casilla del filtro es el peso de una neurona. Por tanto, lo que se aprende en estas capas convolucionales, no es el peso de las conexiones entre neuronas como en las redes neuronales clásicas, sino el valor de cada casilla de cada máscara.

Lo que se hace en esta capa es ir deslizando cada filtro por la matriz de entrada, realizando operaciones de convolución. La región por la que está la máscara sobre la matriz de entrada en un momento determinado, se le denomina campo receptivo.

El filtro, como se ha comentado, se va deslizando sobre la matriz de entrada. Ese deslizamiento lo determina el programador, mediante un parámetro llamado *stride*, que indica el salto que va dando el filtro tras cada convolución. El stride se suele



ajustar de manera que la matriz de salida tenga dimensiones enteras y no una fracción.

Otro parámetro a determinar por el programador es el *padding*, que consiste en el relleno que se hace sobre los bordes de la matriz de entrada. Normalmente se utilizan ceros para rellenar los bordes, denominándose así zero-padding. Esto es útil cuando se quiera conservar toda la información posible de la matriz de entrada, incluido los bordes de esta, o bien, también es necesario determinarlo cuándo establecemos un stride que haga que parte del filtro, en algún deslizamiento, quede fuera de la matriz de entrada.

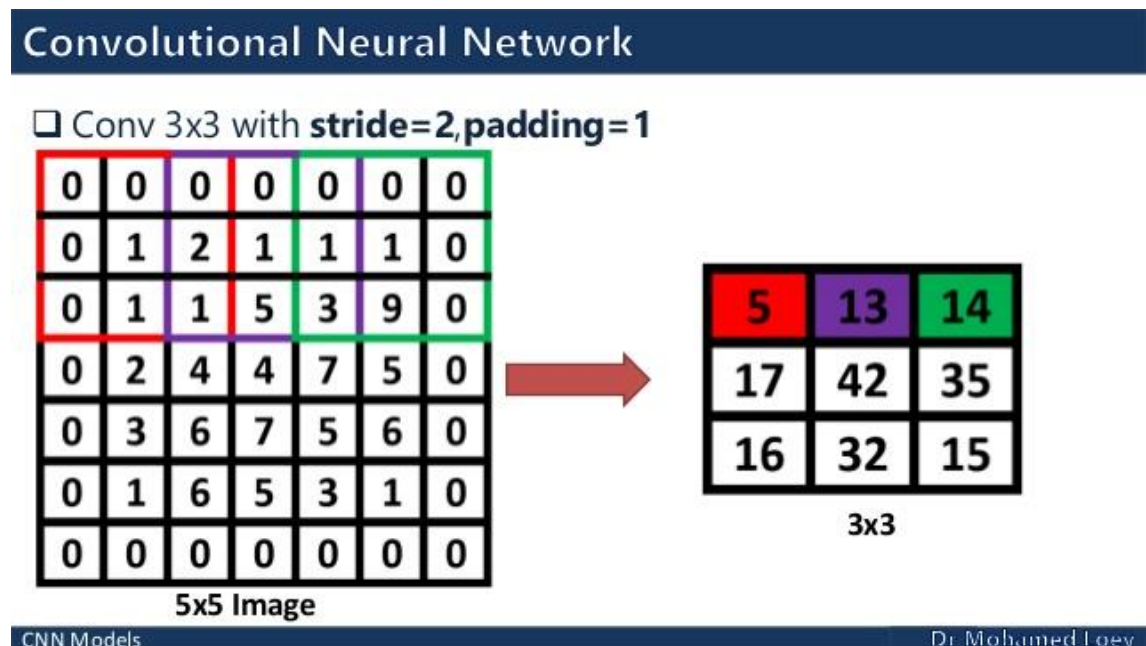


Figura 9. Ejemplo Stride y Padding. Fuente:  
<https://www.slideshare.net/mohamedloey/convolutional-neural-network-models-deep-learning>

En la figura 9 se puede ver un ejemplo en el que se asume un filtro de tamaño 3x3 en el que cada casilla tiene un valor de 1 y un stride de 2 (es decir, una vez haga una convolución, la máscara va a saltar dos píxeles hacía el lado para hacer otra convolución, o en caso de haber llegado a la derecha del todo de la imagen, saltará dos píxeles hacia abajo). También veis que tiene un padding de ceros, con un tamaño de 1 píxel de borde de la imagen. A la derecha veis lo que sería el mapa de características resultante, tras hacer todas las convoluciones.

Normalmente, los programadores usan el zero-padding para no perder la información existente en los bordes de la matriz de entrada. En función del tamaño de filtro que utilicen y el stride, utilizarán un padding más o menos grande. Por

ejemplo, si se utiliza un stride de 1, si el tamaño del padding, P, se ajusta mediante la siguiente fórmula (Eq. 1):

$$P = \frac{K-1}{2}$$

Eq. 1

dónde K es el tamaño del filtro, entonces el tamaño de la matriz de salida será el mismo que el tamaño de la matriz de entrada (mantiene las mismas dimensiones).

Otra ecuación de interés es la que se muestra a continuación (Eq. 2). Permite calcular el tamaño de salida, O, para cualquier capa convolucional:

$$O = \left( \frac{W-K+2P}{S} \right) + 1$$

Eq. 2

dónde:

- W: Dimensión de la entrada
- K: Tamaño filtro
- S: Stride

Y, ¿cómo se puede saber cuántas capas convolucionales utilizar, qué tamaño de filtros usar, qué valor se le dan al stride y padding?

Estas no son preguntas triviales y no existe aún un estándar establecido que sea utilizado por todos los investigadores. Esto se debe a que la red dependerá en gran medida del tipo de datos que tenga que tratar. Los datos pueden variar según el tamaño, la complejidad de la imagen, el tipo de tarea de procesamiento de imágenes y mucho más. Cuando se observa su conjunto de datos, una manera de pensar en cómo elegir los hiperparámetros es encontrar la combinación correcta que crea abstracciones de la imagen a una escala adecuada.

Lo que sí existe es una restricción a la hora de determinar el número de filtros de una capa convolucional. La tercera dimensión del tamaño de los filtros de una capa convolucional, que indica el número de filtros a utilizar, debe ser la misma que la tercera dimensión de la matriz de entrada. Por ejemplo, si se tiene una imagen RGB (3 canales) y se introduce en una capa convolucional, los filtros de esta capa deben

tener tres canales también (AltoxAncrox3) y cada uno actuará sobre el canal correspondiente de la imagen.

Existe también una forma de estructurar las CNNs de forma que se puede reducir la cantidad de pesos que deben ajustarse sin disminuir la capacidad de aprendizaje de la red, utilizándose varias capas seguidas con filtros de tamaño pequeño, en vez de utilizar una sola capa convolucional con tamaño de filtros grande. Esta novedad fue introducida por los autores de la red VGG Net que compitió en la ILSVRC en 2014. Ellos utilizaron filtros de  $3 \times 3$ , debido a que la combinación de dos capas convolucionales  $3 \times 3$  tiene un campo receptivo de  $5 \times 5$ . Esto a su vez, simula un filtro más grande, mientras que mantiene los beneficios de tamaño de filtros más pequeños. Uno de los beneficios es la disminución del número de parámetros, por ejemplo, tres capas convolucionales seguidas de  $3 \times 3$ , tendrían un campo receptivo de  $7 \times 7$ , pero si se usan tres capas de  $3 \times 3$ , el número de parámetros que debe aprender la red en cada filtro es de  $3 \times 3 = 9$ , si se le suma a demás que son 3 capas convolucionales,  $9 \times 3 = 27$  parámetros que tiene que aprender por cada 3 filtros  $3 \times 3$ , mientras que con un solo filtro de tamaño  $7 \times 7$ , la red debería de aprender  $7 \times 7 = 49$  parámetros por filtro.

En estas capas, también existe el parámetro *bias*, como en las redes neuronales clásicas, que son pesos que se suelen poner con un valor inicializado a 1 y que la red aprenda a adaptarlos al conjunto de datos de entrenamiento. Existirá un bias por cada filtro que se utilice, es decir, si se utiliza, por ejemplo, 10 filtros de tamaño  $5 \times 5 \times 3$ , el número de pesos en total que debe de aprender la red para esa capa es de  $5 \times 5 \times 3 \times 10 = 750$ , a lo que habría que sumarle los 10 bias,  $750 + 10 = 760$ , y ese sería el total de número de parámetros que tiene dicha capa convolucional.

- **Capa de activación:**

El propósito de esta capa es introducir la no linealidad a un sistema que básicamente ha estado computando las operaciones lineales durante las capas convolucionales (sólo se realizan sumas y multiplicaciones en dichas capas). En el pasado, se utilizaron funciones no lineales como la tangente hiperbólica o la sigmoide, pero los investigadores descubrieron que las capas Relu funcionan mucho mejor porque la red es capaz de entrenar un poco más rápido, sin haber una diferencia significativa en la precisión ([6]). La función de activación Relu, también ayuda a aliviar el problema de los gradientes de fuga, por el cual las capas inferiores de la red se forman muy lentamente porque el gradiente disminuye exponencialmente a través de las capas.

En la figura 10 se muestran algunas de las funciones de activación más utilizadas para las CNNs.

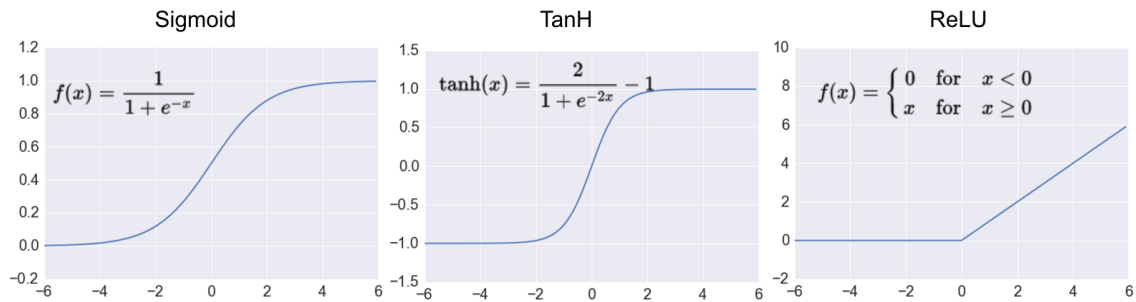


Figura 10. Funciones de activación. Fuente:  
<https://carchenilla.wordpress.com/2016/06/27/80/>

La función de activación más utilizada en la actualidad es la función ReLU, que en resumen, funciona pasándole como entrada una matriz, a la cual, esta función cambia los números negativos que tenga por ceros, y devuelve dicho resultado como salida.

Esta capa aumenta las propiedades no lineales del modelo sin afectar a los campos receptivos de la capa convolucional. Generalmente se utiliza tras una capa convolucional.

- **Capa de agrupación (pooling):**

Las capas de agrupación, también conocidas como capas de *pooling*, se suelen utilizar tras una capa de activación. Los programadores suelen optar generalmente por aplicar esta capa tras cada capa de activación, pero también es muy común no aplicar esta capa tras haber utilizado varias veces seguidas capas de activación.

En esta categoría, al igual que en la capa de activación, también hay varias opciones de funcionamiento, siendo la *maxpooling* la más popular y utilizada, la cual consiste en la aplicación de una máscara de máximos con un stride igual a la longitud de la máscara.

También existen otros métodos de agrupación, como la media, que consiste en hacer la media de los valores que caen dentro del filtro y dicho valor es el que se emite al volumen de salida, pero este método no se utiliza debido a que es bastante más lento que el maxpooling y no se gana una eficacia significativa.

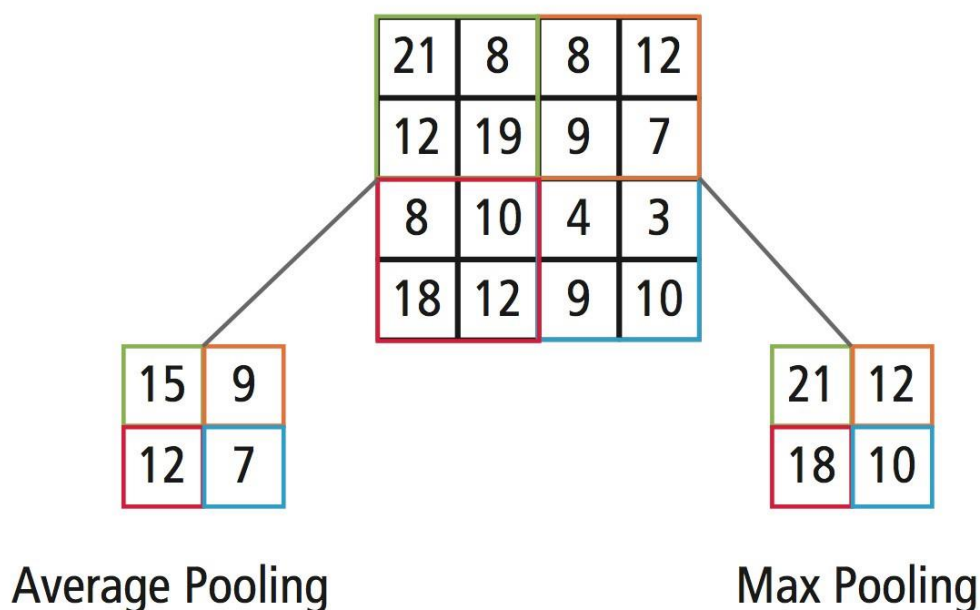


Figura 11. Funciones de agrupación. Fuente:  
<https://medium.com/@Aj.Cheng/convolutional-neural-network-d9f69e473feb>

En la figura 11, podéis observar un ejemplo de agrupación mediante el cálculo de la media (en la parte izquierda de la imagen) y el resultado de aplicar un maxpooling (en la parte derecha de la imagen). En dicho ejemplo, se ha utilizado un tamaño de filtro 2x2, con un stride de 2, sobre un volumen de entrada de 4x4, por lo que el volumen de salida es de 2x2.

Esta capa tiene la función de reducir el tamaño del volumen de la matriz de entrada, tratando de mantener las características más importantes. Esto es importante y necesario para que la duración del entrenamiento se reduzca y se realice en un tiempo razonable, ya que realizar una convolución sobre una matriz más grande, es más costoso computacionalmente que si se realiza sobre una matriz más pequeña.

También tiene la función de hacer invariante a la red ante rotaciones, traslaciones, y otras transformaciones geométricas.

- **Capa de deserción (dropout):**

Las capas de deserción, también conocidas como capas *dropout*, tienen una función muy específica, diferente a las de las otras capas. Existe un problema muy común en los entrenamientos de redes neuronales que es el sobreajuste (*overfitting*), donde después del entrenamiento, los pesos de la red están tan ajustados a los ejemplos de entrenamiento que la red no funciona bien cuando se dan nuevos ejemplos.

La idea de esta capa es simple. Esta capa selecciona un conjunto aleatorio de las activaciones de la matriz que se le pasa como entrada y las pone con un valor 0.

¿Qué beneficios trae esto? En cierto modo, obliga a la red a ser redundante. Nos referimos a que la red debería ser capaz de proporcionar la clasificación o salida correcta para un ejemplo específico, incluso si algunas activaciones se abandonan.

Se asegura de que la red no se adapte demasiado a los datos de entrenamiento y, por lo tanto, ayude a aliviar el problema del sobreajuste.

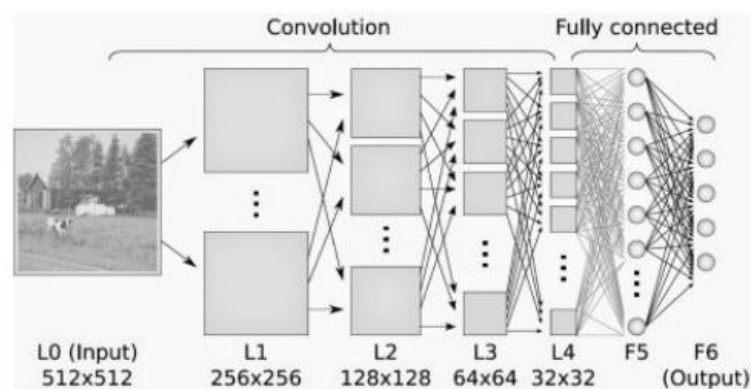
Una anotación importante, es que esta capa solo se utiliza durante el entrenamiento y no durante el tiempo de prueba de la red.

- **Capa FC:**

Esta capa, normalmente, se utiliza al final de la red, tras haber realizado las convoluciones que el programador ha estimado oportunas. Se trata de un perceptrón multicapa. Sus siglas FC provienen de la palabra Fully-Connected, que hace referencia a que cada neurona de una capa tiene una conexión a cada una de las neuronas de la siguiente capa.

A continuación se aclarará cómo es y dónde se sitúa esta capa, mediante la figura 12. Se puede observar la situación que ocupa en una CNN (tras las capas convolucionales, al final del modelo) y vemos la diferencia entre la capa FC, que son simples neuronas como las existentes en las redes neuronales clásicas, y las capas convolucionales, que son filtros, donde cada posición del filtro es el peso de una neurona.

## Fully-Connected (FC) layer



21

Figura 12. Capa FC. Fuente: <https://www.slideshare.net/xavigiro/skin-lesion-detection-from-dermoscopic-images-using-convolutional-neural-networks>

- **Capa softmax:**

La capa *softmax* se suele utilizar en la clasificación de objetos, al final de la CNN, para determinar la probabilidad de que exista un determinado objeto en la imagen. Asigna probabilidades decimales a cada clase en un caso de clases múltiples. Esas probabilidades decimales deben sumar en total 1. Esta capa tendrá un nodo por cada clase que se haya determinado para el problema a resolver. En resumen, la salida de la función softmax es equivalente a una distribución de probabilidad categórica, que te indica la probabilidad de que alguna de las clases esté presente en la imagen.

## 3.2. TENSORFLOW

En este apartado se va a comenzar hablando sobre cómo funciona *TensorFlow* y las particularidades que tiene esta librería. Se contará también los distintos tipos de variables que pueden usarse y algunas técnicas de aumento de datos, técnicas para evitar el sobreajuste y cómo instanciar los distintos tipos de capas que se han visto en el apartado anterior.

Esta biblioteca, desarrollada por Google, es de código abierto desde noviembre de 2015 cuando Google la liberó al público. Originalmente fue desarrollada por Google Brain para llevar a cabo investigaciones de aprendizaje automático y redes neuronales profundas, pero el sistema es lo suficientemente general como para ser aplicable para el desarrollo de otros distintos tipos de programas.

TensorFlow utiliza un grafo de flujo de datos, que será el programa. Para la creación del grafo, hace uso de su unidad elemental, los tensores. Los tensores, en TensorFlow, se pueden representar como una matriz multidimensional de números. Un tensor tiene su rango y forma, donde el rango es el número de dimensiones del tensor y la forma es el tamaño de cada dimensión. Estos tensores admiten una gran variedad de tipos de elementos, como enteros con y sin signo que varían en tamaño desde 8 a 64 bits, float, tipos doubles, un tipo de número complejos y un tipo para strings.

La programación con TensorFlow es algo peculiar debido a que hay que expresar los cálculos numéricos como un grafo dirigido. La figura 13 muestra un ejemplo de grafo, que representa el cálculo de  $h = \text{ReLU}(Wx + b)$ , que es un ejemplo muy típico de redes neuronales, al realizar una combinación lineal de los pesos de las neuronas ( $W$ ), con los datos de entrada ( $x$ ), luego sumarle el bias ( $b$ ) y por último realizar una normalización mediante la función de activación ReLU.

$$h = \text{ReLU}(Wx + b)$$

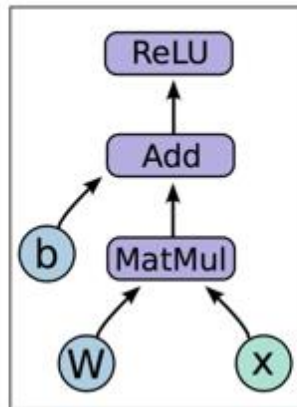


Figura 13. Grafo computacional TensorFlow. Fuente: <https://becominghuman.ai/an-introduction-to-tensorflow-f4f31e3ea1c0>

Ahora vamos a presentar los tres distintos tipos de variables que podemos instanciar en TensorFlow:

- **Constant:** Se utiliza para la creación de un tensor constante, es decir, su valor no puede cambiar durante la ejecución del grafo.
- **Placeholder:** Se utiliza para alimentar los datos de entrada para entrenar el modelo. Los valores de este tipo de variables se introducen en el momento de la ejecución del grafo. Por lo general, se suele utilizar para las entradas de datos, etiquetas o hiperparámetros del modelo.
- **Variable:** Se utilizan para instanciar y actualizar los parámetros. En la mayoría de los modelos de aprendizaje automático, hay muchos parámetros que se deben aprender, que se actualizan durante el entrenamiento.

El modo de trabajar con TensorFlow es, primero, definir el grafo computacional y su estructura, la secuencia que deben seguir los datos, y segundo, ejecutar el grafo una o varias veces.

A continuación vamos a comentar las distintas funciones, existentes en TensorFlow, con las que podemos inicializar los pesos de los parámetros.

- **Random\_normal:** Inicializa dicha variable con un valor aleatorio extraído de una normal, con una desviación típica que se puede especificar como argumento al hacer la llamada a la función.
- **Random\_uniform:** Inicializa la variable con un valor aleatorio entre dos números que se puede indicar como argumento.
- **Inicialización manual:** El programador también tiene la posibilidad de inicializar cada parámetro con el valor que él estime oportuno.

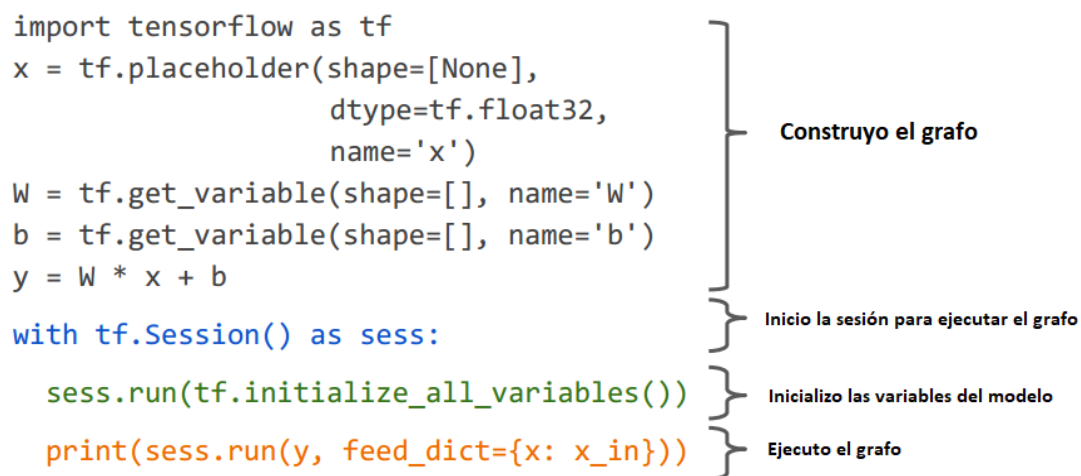


- **Xavier\_initializer:** Esta es la función más utilizada y que más efectividad ha presentado en muchos proyectos. Este inicializador está diseñado para mantener la escala de los degradados aproximadamente igual en todas las capas ([7]).

Estas son las cuatro formas más utilizadas para inicializar los parámetros de un modelo.

Una vez que se haya definido el grafo y se quiera ejecutar, se debe iniciar una sesión, y antes de ejecutarlo es necesario instanciar el valor de todas las variables haciendo una llamada a la función `global_variables_initializer()`.

A continuación se puede observar un ejemplo, que servirá como resumen, donde se muestra los pasos a seguir para trabajar con TensorFlow (figura 14).



```
import tensorflow as tf
x = tf.placeholder(shape=[None],
                    dtype=tf.float32,
                    name='x')
W = tf.get_variable(shape=[], name='W')
b = tf.get_variable(shape=[], name='b')
y = W * x + b
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    print(sess.run(y, feed_dict={x: x_in}))
```

El código se divide en cuatro secciones por corchetes de flujo:

- Construyo el grafo:** Incluye las líneas de importación y la definición de variables y operaciones.
- Inicio la sesión para ejecutar el grafo:** Corresponde a la línea `with tf.Session() as sess:`.
- Inicializo las variables del modelo:** Corresponde a la línea `sess.run(tf.initialize_all_variables())`.
- Ejecuto el grafo:** Corresponde a la línea `print(sess.run(y, feed_dict={x: x_in}))`.

Figura 14. Ejemplo sencillo con TensorFlow. Fuente:  
<http://riosyvalles.com/wp/2016/10/26/small-introduction-to-tensorflow/>

Para evitar el sobreajuste, existen varias funciones útiles que proporciona TensorFlow. Una de esas funciones es `tf.nn.dropout()`, que se trata de la capa de deserción de la que se habló en el apartado anterior. Otras técnicas útiles para evitar el sobreajuste es utilizar una función de optimización del error adecuada, con sus correspondientes parámetros determinados a un valor adecuado. Por ejemplo, existe el optimizador Adam (`tf.train.AdamOptimizer()`), que es de los más utilizados por los desarrolladores, ya que con este optimizador se consigue reducir el error bastante rápido ([8]), pero dependiendo del conjunto de datos, este optimizador puede no ser el mejor ya que tiende a adaptarse demasiado a los datos de entrenamiento y no generaliza tanto como otros optimizadores, que son más lentos en cuanto a reducción del error, como `tf.train.MomentumOptimizer()` o `tf.train.GradientDescentOptimizer()`, pero suelen generalizar más, reduciendo así el sobreajuste de la red ([9]).

Otra técnica para evitar el sobreajuste, muy comúnmente usada por desarrolladores, es la implementación del llamado *weight decay*, que consiste en multiplicar los pesos por

un factor inferior a 1, lo que disminuye los pesos de la red de forma gradual y evita que los pesos crezcan demasiado rápido. Es comúnmente confundido esta función con el parámetro *learning rate*, que es un parámetro cuyo valor tiene que ser determinado por el desarrollador y determina cuánto influye un paso de actualización en el valor actual de los pesos. Se suelen confundir porque ambos afectan a la actualización de pesos mediante la multiplicación de un factor.

Por último, se mostrará cómo pueden instanciarse las capas que se explicaron en el apartado anterior, con TensorFlow.

- **Capa convolucional:**  
`Matriz_salida = tf.nn.conv2d(entrada, filtro, stride, padding)`
- **Capa de activación ReLu:**  
`Matriz_salida = tf.nn.relu(entrada + bias)`
- **Capa de agrupación:**  
`Matriz_salida = tf.nn.max_pool(entrada, tam_filtro, strides, padding)`
- **Capa de deserción:**  
`Matriz_salida = tf.nn.dropout(entrada, prob_desact_activ)`
- **Capa FC:**  
`Matriz_salida = tf.matmul(entrada, pesos) + bias`

### 3.3. CLASIFICACIÓN

Para comenzar este apartado, se definirán los términos clasificación, localización, detección, segmentación y segmentación por instancia, en el ámbito de la visión artificial.

- **Clasificación:** Este tipo de sistemas indica la probabilidad de que haya un determinado objeto en la imagen.
- **Localización:** Además de indicar la probabilidad de que haya un determinado objeto en la imagen, indica donde se sitúa este en la imagen, mediante un rectángulo que lo encapsula.
- **Detección:** Localización en la imagen de entrada de todos los objetos pertenecientes a nuestro problema.
- **Segmentación:** Este sistema se encarga de marcar el contorno de los objetos pertenecientes a nuestro problema.
- **Segmentación por instancia:** Se trata de una segmentación de los objetos presentes en la imagen, en la que se consigue diferenciar correctamente los objetos que pertenezcan a una misma clase y se solapen en la imagen.

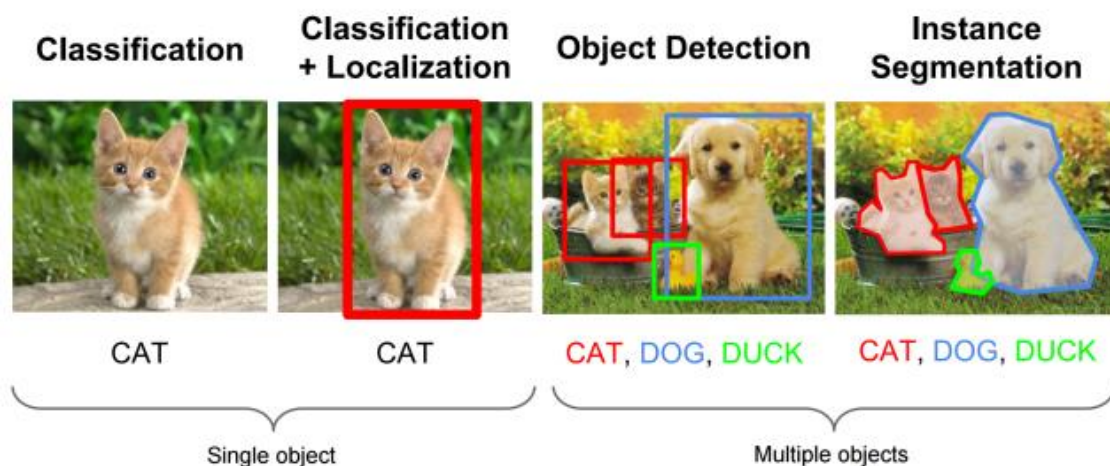


Figura 15. Ejemplo clasificación, localización, detección y segmentación por instancia.

Fuente: [https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/object\\_localization\\_and\\_detection.html](https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/object_localization_and_detection.html)

Este apartado se va a centrar en la clasificación sobre imágenes. Para realizar un sistema de clasificación de imágenes con DL, la técnica más utilizada y que mejores resultados proporciona, es el uso de CNN, por lo que, a continuación, se verá cómo hay que plantear este problema de forma teórica.

Para comenzar, se tienen que definir que clases se quieren clasificar en las imágenes, puede ser un solo objeto, o bien, varios. Por ejemplo, LeCun en 1993 ([5]), diseñó un sistema de clasificación mediante el uso de CNN, el cual clasificaba números escritos a mano, desde el 0 hasta el 9, por tanto su sistema tiene 10 distintas clases. En el caso de este trabajo, a modo de ejemplo, se ha implementado una CNN para la clasificación de personas, por lo que existen únicamente 2 clases, “hay persona” o “no hay persona”.

Una vez se tenga definidas las clases que se quieren clasificar, es necesario crear una base de datos de imágenes donde aparezcan objetos de dichas clases. No hay un máximo ni mínimo de imágenes necesarias, dependerá de la dificultad y variabilidad presente en las imágenes que se vayan a clasificar, aunque lo que sí se ha demostrado, es que cuantas más imágenes se utilicen para el entrenamiento, mejor, ya que así el modelo consigue generalizar aún más ([10]). Se pueden realizar técnicas de aumento de datos, que consisten en realizar transformaciones geométricas a las imágenes que ya se tienen en la base de datos para obtener así aún más imágenes. Las transformaciones geométricas más utilizadas para estos problemas son rotaciones, cambios del brillo y contraste y volteos. Así podemos duplicar o triplicar el conjunto de datos de manera sencilla y efectiva. También se puede optar por hacer una transferencia del aprendizaje, en vez de un entrenamiento desde cero. Un entrenamiento desde cero consiste en crear una CNN y entrenar todos sus pesos desde cero con el nuevo dataset. La transferencia del aprendizaje consiste en tomar un modelo pre-entrenado (los pesos de una red han sido entrenados en un gran conjunto de datos, para realizar clasificación de diferentes

objetos), y afinar ese modelo con el nuevo conjunto de datos. La idea es que este modelo pre-entrenado actúe como extractor de características. Se cambiaría la última o últimas capas de la red para adaptarlo al nuevo problema, se congelan los pesos de las capas anteriores (la congelación de los pesos significa que no se pueden modificar dichos pesos durante el descenso de gradientes) y se entrenan esos pesos nuevos. Este tipo de entrenamiento funciona debido a que las primeras capas de cualquier red extraen características comunes a cualquier objeto, como curvas o bordes, que siguen siendo válidas también para el nuevo conjunto de datos, así que solo nos enfocamos en las últimas capas que se encargarán de extraer características globales de los objetos. La transferencia del aprendizaje también tiene la ventaja de que no es necesario un conjunto de datos de entrenamiento tan grande como el que sería necesario para entrenar el modelo desde cero.

Cuando se tenga la base de datos de imágenes, habrá que diferenciar dos conjuntos de datos, de entrenamiento y de test, el primero de ellos se utilizará para entrenar el modelo, y el segundo de ellos para comprobar la eficacia del modelo con imágenes con las que no se ha entrenado. Tras tener esto, se comenzará a crear el modelo. Se implementará un modelo más o menos complejo, es decir, con más o menos capas convolucionales, según la cantidad y el tipo de objetos que van a clasificarse. No hay un estándar establecido que indique cuántos números de capas utilizar, pero si existen evidencias de que si un problema es sencillo de resolver y ponemos muchas capas convolucionales en nuestra CNN, el modelo se sobreajustará con mucha seguridad al conjunto de entrenamiento y no generalizará bien. Para crear sistemas de clasificación con CNNs, lo más común es la siguiente secuencia de capas. Se utilizan secuencialmente capas de convolución siempre seguidas de una capa ReLu. Además se suelen incluir en la estructura unas capas de pooling, para disminuir el tamaño del mapa de características, para que el entreno no se demore mucho en el tiempo y el sistema sea invariante a rotaciones y escalas. Por último, se utilizan comúnmente una o dos capas FC y seguidamente una capa softmax que determinará que objeto hay presente en la imagen (ejemplo en figura 16).

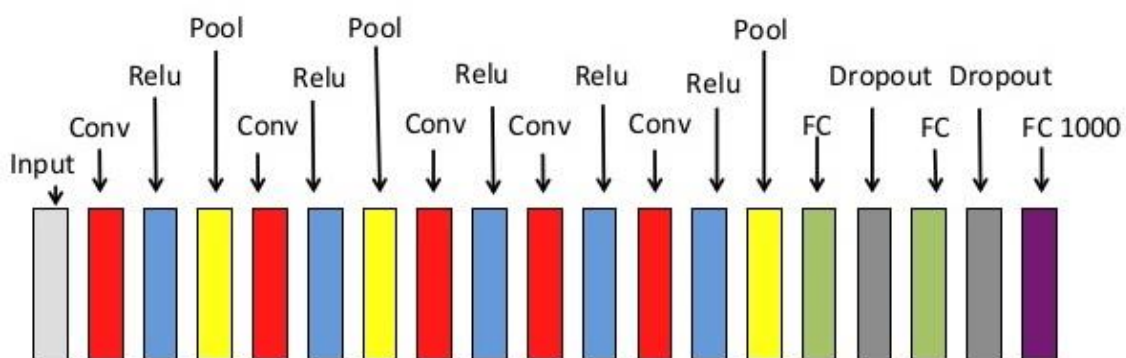


Figura 16. Ejemplo CNN. Fuente: <https://www.slideshare.net/MarkScully5/intro-to-convolutional-neural-networks>

El número de capas convolucionales que se utilicen y los hiperparámetros que tiene cada una de las capas serán definidos por el programador. Para ello deberá de realizar un estudio del problema que está tratando resolver y buscar problemas similares que se hayan resuelto también mediante CNN y determinar así dichos valores.

Una vez se tenga el modelo implementado, es necesario que el programador determine cada uno de los hiperparámetros que hay en cada capa de la red neuronal. Se van a explicar a continuación dichos hiperparámetros, que ya introdujimos en el apartado 3.1.1.

- **Capa Convolucional:**

- **Tamaño y Número de filtros en cada capa convolucional.** Normalmente, los programadores en las primeras capas utilizan un menor número de filtros y conforme se va profundizando en la red, utilizan cada vez más filtros en las convoluciones. Esto es debido a que en las primeras capas, los filtros aprenden a encontrar características sencillas, como bordes, y para ello no es necesario mucha cantidad de filtros; sin embargo, en las últimas capas, los filtros deberían aprender características de alto nivel y para ello es necesario el uso de un mayor número de filtros. En cuanto al tamaño, ocurre lo contrario, se suelen utilizar filtros más grandes en las primeras capas, debido a que la matriz de entrada es mayor al principio y esta va disminuyendo conforme va avanzando por la red, debido a la aplicación de capas de pooling.
- **Stride de los filtros.** Salto que tienen que dar los filtros al hacer una convolución en una capa. Este valor se determina en función del tamaño de la matriz de entrada, el tamaño del filtro y el padding utilizado, para que el filtro se desplace por toda la matriz de entrada sin perder ninguna característica de este.
- **Tamaño del Padding para las convoluciones.** Este valor se suele determinar en función del tamaño del filtro escogido, para que los bordes de la matriz de entrada también sean tratados por las convoluciones.

- **Capa Pooling:**

- **Tamaño de la ventana del pooling.** El tamaño de la ventana del pooling se suele determinar junto con su stride, en función del deseado tamaño de la matriz de salida.
- **Stride de la ventana de pooling.**

- **Capa FC:**

- **Número de neuronas de la capa FC.** Se utilizarán un mayor número de neuronas cuanto mayor sea la complejidad del problema a resolver. La última

capa FC debe tener cómo número de neuronas la cantidad de clases que queramos clasificar.

- **Capa Dropout:**

- **Cantidad de activaciones de la matriz de entrada a desactivar.** Un valor de 1 en esta capa, significa que la capa no hace nada sobre la matriz de entrada. Un valor de 0 significa que se van a desactivar todas las activaciones de la matriz de entrada. Usualmente, la mayoría de los programadores utilizan un valor de 0.5 para este parámetro, que indicaría que se van a desactivar el 50% de las activaciones de la matriz de entrada.

Para que el modelo pueda entrenarse para la clasificación, es necesario que los datos estén etiquetados. En el caso de estar trabajando con imágenes y queremos que el clasificador indique qué objeto hay presente en la imagen, es necesario etiquetar las imágenes de entrenamiento, indicando qué objeto hay presente en cada imagen. Así, cuando se entrene, se calculará el error que ha tenido la red al clasificar cada imagen en función de la etiqueta que ha devuelto la red y la etiqueta real. Mediante un optimizador, se ajustarán los pesos para que la red se adapte de la mejor forma posible para que devuelva las etiquetas reales del conjunto de datos de entrenamiento.

Mientras se va entrenando la red, es conveniente ir reflejando en un gráfico dos métricas, el error de la red y la precisión, tanto para el conjunto de datos de entrenamiento, como para el conjunto de datos de test. De esta forma, se pueden sacar conclusiones de cómo evoluciona el proceso o si se produce o no sobreaprendizaje.

La precisión de una red viene dada en función de su nivel de acierto para un conjunto de imágenes. Es decir, si por ejemplo se tienen 100 imágenes de perros, y en 60 de ellas el clasificador acierta y en 40 de ellas indica que son gatos, entonces dicho clasificador tiene un 60% de acierto sobre dicho conjunto de datos. Se considera que el clasificador ha acertado en la clasificación de una imagen si la mayor probabilidad se la asigna a la etiqueta correcta.

El error se calcula en función de la distancia que hay entre la etiqueta proporcionada por la red, con respecto a la etiqueta deseada. Por ejemplo, en un problema de clasificación de imágenes con personas, si se tiene un conjunto de 50 imágenes de personas y el clasificador indica que en cada una de ellas hay un 80% de probabilidad de que haya personas, entonces dicho clasificador tiene un 100% de acierto para ese conjunto de datos, pero tiene un 20% de error en cada imagen, ya que lo deseado sería que lo indicara con un 100% de probabilidad. El error que comete el clasificador para cada imagen se calcula mediante un valor comprendido entre 0 y 1. El error será 0 cuando devuelva con una probabilidad del 100% la etiqueta real y tendrá un valor de 1 cuando devuelva con una probabilidad del 0% la etiqueta real.

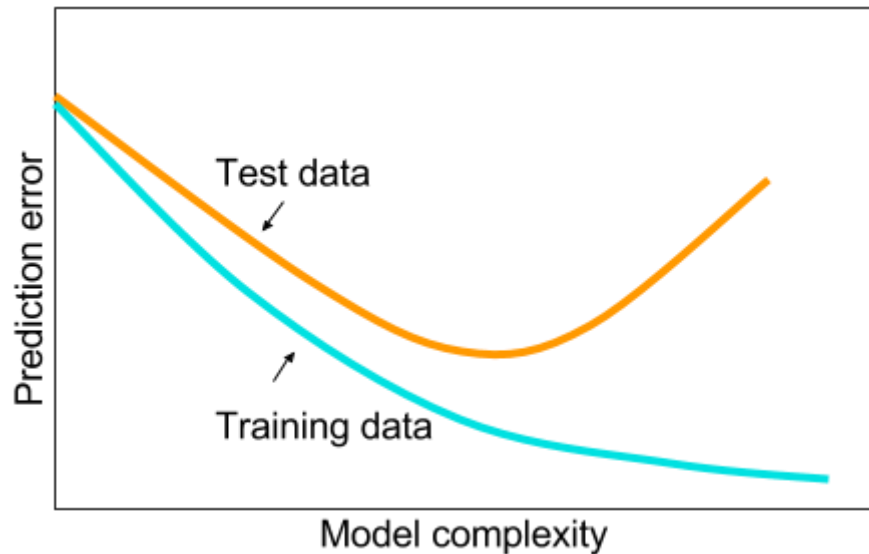


Figura 17. Ejemplo de sobreaprendizaje detectado mediante el análisis del error del modelo. Fuente: [https://gluon.mxnet.io/chapter02\\_supervised-learning/regularization-scratch.html](https://gluon.mxnet.io/chapter02_supervised-learning/regularization-scratch.html)

En la figura 16 se puede apreciar un claro ejemplo de sobreaprendizaje. Dicha gráfica, representa el error durante el entrenamiento de un modelo, para el conjunto de datos de entrenamiento y para el de test. Se puede apreciar al inicio, que el error en ambos conjuntos va disminuyendo, pero se llega a un punto en la gráfica, en el que el error del conjunto de test empieza a ascender, mientras el error del conjunto de entrenamiento sigue disminuyendo. En un ejemplo donde no se produjera sobreaprendizaje, el error del conjunto de test no empezaría a aumentar, sino que seguiría disminuyendo a la par con el error del conjunto de entrenamiento.

### 3.4. DETECCIÓN

En este apartado se van a explicar los conceptos teóricos necesarios para llevar a cabo la implementación de un sistema de detección de objetos en imágenes. La detección de los objetos presentes en una imagen consiste en delimitar cada uno de los objetos para los que el sistema se ha entrenado, mediante un rectángulo al que llamaremos bounding box.

Se va a comenzar citando las distintas ideas que han surgido en los últimos años sobre la implementación de sistemas de detección mediante CNN; continuaremos explicando más en profundidad la idea del sistema Faster R-CNN, que se ha seguido para realizar el detector de personas en imágenes debido a su rapidez en la detección y su precisión con respecto a otras opciones.



Estas son las tres ideas que se han propuesto hasta el día de hoy sobre detección de objetos mediante CNN:

- **R-CNN ([11]):**

Se basa en la búsqueda selectiva, que consiste en generar un número determinado de regiones, por ejemplo 2000, que tienen la mayor probabilidad de contener un objeto. Después de haber elaborado un conjunto de propuestas regionales, estas propuestas se transforman en un tamaño de imagen que puede ser introducido como entrada a una CNN, que extrae un mapa de características para cada región. Este mapa de características se utiliza entonces como entrada de un conjunto de SVM lineales que se entrenan para cada clase y producen una clasificación. De esta forma, se determina a que clase pertenece cada una de las regiones propuestas. El problema de este método es la ineficiencia, ya que es necesario pasar cada una de las regiones propuestas a una CNN y esto ralentiza mucho el sistema. Como se verá a continuación, han surgido mejoras sobre esta idea que han mejorado la eficiencia del sistema.

- **Fast R-CNN ([12]):**

Como se ha comentado, se hicieron mejoras al modelo R-CNN original debido a tres problemas. El entrenamiento tomó varias etapas, tiene bastante carga computacional y es demasiado lento para resolver problemas en tiempo real.

Fast R-CNN fue capaz de resolver el problema de la velocidad compartiendo básicamente el cómputo de las capas convolucionales entre diferentes propuestas e intercambiando el orden de generar propuestas de región y ejecutar la CNN. En este modelo, la imagen se introduce primero en una CNN, las características regionales se obtienen a partir del último mapa de características y por último se tienen dos capas FC, una para determinar la probabilidad de que haya un objeto en la propuesta de región y otra para hacer una regresión del bounding box propuesto, para adaptarlo mejor al objeto.

- **Faster R-CNN ([13]):**

Esta idea surgió para combatir el entrenamiento algo complejo que tanto R-CNN como Fast R-CNN exhibieron. Los autores insertan una Red de Propuesta de Región (RPN) después de la última capa convolucional. Esta red es capaz de mirar el último mapa de características y producir propuestas de regiones a partir de ello. A partir de esa etapa, se utiliza la misma tubería que el R-CNN (ROI pooling (RoIP) y 2 capas FC, para clasificación y regresión del bounding box). El Faster R-CNN se ha convertido hoy en día en el estándar para los sistemas de detección de objetos.



### 3.4.1. DETECCIÓN CON FASTER R-CNN

Para la implementación del sistema de detección, nos hemos basado en la idea del Faster R-CNN. En nuestro caso no es necesario implementar todas las capas, debido a que en se trabaja con una sola clase (la clase “persona”), de esta forma, la parte del sistema RoIP y las dos capas FC, que se encargan de determinar a qué clase pertenecen cada una de las regiones propuestas, no han sido necesario implementarlas, ya que en nuestro sistema las regiones propuestas solo tienen la posibilidad de contener únicamente personas. Por ello, tan solo se ha tenido que implementar la capa RPN para el desarrollo del detector de personas en imágenes.

A continuación se va a mostrar un esquema que resume las etapas que tiene el método Faster R-CNN, para problemas con solo una clase (figura 17).

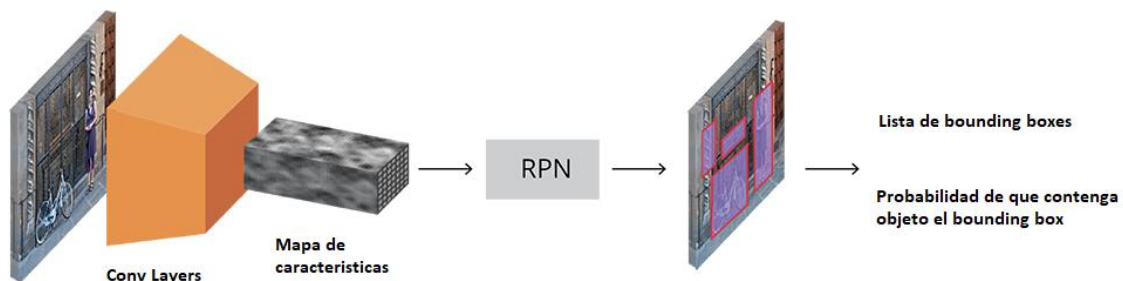


Figura 18. Esquema Faster R-CNN. Fuente:

<https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>

Al comienzo, se tienen varias capas convolucionales, como en la clasificación. Esto se hace para obtener un mapa de características, el cual se pasará a la capa RPN, que ahora se explicará su funcionamiento con detalle, y esta devuelve una lista de cuadros delimitadores (bounding boxes), y una probabilidad de que contenga o no un objeto, para cada uno de ellos.

La capa RPN tiene como misión principal resolver un problema de longitud variable (no se saben cuántos objetos puede haber en una imagen, ni que forma tienen), para ello utiliza anclas, cuadros delimitadores de referencia, de tamaño fijo que se colocan uniformemente en toda la imagen original. En lugar de tener que detectar dónde están los objetos, se modela el problema en dos partes. Para cada ancla, se pregunta:

- ¿Este ancla contiene un objeto?
- ¿Cómo se ajustaría este ancla para que se ajuste mejor al objeto?

Ya que se tiene un mapa de características de un determinado ancho (A), alto (Al) y profundidad (Pr), como entrada en la RPN, se crea un conjunto de anclas para cada uno

de los puntos en  $A \times A$ . Es importante comprender que, aunque los anclajes se definen según el mapa de características, los anclajes finales hacen referencia a la imagen original. Como solo se tienen capas convolucionales y de pooling antes de la RPN, las dimensiones del mapa de característica serán proporcionales a las de la imagen original. Si se define un anclaje por cada posición espacial del mapa de características, la imagen final terminará con un grupo de anclas separadas por 'r' píxeles:

Tamaño imagen:  $W * H$

Tamaño Mapa características:  $\left(\left(\frac{W}{r}\right) * \left(\frac{H}{r}\right)\right)$

$$r = \frac{\text{tamaño imagen}}{\text{tamaño mapa características}}$$

Eq. 3

dónde:

- W: Ancho de la imagen de entrada.
- H: Alto de la imagen de entrada.
- r: Separación en píxeles entre el centro de cada ancla.

A modo de ejemplo visual, en la figura 19 se muestra como quedarían distribuidos los puntos de anclaje de una imagen. Dichos puntos, han sido calculados como se ha explicado anteriormente. Entre cada uno de dichos puntos existe una separación de 'r' píxeles, que se ha calculado mediante las fórmulas que se han indicado arriba.

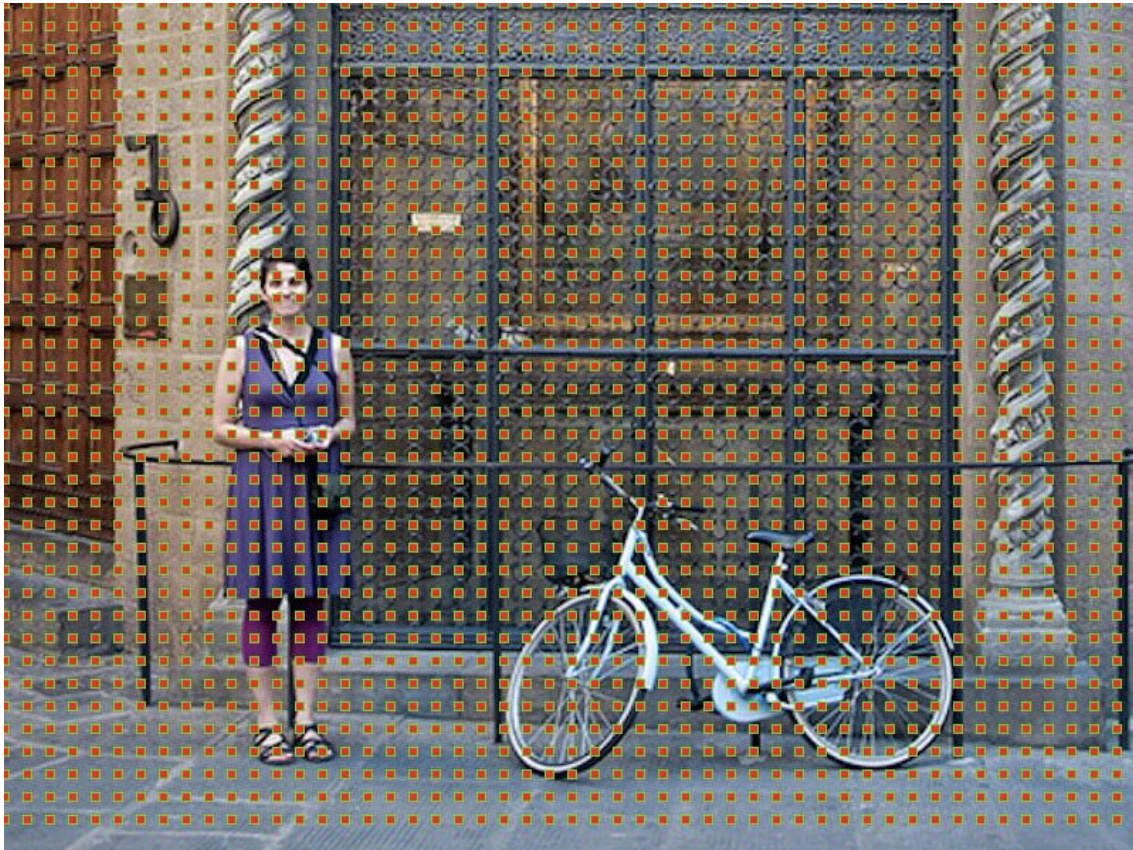


Figura 19. Ejemplo anclas RPN. Fuente: <https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>

Sobre cada uno esos puntos de anclaje se definirán ‘k’ anclas, de distintos tamaños y relaciones de aspecto. Por ejemplo, se podrían tener 9 anclas, con tres tamaños diferentes (64px, 128px, 256px) y tres relaciones de aspectos diferentes (0.5, 1, 1.5), de esta forma, usando las combinaciones posibles entre ellos, se tendría 9 anclas. El tamaño escogido para las anclas dependerá del tamaño de las imágenes a tratar. El RPN coge todos los cuadros de referencia (anclas) y genera un conjunto de buenas propuestas para los objetos. Lo hace teniendo dos salidas diferentes para cada una de las anclas:

- Probabilidad de que dicho ancla contenga un objeto.
- Regresión del cuadro delimitador para ajustar los anclajes mejor al objeto que está marcando.

RPN se implementa eficientemente de una manera totalmente convolucional, utilizando el mapa de características devuelto por la red base como entrada. Primero se utiliza una capa convolucional que se aplicará al mapa de características, y luego se tienen dos capas convolucionales paralelas, usando un filtro de tamaño 1x1 para cada una de ellas, cuyo número de canales dependerá del número de anclas que se tengan por anclaje.

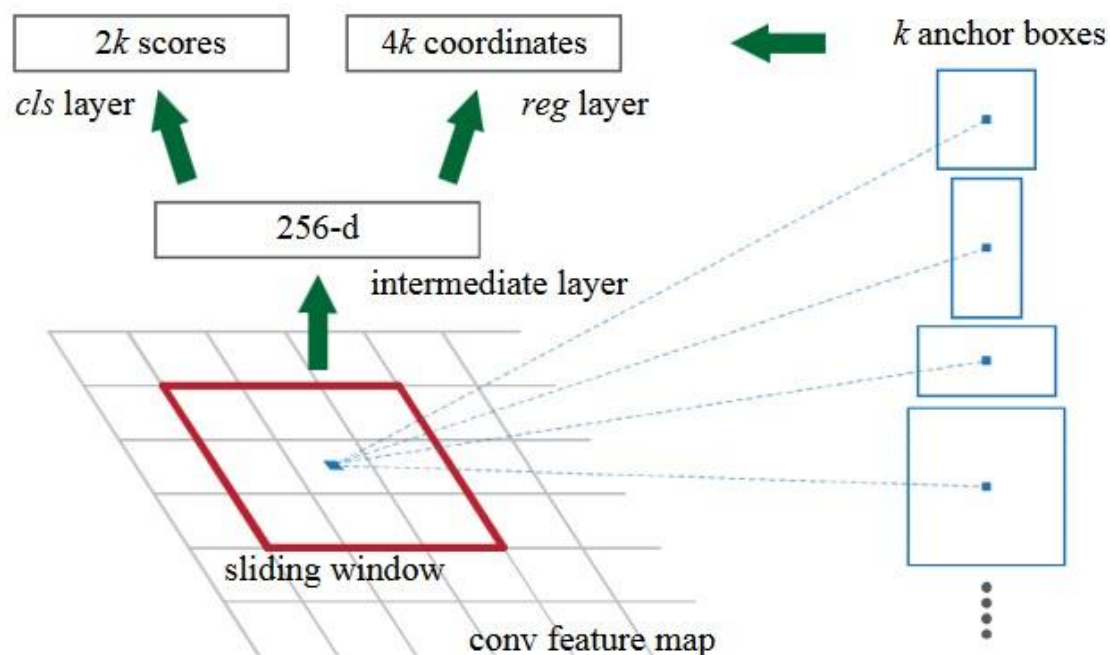


Figura 20. Esquema RPN. Fuente: <https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>

Atendiendo al diseño mostrado en la figura 20, una de las capa convolucionales paralelas tendrá  $2 \cdot k$  canales, y la otra  $4 \cdot k$  canales. La de  $2 \cdot k$  canales es la que se encargará de asignar la probabilidad de que haya objeto o no dentro del ancla, por eso tiene dicho tamaño, por cada ancla (se tienen ' $k$ ' anclas), determinará dos probabilidades (la de que haya objeto y la de que no haya objeto). La capa de  $4 \cdot k$  canales, se encargará de la regresión del cuadro delimitador para adaptar mejor el ancla al objeto que está marcando, por ello tiene ese tamaño de  $4 \cdot k$ , porque cada ancla (se tienen ' $k$ ' anclas) viene definido por 4 puntos (X, Y, Ancho, Alto).

Para el entrenamiento de la RPN, se toman todos los anclajes y se clasifican en dos categorías diferentes. Aquellos que se superponen con un objeto de verdad con una Intersección sobre Union (IoU) mayor que 0.7 se considera un ancla positiva y aquellos que tienen un IoU menor que 0.3 con respecto a cualquier objeto real presente en la imagen, serán consideradas como anclas negativas. Luego se seleccionan aleatoriamente un conjunto de esos anclajes positivos y negativos, de forma que se tengan el mismo número de muestras positivas que negativas.

En el caso de que no haya ningún anclaje con un IoU mayor que 0.7, es decir, no habría anclajes positivos para dicha imagen, lo que hacemos es asignar como anclaje positivo aquel que tenga mayor IoU con respecto a algún bounding box real, así siempre se tiene al menos 1 muestra positiva para el entrenamiento por cada imagen.

La Intersección sobre Unión consiste en un valor comprendido entre 0 y 1, que indica el grado con el que se superponen dos rectángulos (cuadros delimitadores). De esta forma, se puede medir lo bien o mal que se ajusta un bounding box predicho con respecto a un bounding box real.

Para el etiquetado de las imágenes de entrenamiento, donde se ha tenido que marcar mediante un cuadro delimitador a todas las personas que apareciesen en dichas imágenes, se ha utilizado un software de libre distribución que ha facilitado dicha tarea ([14]). De esta forma, se sabe dónde se encuentra el objeto real en la imagen para el entrenamiento y así se puede etiquetar las anclas como positivas o negativas.

En [15] donde los autores de Faster R-CNN explican su funcionamiento, recomiendan para el entrenamiento de la regresión de los cuadros delimitadores usar la pérdida Smooth L1. Para el entrenamiento de la regresión proponen la siguiente parametrización, que hay que realizarla para calcular la pérdida Smooth L1, con dichas parametrizaciones:

$$\begin{aligned} t_x &= (x - x_a)/w_a, & t_y &= (y - y_a)/h_a, \\ t_w &= \log(w/w_a), & t_h &= \log(h/h_a), \\ t_x^* &= (x^* - x_a)/w_a, & t_y^* &= (y^* - y_a)/h_a, \\ t_w^* &= \log(w^*/w_a), & t_h^* &= \log(h^*/h_a), \end{aligned}$$

Figura 21. Parametrizaciones Faster R-CNN. Fuente:  
<https://arxiv.org/pdf/1506.01497.pdf>

En las ecuaciones mostradas en la figura 21,  $x$ ,  $y$ ,  $w$  y  $h$  denotan las coordenadas centrales del bounding box y su altura y anchura. Las variables  $x$ ,  $x_a$ ,  $x^*$  son para el bounding box pronosticado, el bounding box del anclaje y el bounding box verdadero, respectivamente (igual para  $y$ ;  $w$ ;  $h$ ).

En [15], se resalta que para el cálculo de la pérdida de la regresión de los anclajes solo se utilizarán los anclajes con etiquetas positivas.

Para el cálculo de la pérdida para la parte de la red que se encarga de determinar la probabilidad de que en cada anclaje haya o no un objeto, en el documento de los autores se recomienda utilizar la pérdida logarítmica. En este caso, a diferencia que para el cálculo de la pérdida de la regresión, se utilizan tanto las muestras positivas como las negativas que se han elegido como batch.

La pérdida total, que va a ser la que se va a optimizar mediante alguna función de las que proporciona TensorFlow para optimizar las pérdidas, vendrá calculada por la suma



de la pérdida de la parte de regresión y de la pérdida de la parte de clasificación de cada anclaje. En [15] se recomienda utilizar la función de descenso por gradiente estocástico, con el parámetro ‘momentum’ a un valor de 0.9, con el learning rate inicializado a 0.001 y disminuyendo progresivamente con el paso de las iteraciones hasta llegar a un valor de 0.0001.

$$Pérdida\ Total = Pérdida_{CLS} + Pérdida_{REG}$$

Eq. 4

A la parte de clasificación se hace referencia mediante las siglas CLS y a la parte de la regresión mediante las siglas REG.

Para el entrenamiento de este modelo también se puede realizar la transferencia del aprendizaje que fue explicado en el apartado de clasificación. Las primeras capas, que son convolucionales, con las que se obtiene el mapa de características, pueden estar ya entrenadas y no sería necesario volver a entrenar dichas capas de nuevo, sino que se congelan esos pesos y se entrena solo el resto de las capas del modelo. En este trabajo fin de grado, a modo de ejemplo, se ha realizado primero un clasificador de imágenes con personas, que tenía varias capas convolucionales, por lo que se ha mantenido esas capas convolucionales y se han congelado sus pesos para así obtener el mapa de características para el entrenamiento de la detección. También se podría haber utilizado dichas capas sin congelar sus pesos, de forma que se puedan seguir moldeando para el entreno del detector.

Otra forma de entrenarlo es desde cero, con todos los pesos de todas las capas inicializados mediante algún inicializador de pesos de los que se comentaron en el apartado 3.2.

Para evaluar el rendimiento del detector se utiliza una métrica denominada mAp. De forma muy resumida esta métrica se puede calcular, teniendo sólo una clase como es nuestro caso, calculando el promedio de aciertos y fallos de los bounding boxes predichos con respecto a los bounding boxes reales. Para ello, se considerará un acierto aquellos bounding boxes con un IoU mayor o igual a 0.5 con respecto a algún bounding box real. Se considerará un fallo si tiene un IoU menor que 0.5 con respecto a cualquier bounding box real. Luego, también recorreremos cada uno de los bounding boxes reales, comprobando si hay algún bounding box predicho que lo cubra. En caso afirmativo, se contará como un acierto más, y, en caso contrario, se contará como fallo. Por último se calcula el promedio de acierto y fallos para obtener el valor de rendimiento mAp.

Por último, se va a explicar la supresión no máxima. Dado que los anclajes generalmente se superponen, las propuestas también. Para resolver esto se utiliza el

algoritmo de supresión no máxima (NMS). NMS toma la lista de propuestas ordenadas por puntuación e itera sobre ella, descartando aquellas propuestas que tienen un IoU mayor que un umbral predefinido con una propuesta que tiene una puntuación más alta. El umbral IoU que se defina es importante, ya que si se establece demasiado bajo puede terminar perdiendo propuestas de objetos y si se establece demasiado alto podría terminar con demasiadas propuestas un mismo objeto. El valor más utilizado por los desarrolladores es 0.6.

## **4. EXPERIMENTACIÓN**

### **4.1. INTRODUCCIÓN**

En este capítulo, se comentará cómo se ha elaborado tanto el sistema de clasificación como el detector de personas en imágenes, cada una de las etapas por las que se ha pasado y se explicará mediante esquemas los dos modelos desarrollados, así como el valor de los hiperparámetros escogidos para cada capa de los modelos.

A su vez, se mostrarán los resultados obtenidos tras el entrenamiento, mediante gráficas en las que se muestran el error y la precisión obtenida, tanto para el conjunto de imágenes de entrenamiento como para el de test, y se visualizarán pruebas que se han realizado sobre imágenes reales para mostrar como trabaja el detector; de esta forma se podrá comprobar la eficacia de ambos sistemas.

### **4.2. CLASIFICACIÓN**

#### **4.2.1. BASE DE DATOS**

Se han seleccionado 2106 imágenes que contienen personas, y otras 2106 imágenes que no contienen personas y ese ha sido nuestro conjunto de datos de entrenamiento. Para el conjunto de datos de test, se han seleccionado 100 imágenes que contienen personas y otras 100 imágenes que no contienen personas. Las imágenes seleccionadas se han

escogido al azar, realizando búsquedas en Google y no se ha cogido una base de datos de imágenes que hubiese en internet preparada para el entrenamiento, debido a que no se ha encontrado ninguna base de datos que contuviera una cantidad de imágenes lo suficientemente grande para el entrenamiento del modelo.

El etiquetado de la base de datos se ha hecho de forma que se han metido las imágenes de personas en una carpeta y las imágenes de fondo en otra y se ha ido nombrando cada imagen con un número, para que luego en el programa, al generar el batch para cada iteración del entrenamiento, se supiera a que clase pertenece cada imagen. El batch, no es más que un conjunto de imágenes de entrenamiento que contiene imágenes tanto de la clase “hay persona” como de la clase “no hay persona” y va a ser utilizado en cada iteración del entrenamiento para alimentar la red. En cada iteración, el batch estará formado por distintas imágenes. En nuestro caso utilizamos un batch con 14 imágenes de clases escogidas aleatoriamente, con la misma probabilidad de escoger una imagen de una clase que de otra.

## 4.2.2. ARQUITECTURA

En este apartado, se van a ver los distintos hiperparámetros que se han seleccionado para la implementación de nuestro modelo de clasificación y a su vez, se verá mediante la figura 22, cómo es la estructura de capas de nuestro modelo.

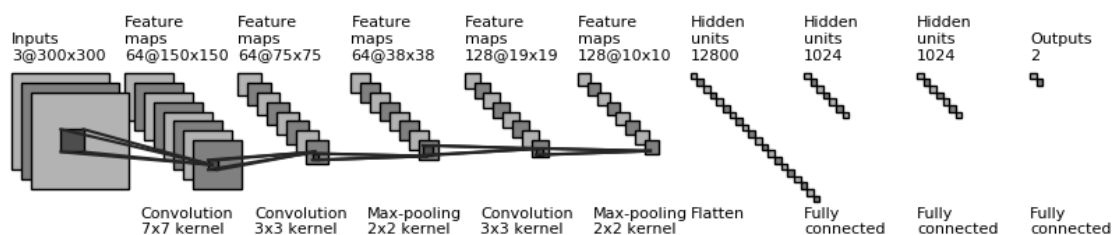


Figura 22. Modelo de clasificación. Fuente: Elaboración propia

La figura 22 representa la estructura de capas de nuestro modelo de clasificación. Se puede ver que las imágenes de entrada a la red tienen un tamaño de 300x300x3, es decir, utilizamos imágenes en RGB.

Luego, se tiene una capa convolucional, con un stride de 2, por lo que la salida de dicha capa son unas matrices, cuyos tamaños son la mitad del tamaño de la matriz de entrada, que en ese caso, entra tres matrices de 300x300 (la imagen de entrada) y, tras aplicarle



los filtros de  $7 \times 7 \times 3 \times 64$  (64 filtros de  $7 \times 7 \times 3$ ), obtenemos como salida 64 matrices de  $150 \times 150$ .

A esas matrices obtenidas tras la primera convolución, se le aplica de nuevo otra capa convolucional, con filtros de  $3 \times 3 \times 64 \times 64$  y utilizando un stride de 2, por lo que la salida de esta capa es una matriz de  $75 \times 75 \times 64$ .

Tras esto, se va a utilizar una capa max-pooling, para reducir el tamaño del mapa de características. Se utilizará un filtro de tamaño  $2 \times 2$ , con un stride de 2, por lo tanto, el resultado es que la matriz de salida tiene un tamaño de  $38 \times 38 \times 64$ .

A continuación se aplica a dicha matriz de salida otra capa convolucional, con filtros de tamaño  $3 \times 3 \times 64 \times 128$  y con un stride de 2. El resultado, por tanto, es una matriz de tamaño  $19 \times 19 \times 128$ .

Ahora se aplica una capa de max-pooling, con un filtro de  $2 \times 2$  y un stride de 2, consiguiendo así reducir el mapa de características de entrada, de tamaño  $19 \times 19 \times 128$ , en un mapa de características de tamaño  $10 \times 10 \times 128$ .

Este último mapa de características se introduce por la primera capa FC que existe en nuestro modelo. Esta capa, tendrá como número de neuronas de entrada  $10 \times 10 \times 128$  neuronas, es decir, 12800 neuronas, que estarán unidas a 1024 neuronas. Esas conexiones son los pesos de esta capa, por lo que existen  $12800 \times 1024$  pesos que ajustar en esta capa.

Luego, existe otra capa FC, que parte de las 1024 neuronas de la anterior capa y se une a otras 1024 nuevas neuronas. Por lo tanto en esta capa existen  $1024 \times 1024$  pesos que deberán ajustarse.

Por último, se tiene la última capa FC del modelo, que une las 1024 neuronas de la capa anterior a 2 nuevas neuronas. Se tienen 2 neuronas en esta capa porque la red solo tiene que clasificar las imágenes en 2 clases, hay persona, o bien, no hay persona. Es decir, en la última capa FC se debe poner como número de neuronas, el número de clases que se quieran clasificar. Esta última capa FC alimentará a una capa softmax, que indicará la probabilidad de que la imagen de entrada al modelo pertenezca a una clase u otra.

Se puede ver como todo el modelo se ejecuta de forma secuencial, es decir, no hay ninguna capa en paralelo, cosa que sí se podrá ver en el modelo de detección para diferenciar las capas que se encargan de la clasificación de las anclas, de las capas que se encargan de hacer la regresión de las anclas para ajustarlas mejor a la persona que está marcando.

Nos hemos quedado con esta estructura y no con otra, porque se ha considerado que era la que mejor resultados arrojaba en las gráficas de error y precisión. Se empezó con una red más sencilla que la que se ha presentado y se comprobó que el modelo no tenía suficientes capas convolucionales para poder clasificar con buena precisión el conjunto de entrenamiento. Por ello, se fueron incorporando capas convolucionales hasta

conseguir una precisión decente para el conjunto de imágenes de entrenamiento. Una vez llegado a ese punto, se comprobó como se comportaba la red para el conjunto de test, en cuanto al error obtenido, y observamos que se producía sobreaprendizaje de forma severa. Para aliviar este problema, se han utilizado capas dropout entre las capas FC del modelo, lo que redujo bastante el sobreaprendizaje, pero aún así seguía existiendo. Se implementó la técnica del weight decay, pero el resultado era muy parecido al anterior. Así que nos quedamos con el modelo con dropout que es el que se ha representado en la figura 27. Se partió de una red pequeña y se le ha ido haciendo más compleja, añadiéndole más capas convolucionales, porque esa es la forma más fácil de obtener un modelo que generalice mejor y se disminuyen las probabilidades de que se produzca sobreaprendizaje, ya que si se tiene un modelo pequeño, se tienen menos pesos que ajustar, lo que hace que el modelo no se ajuste demasiado al conjunto de entrenamiento porque no tiene dicha capacidad al no disponer de tantos pesos.

### **4.2.3. ENTRENAMIENTO Y RESULTADOS**

En cada iteración se van ir cogiendo 14 imágenes y se va a ir formando un batch con estas y sus etiquetas. Para crear este batch, se va seleccionando imagen a imagen de forma aleatoria, con la misma probabilidad de seleccionar una imagen de una clase que de la otra, de forma que el batch puede contener más imágenes de una clase que de otra. Se tendrá un error en la red que se calcula comparando la etiqueta real de la imagen con la etiqueta que devuelve el modelo. Las imágenes positivas se etiquetarán con el valor [1, 0], indicando que la probabilidad de que contenga una persona es de un 100%, mientras que se etiquetará con un valor [0, 1] las imágenes negativas.

Se irán realizando iteraciones, calculando el error que se va cometiendo y mediante un algoritmo de optimización del error, se van ajustando los pesos del modelo, cometiendo así cada vez un error menor para el conjunto de imágenes de entrenamiento.

Se va ahora a visualizar y comentar la figura 23, en la que se representa el error obtenido durante el entrenamiento del clasificador. Podréis observar el error cometido durante el entrenamiento para el conjunto de imágenes de entrenamiento, en color azul, y para el conjunto de imágenes de test, en color rojo.

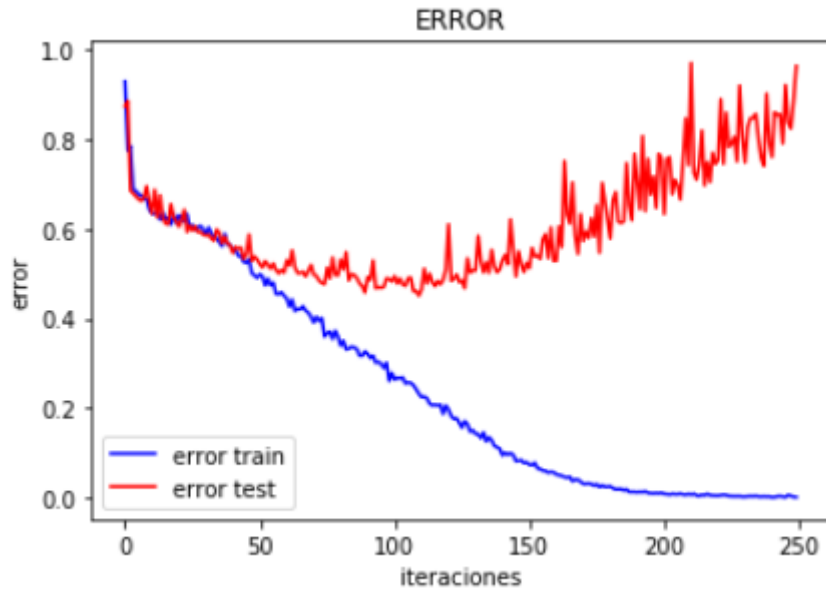


Figura 23. Error clasificación. Fuente: Elaboración propia

Se puede apreciar cómo el error del entrenamiento va disminuyendo, desde que empieza hasta que acaba de forma continuada. Al comienzo, el error va disminuyendo rápidamente con el paso de las iteraciones, y al final, cuando se va acercando a un valor de error 0, el error se queda estancado y no se consigue disminuir más con el paso de las iteraciones. En cuanto al error del test, se puede apreciar que al inicio, se va disminuyendo, pero a partir de la iteración 100, el error de train sigue bajando, pero el error de test comienza a ascender. Esto es un claro ejemplo de sobreaprendizaje, en el que el sistema ha aprendido muy bien clasificar los datos de entrenamiento, pero cuando el sistema se prueba con otro conjunto de datos, como el de test, no obtiene buenos resultados.

A continuación se va a mostrar la figura 24, en la que se representa la precisión del sistema de clasificación para el conjunto de entrenamiento y el de test.

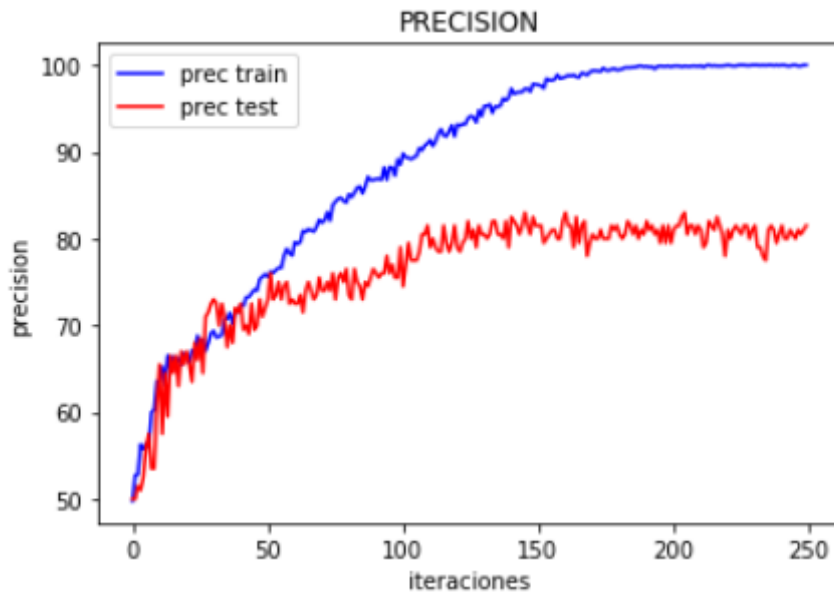


Figura 24. Precisión clasificación. Fuente: Elaboración propia

Podéis observar un comportamiento muy parecido al representado en la figura del error (figura 23). Se aprecia como al inicio, hasta la iteración 50, tanto la precisión del entreno como del test van mejorando a la par, pero a partir de dicha iteración, la precisión del entreno sigue aumentando con el paso de la iteraciones, pero la precisión del test se comienza a estancar, ascendiendo un poco más hasta la iteración 100, cuando se queda totalmente estancado, con una precisión en torno al 80%.

#### 4.2.4. ANÁLISIS

El sobreprendizaje que se produce es debido a las imágenes seleccionadas para el conjunto de datos de entrenamiento. El problema viene porque las imágenes seleccionadas se han escogido al azar, realizando búsquedas en Google y no se ha cogido una base de datos de imágenes que hubiese en internet preparada para el entrenamiento, por lo que la variedad que hay entre cada una de las imágenes es muy grande. Al haber seleccionado imágenes con una variedad tan grande, sería necesario disponer de muchas más imágenes para que durante el entrenamiento no se produzca sobreaprendizaje, o bien, haber elegido imágenes que no tuvieran tanta variedad entre sí. Al hablar de variedad nos referimos a que, por ejemplo, las personas que aparecen en las imágenes, algunas son de piel negra y otras de piel blanca, algunas visten con bañadores y otras con ropa de invierno, o bien, el fondo de una imagen cambia mucho con respecto al fondo de otra imagen. Por ello, si las imágenes seleccionadas para el

entrenamiento presentan gran variedad entre sí, será necesario tener más imágenes para que no se produzca sobreaprendizaje, o bien, obtener para el entrenamiento, imágenes que no presenten demasiada variabilidad entre sí, pudiendo reducir así el número de imágenes necesarias para realizar el entreno.

Una buena pregunta sería, ¿en qué iteración sería conveniente parar y quedarnos con ese modelo? La respuesta habría que buscarla observando las figuras 23 y 24. Se quiere obtener un modelo que tenga el mayor porcentaje de precisión para el conjunto de test y el menor error posible. Para ello, se visualizan las dos gráficas y se aprecia que entre las iteraciones 100 y 125, el error comienza a subir y la precisión se queda estancada, por tanto, lo más conveniente es parar el entrenamiento entre alguna de esas iteraciones.

En nuestro caso, nos hemos quedado con los pesos que teníamos en la iteración 110, obteniendo un error en el test del 0.5 y del 0.2 en el entrenamiento y con una precisión sobre el test del 82%, y en el entrenamiento del 90%.

## **4.3. DETECCIÓN**

### **4.3.1. BASE DE DATOS**

Para el entrenamiento, es necesario tener un conjunto de imágenes de entrenamiento etiquetadas. Solo es necesario tener imágenes de las clases que se quieran detectar, en nuestro caso, imágenes que contengan personas, que ya se tenían recopiladas de la parte de clasificación. Para etiquetar las imágenes, se ha hecho uso de un software de libre distribución [16].

Dicho software ha facilitado muchísimo el etiquetado de las imágenes, aunque aun así, ha costado bastante tiempo etiquetar cada una de las 2106 imágenes que se han utilizado para el entrenamiento. El procedimiento para etiquetar es, abrir cada imagen con el software e ir marcando mediante un rectángulo a cada persona que aparezca en la imagen. El software, automáticamente crea un fichero .xml para cada imagen, donde almacena los datos de los rectángulos creados, es decir sus coordenadas 'x' e 'y', y su ancho y alto. Este fichero .xml era complicado y costoso computacionalmente de leerlo mediante un programa python, así que se utilizó matlab, que tiene una librería que facilita mucho la lectura de archivos .xml, para hacer un programa que leyese cada uno de estos ficheros creados, y crease nuevos ficheros con el mismo nombre, pero con extensión .txt, estructurados de una forma más sencilla de entender, para que fuese más fácil luego poder trabajar con python al momento de entrenar el modelo. Estos nuevos ficheros se estructuraron de forma que cada línea tenía únicamente los datos de un rectángulo ('x', 'y', alto y ancho). Cada fichero tiene el mismo nombre que la imagen a

la que hace referencia, pero con la extensión .txt, es decir, los datos de etiquetado de la imagen 'persona\_1.bmp' se encuentran en el fichero 'persona\_1.txt'.

De esta forma, ya se tienen las imágenes de entrenamiento y las etiquetas de estas. Ahora, lo que se hizo fue almacenar todos los datos de las anclas, ya que un ancla en realidad es un rectángulo, que también tiene sus coordenadas 'x', 'y', alto y ancho, que son las mismas para cualquier imagen. Por tanto, en el fichero 'anclas.txt', se almacenaron las coordenadas de cada ancla.

Por último, queda etiquetar, para cada imagen de entrenamiento, sus anclas positivas y negativas, y almacenarlas en un fichero, para que en cada iteración del entreno no haya que calcularlo de nuevo. Para llevar a cabo este etiquetado, lo que se puede hacer es ir recorriendo cada una de las anclas e ir comparándolas con los bounding boxes reales que contienen a las personas que se encuentran en la imagen. Estos bounding boxes reales son los que indicamos mediante el software que explicamos anteriormente y cuyos datos se almacenan en el fichero 'persona\_x.txt'. Consideramos que un ancla es positiva si tiene un IoU mayor a 0.7 con respecto a algún bounding box real. Se considera que un ancla es negativa si tiene un IoU menor a 0.3 con respecto a todos los bounding boxes reales. En caso de que no haya ningún ancla con un IoU mayor a 0.7, se considera como única ancla positiva el ancla con mayor IoU, por lo tanto, siempre se tendrá al menos 1 ancla positiva. Estos datos se almacenaron en ficheros asociados a cada imagen, de forma que la imagen 'persona\_22.bmp' tendrá los datos de sus anclas positivas y negativas almacenadas en el fichero 'img\_22.pkl'.

### 4.3.2. ARQUITECTURA

En este apartado, se van a ver los distintos hiperparámetros que han sido seleccionados para la implementación del modelo de detección y a su vez se verá mediante un esquema (figura 25) cómo es la estructura de capas de nuestro modelo. Se hablará también de las imágenes seleccionadas para el entrenamiento y cómo han sido etiquetadas y de algunas otras consideraciones que se han tenido en cuenta al implementar la red.

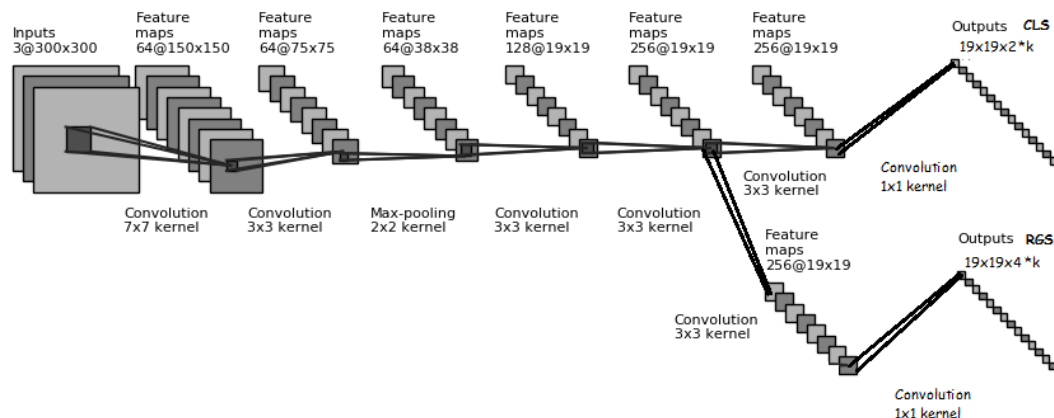


Figura 25. Modelo de detección. Fuente: Elaboración propia

La figura 25 representa la estructura de nuestro modelo de detección. Se puede ver, como en el modelo de clasificación, que las imágenes de entrada tienen también un tamaño de 300x300x3. Para elaborar este modelo, se ha cogido una parte del modelo de clasificación, con sus pesos ya ajustados, y se han congelado, para que durante el entrenamiento del detector no cambien de valor. Se ha cogido la primera parte del modelo de clasificación, desde la primera capa hasta la cuarta capa, en la que obtenemos un mapa de características de 19x19x128. El resto de capas son nuevas, añadidas para la detección y por tanto, se tendrán que ajustar sus pesos.

Tras las cuatro primeras capas, que se repiten de la clasificación, encontramos una nueva capa convolucional con 256 filtros de tamaño 3x3x128 y un stride de 1. Tras esto, se empezará a diferenciar la parte de clasificación (CLS) de la que se encarga de la regresión de las anclas (RGS). Se tienen dos capas convolucionales en paralelo, una para la parte CLS y otra para la RGS. Estas capas convolucionales están compuestas por 256 filtros de tamaño 3x3x256, utilizando un stride de 1.

Tras estas capas, les seguirán a cada una de ellas otra nueva capa convolucional con filtros de tamaño 1x1x256 y utilizando un stride de 1. En el caso de la parte de CLS, se utilizarán  $2*k$  filtros (2, porque tiene que determinar la probabilidad de que contenga un objeto y la probabilidad de que no lo contenga) y en el caso de la parte de RGS se utilizarán  $4*k$  filtros (4, porque las anclas vienen expresadas mediante 4 valores, 'x', 'y', alto, ancho), siendo 'k' el número de anclas por punto, que en nuestro caso  $k=9$ , ya que se tienen como posibles tamaños de anclas 100, 200, 300 píxeles y como posibles relaciones de aspecto 0.5, 1, 1.5, así que en total, combinándolas, se tienen 9 distintas anclas.

En este caso no se han utilizado capas dropout para tratar de reducir el sobreajuste del modelo.

Como se ha dicho anteriormente, se parte del modelo de clasificación, utilizando las primeras capas de dicho modelo hasta obtener el mapa de características de tamaño 19x19x128, porque se ha considerado que era el tamaño más adecuado para alimentar a la red de detección, ya que como salida se tendrá un vector de 19x19x9, en nuestro caso, ya que se tiene un mapa de características de 19x19 y 9 distintas anclas, por tanto se

tiene un vector de salida de tamaño 3249. Teniendo este tamaño de mapa de características, se tendrá un grupo de 9 anclas cada 'r' píxeles, siendo

$$r = \left( \frac{(Tam.Img.Input)}{(Tam.MapaCaract.)} \right)$$

Eq. 5

En nuestro caso,  $r = 300/19 = 15.79$ , por tanto, representando las anclas en la imagen original se tendrá un grupo de 9 anclajes definidos cada 15.79 píxeles. Para entender mejor el efecto de esto, observad la figura 19. Cada punto rojo que se ve en dicha figura representa que ahí se definirán un grupo de anclas. En nuestro caso, la distancia entre cada punto rojo es de 15.79 píxeles. Consideramos este una buena distancia entre cada grupo de anclajes y por ello se ha cogido un tamaño de 19x19 para el mapa de características.

Luego, se han añadido unas nuevas capas convolucionales para la detección. Se ha comenzado añadiendo el menor número de capas posibles, como se hizo en la clasificación, para tratar de evitar el sobreajuste. Nos percatamos que añadiendo el mínimo número de capas convolucionales posibles, ya se producía un gran sobreajuste y además el modelo no era capaz de detectar bien ni siquiera para las imágenes de entrenamiento. Por ello, se añadieron dos capas convolucionales más, una para la parte CLS y otra para la parte RGS, y así se consiguió que el modelo tuviese una gran precisión y un error muy bajo para el conjunto de entrenamiento, aunque para el conjunto de test se seguía manteniendo el mismo sobreajuste. El objetivo principal de este proyecto era aprender cómo funcionaban las CNNs e implementar una para hacer clasificación y otra para hacer detección de objetos en imágenes, objetivo que ya se tenía cumplido; sin embargo, se decidió también tratar de reducir el sobreajuste del modelo para obtener un menor error en el conjunto de imágenes de test, así que aprendimos la técnica del weight decay. Esta técnica consiste en que cada iteración del entrenamiento, los pesos de la red se disminuyen de forma proporcional a un valor determinado, teniendo como efecto que la red no se adapte demasiado rápido a las imágenes de entrenamiento y consiguiendo así, en algunos casos, reducir el sobreajuste. En nuestro caso, el sobreajuste se mantuvo tras la implementación de esta técnica, por tanto, también se probó añadir capas dropout entre las nuevas capas convolucionales que se añadieron para la detección, pero el error para el test no varió mucho con respecto a la red sin la técnica del weight decay y el dropout.



### 4.3.3. ENTRENAMIENTO Y RESULTADOS

Una vez se tienen todos los datos para comenzar el entrenamiento, en cada iteración se va cogiendo una imagen y sus correspondientes datos de anclas positivas y negativas, y se forma un batch con dichas anclas, de forma que este contenga el mismo número de anclas positivas que negativas. Se tendrán dos errores en la red, el de la parte de clasificación, CLS, y el de la parte de regresión, RGS. El error de la parte CLS se calcula comparando cada ancla que se había etiquetado como positiva y negativa del batch con las etiquetas que devuelve el modelo en la parte CLS, en esa misma posición del ancla del batch. Es decir, si por ejemplo se tiene como ancla positiva, un ancla que se encontraba en la posición  $8 \times 8 \times 2$  con respecto a la matriz de salida de la parte CLS para una imagen, cuya matriz de salida tiene un tamaño total de  $19 \times 19 \times 9$ , entonces para calcular el error cometido en ese ancla, se tendrá que comparar con respecto a esa misma posición  $8 \times 8 \times 2$  de la matriz de salida que devuelva la parte CLS. Las anclas positivas se etiquetarán con el valor  $[1, 0]$ , indicando que la probabilidad de que contenga una persona es de un 100%, mientras que se etiquetará con un valor  $[0, 1]$  las anclas negativas.

Para la parte RGS el error se calcula solo para las anclas positivas, de forma que se recorrerán cada una de las anclas positivas y se verá qué salida devuelve la matriz de la parte RGS, en esa misma posición. El bounding box que se dibujará en la imagen vendrá dado por la suma de las coordenadas del ancla en cuestión más la salida de que obtenemos de la matriz de la parte RGS. Es decir, dicha matriz indica el ajuste que hay que hacerle al ancla para ajustarla mejor a la persona que está conteniendo. Para entrenar esta parte, a las anclas positivas, se les suman la salida de la parte RGS y se calcula el error con respecto a las coordenadas de los bounding boxes reales correspondientes, ya que un ancla es positiva porque está muy próxima a un bounding box real, por tanto el error de dicho ancla se calcula con respecto a ese bounding box real y no con respecto a otro que pueda haber en la imagen.

Eso es todo, se irán realizando iteraciones, calculando el error que se va cometiendo tanto para la parte CLS como para la parte RGS y mediante un algoritmo de optimización del error, se van ajustando los pesos del modelo, cometiendo así cada vez un error menor para el conjunto de imágenes de entrenamiento. El error que se optimiza es la suma del error de la parte CLS y el error de la parte RGS.

Se va a visualizar la figura 26 donde se representa el error obtenido durante el entrenamiento del detector, para las imágenes de entrenamiento y para las imágenes de test.

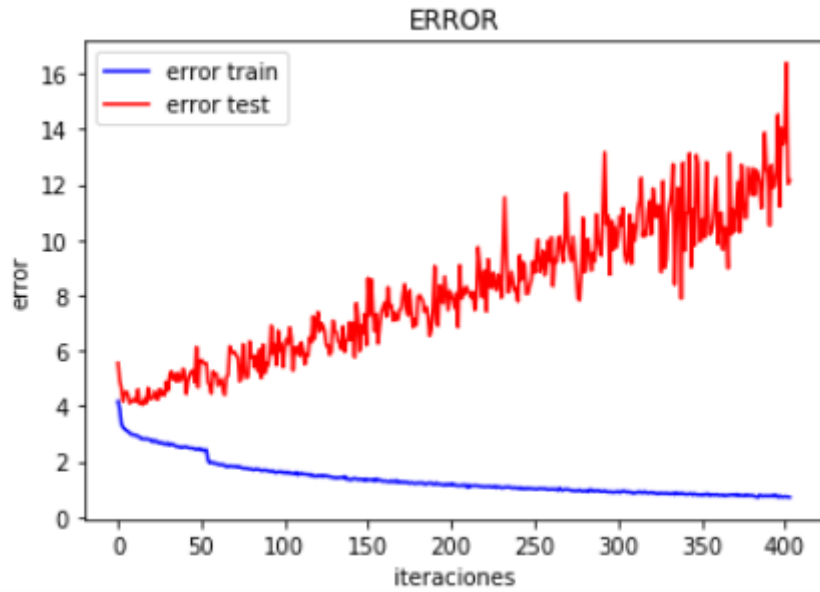


Figura 26. Error detección. Fuente: Elaboración propia

Se aprecia como el error para el conjunto de imágenes de entrenamiento va disminuyendo desde el principio, hasta la iteración 400 en la que se para el entrenamiento, donde consiguiendo un error del 0.3 para dicho conjunto. Sin embargo, para el conjunto de test, el error va en constante aumento desde el comienzo. Esto es otro claro ejemplo de sobreaprendizaje, en el que el modelo aprende muy bien a detectar en las imágenes de entrenamiento, pero se adapta demasiado a estas imágenes, y cuando se prueba el modelo en otras imágenes distintas, como el conjunto de imágenes de test, el modelo se comporta de manera muy diferente y no consigue detectar a las personas con la misma precisión con la que lo hacía en las imágenes de entrenamiento.

Se va a mostrar ahora una prueba realizada (figura 27), una vez entrenado el detector de personas en imágenes, con una selección de 6 imágenes del conjunto de imágenes de entrenamiento, en las que se puede apreciar la variedad existente en dicho conjunto. Esto servirá para ver cómo trabaja el modelo en dicho conjunto.



Figura 27. Pruebas detector Train. Fuente: Elaboración propia

A continuación se mostrará una prueba (figura 28) con 6 imágenes del conjunto de imágenes de test, para comprobar el funcionamiento del modelo en dicho conjunto. El resultado obtenido en estas 6 imágenes de test seleccionadas es similar al resultado obtenido en el resto de imágenes del conjunto de test. Se han seleccionado estas 6 imágenes porque las consideramos representativas del resto del conjunto.

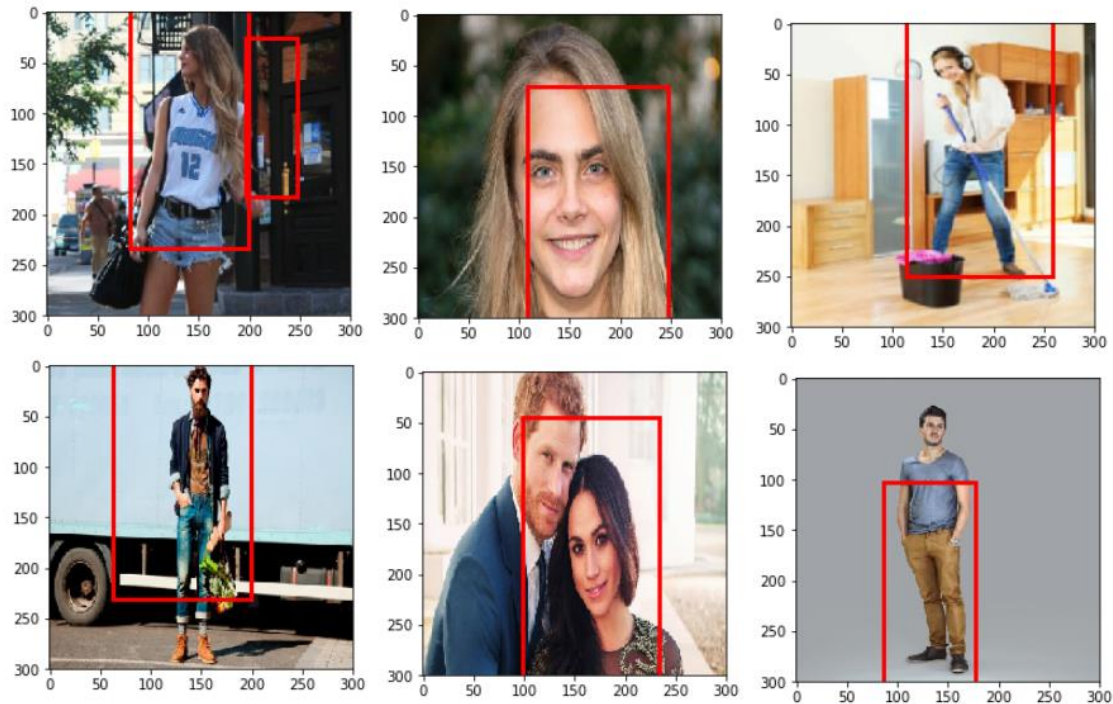


Figura 28. Pruebas detector Test. Fuente: Elaboración propia

#### 4.3.4. ANÁLISIS

Como se puede ver en las imágenes que se han mostrado (figura 27 y 28), tanto del conjunto de entrenamiento como del conjunto de test, se aprecia la gran variedad existente entre cada imagen, que, como se comentó anteriormente, puede tener consecuencias como que se de sobreaprendizaje, que es lo que ha ocurrido en nuestro modelo, como se vio en las figuras 23 y 26, en las que se representaba el error en ambos modelos.

Como se puede apreciar en la figura 27, para el conjunto de imágenes de entrenamiento, el detector de personas en imágenes funciona muy bien, no solo para estas 6 imágenes que han sido seleccionadas por ser las más representativas, sino para cualquier otra imagen del conjunto. La cosa cambia bastante cuando se hacen las pruebas para el conjunto de imágenes de test, como se puede ver en la figura 28. Se puede ver como la precisión del modelo disminuye bastante cuando se utilizan imágenes distintas a las de entrenamiento.

En el caso de la figura 28 se ve como el detector trabaja medianamente bien. El bounding box que devuelve la red, salvo en ocasiones muy puntuales, contiene una persona dentro, aunque en algunos casos el bounding box no se ajusta demasiado bien a

la persona que está marcando. También suele cometer fallos cuando hay varias personas muy juntas en la misma imagen, como es el caso de la imagen de la fila inferior, la columna central, que solo marca un bounding box que cubre a las dos personas existentes, pero el resultado deseado es que marcarse con un bounding box a cada persona, de forma independiente.

## **5. CONCLUSIONES**

En este último capítulo de la memoria se expondrán los objetivos conseguidos con la realización de este proyecto, se hablarán de las posibles mejoras que se podrían realizar en el futuro a los sistemas de clasificación y detección implementados y se comentará cómo se piensa que va a evolucionar esta tecnología en los próximos años.

### **5.1. PRINCIPALES APORTACIONES**

Mediante la realización de este proyecto se ha aprendido a saber cómo funcionan las CNNs e incluso a implementarlas con la ayuda de la librería TensorFlow. Además de esto, también nos ha ayudado para reforzar algunos conocimientos que se habían adquirido en algunas de las asignaturas que se imparten en el grado, de forma que ahora se han entendido más en profundidad.

No solo se han aprendido los objetivos que se habían propuesto al principio, sino que también se ha aprendido a saber cómo funcionan otras cosas interesante y útiles para realizar sistemas con técnicas de aprendizaje automático. Se exponen a continuación un resumen de los ítems conseguidos.

- Comprender la historia y evolución del DL.
- Aprender los fundamentos del DL y las principales arquitecturas y capas de las CNNs.
- Aprender el funcionamiento del lenguaje de programación Python y de la librería TensorFlow.
- Técnicas de aumento de datos para imágenes.
- Elaboración de bases de datos de imágenes etiquetadas para ML.
- Fundamentos e implementación de un sistema de clasificación mediante CNN.
- Fundamentos e implementación de un sistema de detección mediante CNN.

- Fundamentos e implementación de un sistema de detección mediante la API disponible en la web de la librería TensorFlow. En el siguiente apartado se explicará en profundidad cómo funciona esta API.
- Comprensión del potencial que tiene ML de cara al futuro para resolver problemas que hoy en día no tienen solución.
- Por curiosidad y como son también unas de las redes más utilizadas del momento en DL, también se han aprendido los fundamentos de las Redes Neuronales Recurrentes y del Deep Q-Learning, que es una evolución del algoritmo Q-Learning utilizado para aprendizaje por refuerzo aplicando DL. Se ha estudiado este algoritmo debido a que en la asignatura del grado, Aprendizaje Automático, se pedía realizar un trabajo sobre aprendizaje por refuerzo y se consideró interesante estudiar este algoritmo ya que se estaba trabajando también en este proyecto.

Como conclusión personal del trabajo realizado, considero que he aprendido muchísimas cosas nuevas que no se enseñan en ninguna de las asignaturas actuales del grado y que en la actualidad han tenido un gran impacto en la industria y pienso que en el futuro el impacto será aún mayor a medida que las computadoras más potentes de hoy en día vayan reduciendo su precio, pudiendo así cualquier persona investigar e implementar sistemas de DL. Además, este proyecto me ha ayudado a entender mejor la importancia del aprendizaje automático en la industria dada la cantidad de posibilidades que esta tecnología proporciona para resolver problemas que hace algunos años no eran posibles de resolver mediante técnicas clásicas.

## 5.2. LÍNEAS FUTURAS

En el futuro, como posibles mejoras a los sistemas de clasificación y detección implementados, se trataría de mejorar la precisión de estos en el conjunto de imágenes de test de forma que se evite el sobreajuste que se ha dado. Para ello, se podrían obtener más imágenes para el entrenamiento y seguir teniendo esa variedad entre las imágenes, pero al tener más cantidad de datos el sobreaprendizaje se reduciría, o bien, se seleccionarían de las imágenes que ya se tienen, aquellas que tengan poca variedad entre sí, y en función de la cantidad de imágenes que se obtengan, después se buscarán más o si se tiene una cantidad suficiente para entrenar el modelo, se entrenará la red con ese conjunto de entrenamiento y se espera que así se reduzca el sobreaprendizaje que se produce actualmente.

También, se quisiera optimizar el tiempo que tardan los modelos en analizar cada imagen, para que tarden el menor tiempo posible, para que en el futuro estos sistemas puedan implementarse y ejecutarse en una Raspberry Pi, para realizar, por ejemplo, un sistema de seguridad que detecte personas, o bien, un proyecto que me gustaría realizar, ya que considero que aprendería muchas cosas importantes, sería uno que consistiría en mover un motor, el cual tiene pegado un laser, y mediante una cámara, cuando se



detectase una persona, el motor se moviese de forma que el laser apuntara a esa persona. Ese proyecto sería algo más sencillo de lo que hacen los brazos robóticos actuales, que cogen determinados objetos situados en un espacio determinado detectándolos con una cámara. En el caso de los brazos robóticos, tienen que mover varios motores para situar la pinza en el lugar correcto y el proyecto que he comentado solo sería necesario mover un motor, por lo que sería bastante más sencillo de realizar pero pienso que serviría para entender cómo funcionan esos brazos robóticos que están teniendo una gran importancia en la industria para la automatización de procesos.

En cuanto a la tecnología de DL, en el futuro, se considera que seguirá evolucionando y seguirá abriendo puertas en cada sector, como ya ha hecho en la actualidad, por ejemplo, en el sector automovilístico, aportando un gran avance en los coches autónomos, convirtiéndolo hoy en día en una gran tendencia y un problema que muchas grandes empresas se han puesto a tratar de resolver, ya que se ha dado un gran salto gracias al DL.

En la actualidad surge el problema de que recopilar una gran cantidad de datos para el entrenamiento de sistemas de DL se hace pesado, ya que suele ser necesario una gran cantidad de datos para ello. Además, en el caso de realizar un detector de objetos, como ha sido nuestro caso, es necesario etiquetar todas esas imágenes de forma que hay que ir seleccionando uno a uno cada objeto que aparece en cada imagen, y esto se vuelve un trabajo demasiado pesado y que se tarda mucho en realizar. En el apartado anterior se comentó que existía una API que proporcionaba TensorFlow, para la creación de sistemas de detección. Se comentarán a continuación aspectos de esta API porque se considera que en el futuro próximo esta será la forma más utilizada de crear sistemas de detección dada la comodidad, facilidad y eficacia que proporciona.

Esta API se basa en la idea de la transferencia del aprendizaje. TensorFlow proporciona en su repositorio de Github dicha API y varios modelos preentrenados. Funciona de forma que reentrena uno de los modelos que se elija, de los que TensorFlow proporciona, que estaba preentrenado por otro dataset, de forma que la API congela algunas de las primeras capas del modelo, las que considere oportunas, y reentrena el modelo, actualizando los pesos de las últimas capas para que el modelo se ajuste al nuevo dataset. Algunos de los modelos preentrenados que proporciona TensorFlow, actualmente, son los que se muestran en la figura 29. Se puede ver que cada modelo tiene tres columnas. La columna ‘Speed’ indica la velocidad que tarda el modelo, en milisegundos, en tratar una imagen. La columna ‘COCO mAP’ indica el porcentaje de acierto de dicho modelo para un conjunto de datos de test. La última columna ‘Outputs’ indica lo que devuelve el modelo, el valor ‘boxes’ indica que devuelve bounding boxes, ya que es un modelo que se encarga de detectar. En función de si se quiere un modelo más o menos rápido y más o menos preciso, se elegirá un modelo u otro.

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_0.75_depth_coco ☆	26	18	Boxes
ssd_mobilenet_v1_quantized_coco ☆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ☆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ☆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ☆	56	32	Boxes
ssd_resnet_50_fpn_coco ☆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes
ssdlite_mobilenet_v2_coco	27	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes
mask_rcnn_inception_resnet_v2_atrous_coco	771	36	Masks
mask_rcnn_inception_v2_coco	79	25	Masks

Figura 29. Modelos API TensorFlow. Fuente:

[https://github.com/tensorflow/models/blob/master/research/object\\_detection/g3doc/detection\\_model\\_zoo.md](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md)

Esos modelos, están entrenados mediante el conocido dataset de imágenes denominado COCO, que contiene más de 330.000 imágenes de entrenamiento de 80 distintas clases de objetos. Esto implica que dichos modelos preentrenados con ese dataset, detectan perfectamente características de bajo nivel, como bordes, esquinas o circunferencias, que son características comunes de cada objeto y son necesarias de detectar para casi cualquier problema de detección de objetos en imágenes. Esas características de bajo nivel, que servirán para casi cualquier problema, son las que se detectan en las primeras



capas del modelo, por ello se congelan sus pesos, y solo se actualizan los pesos de las últimas capas durante el entrenamiento.

Para crear un sistema de detección mediante esta API, no es necesario tener una cantidad de imágenes tan grande como al tratar de entrenar un modelo desde cero, teniendo que ajustar todos los pesos de la red, por ello pienso que en el futuro próximo esta API será muy utilizada, ya que tiene una gran eficacia y el tiempo que se tarda en el proceso de recopilar imágenes para el entrenamiento lo acorta mucho ya que no es necesario mucha cantidad de imágenes. Tan sólo se tiene que preocupar de recopilar las imágenes necesarias para el conjunto de imágenes de entrenamiento y test, y etiquetar dichas imágenes marcando cada objeto que se quiera detectar. Luego, tan sólo hay que adaptar dichas etiquetas a la forma que tiene la API de leerlas, pero para ello ya existen programas de libre distribución que hacen esa conversión de las etiquetas.

Se ha probado a utilizar esta API para nuestro problema de detección en imágenes de personas, para comparar el resultado obtenido en cuanto a precisión, con el sistema de detección, Faster R-CNN, que se ha creado desde cero. Como ya se tenía el conjunto de imágenes de entrenamiento y de test, solo se han tenido que adaptar las etiquetas que se tenían para que la API pudiese leerlas correctamente. En tan solo unas 2 horas, ya se tenía todo listo para el entrenamiento mediante la API y 2 horas más tarde, tras ponerla en marcha para el reentrenamiento, ya se tenía el modelo listo, ya que no se conseguía reducir más el error para el test ni el entrenamiento. Se han comparado los resultados obtenidos, de forma visual, mostrando cómo actuaba cada uno de los dos detectores para las mismas imágenes de test. Se observó que la API había conseguido un modelo que mejoraba un poco la detección de las personas en las imágenes con respecto al modelo que se había creado desde cero, y eso que se seleccionó el modelo preentrenado `ssd_mobilenet_v1_coco`, que es uno de los modelos que menor precisión tiene pero a su vez es de los más rápidos.

Por todo esto, pienso que en el futuro próximo, cuando a una empresa o a un proyecto les sea necesario crear un sistema de detección de objetos en imágenes, acudirán a esta API debido a que el costo en cuanto a tiempo y por lo tanto en presupuesto será mucho menor que crear un sistema desde cero y obtendrán una gran eficacia y precisión. El problema se puede dar cuando se pretendan detectar objetos en imágenes que no se parecen en nada a las que hay en el dataset COCO. En ese caso, quizás no sirva la utilización de esta API y sea necesario crear un sistema de detección desde cero, como se ha hecho en este proyecto, pero como el tiempo requerido para obtener un modelo entrenado para un dataset mediante esta API es pequeño, a mi parecer, lo más recomendable sería probar a ver qué resultados se obtienen y en función de si se necesita más efectividad o no, entonces se creará una red de detección desde cero o se quedarán con el modelo que ha proporcionado la API.

## 6. REFERENCIAS

- [1] <https://psicologiaymente.com/neurociencias/ley-de-hebb>
- [2] <http://www.mind.ilstu.edu/curriculum/modOverview.php?modGUI=212>
- [3] <https://myclouddoor.com/es/deep-learning-un-recorrido-historico/>
- [4] [https://es.wikipedia.org/wiki/Redes\\_neuronales\\_convolucionales](https://es.wikipedia.org/wiki/Redes_neuronales_convolucionales)
- [5] <https://www.kaggle.com/toregil/welcome-to-deep-learning-cnn-99>
- [6] <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>
- [7] [https://www.tensorflow.org/api\\_docs/python/tf/contrib/layers/xavier\\_initializer](https://www.tensorflow.org/api_docs/python/tf/contrib/layers/xavier_initializer)
- [8] <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- [9] <http://ruder.io/optimizing-gradient-descent/>
- [10] <https://hackernoon.com/%EF%B8%8F-big-challenge-in-deep-learning-training-data-31a88b97b282>
- [11] <https://arxiv.org/pdf/1311.2524.pdf>
- [12] <https://arxiv.org/pdf/1504.08083.pdf>
- [13] <https://github.com/tryolabs/luminoth/tree/master/luminoth/models/fasterrcnn>
- [14] <https://github.com/tzutalin/labelImg>
- [15] <https://arxiv.org/pdf/1506.01497.pdf>
- [16] <https://github.com/tzutalin/labelImg>

## 7. BIBLIOGRAFÍA

- Guía para principiantes para comprender el funcionamiento de las CNNs de forma básica: <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/>
- Visualización de los filtros de las distintas capas de una CNN tras su entrenamiento, para comprender en profundidad el funcionamiento de estas: <https://www.youtube.com/watch?v=AgkfIQ4IGaM>, <http://yosinski.com/deepvis>
- Charla en la Universidad de Chile sobre DL y CNN, en la que explican los conceptos claves de estas y aportan consejos para la implementación de las CNNs: [https://www.youtube.com/watch?v=GPz0VqS\\_5VE](https://www.youtube.com/watch?v=GPz0VqS_5VE)
- Libro introductorio a la biblioteca TensorFlow para iniciarse en la programación de DL y en concreto CNNs: <http://jorditorres.org/libro-hello-world-en-tensorflow/>
- Curso Online de la Universidad de Stanford en el que se explica en profundidad las CNNs aplicadas a reconocimiento de imágenes, desde los conceptos teóricos más complejos de estas hasta su implementación y evaluación de resultados: <http://cs231n.github.io/>, <http://cs231n.stanford.edu/syllabus.html>
- Resumen general de DL y del funcionamiento e implementación de las CNNs, comentando las diferentes arquitecturas existentes, las distintas herramientas de software existentes para su implementación y sus aplicaciones: <https://ccc.inaoep.mx/~pgomez/deep/presentations/2016Loncomilla.pdf>
- Explicación en profundidad del funcionamiento de Faster R-CNN, de sus conceptos teóricos, su arquitectura y de cada parte en la que esta se divide: <https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/>
- Explicación del funcionamiento de Intersection over Union (IoU) mediante su teoría y mediante ejemplos y su posterior implementación en código: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>
- Explicación del funcionamiento de la API que ofrece TensorFlow para la creación de sistemas de detección de objetos en imágenes. Se trata de un tutorial donde se explica mediante un ejemplo como hacer uso de esta API para generar un sistema de

detección de objetos en imágenes adaptado a los objetos que se quieran detectar. El tutorial viene dividido en 6 partes cuyos links se encuentran aquí:

(Parte 1) <https://www.youtube.com/watch?v=COlbP62-B-U&t=472s>

(Parte 2) <https://www.youtube.com/watch?v=MyAOtvwTkT0>

(Parte 3) [https://www.youtube.com/watch?v=K\\_mFnvzyLvc](https://www.youtube.com/watch?v=K_mFnvzyLvc)

(Parte 4) [https://www.youtube.com/watch?v=kq2Gjv\\_pPe8](https://www.youtube.com/watch?v=kq2Gjv_pPe8)

(Parte 5) <https://www.youtube.com/watch?v=JR8CmWyh2E8>

(Parte 6) <https://www.youtube.com/watch?v=srPndLNMMpk>

- Redes neuronales recurrentes para principiantes, explicando los conceptos básicos de estas y sus aplicaciones: <https://medium.com/@camrongodbout/recurrent-neural-networks-for-beginners-7aca4e933b82>
- Redes neuronales recurrentes en profundidad y tutorial de LSTM en Python con Tensorflow: <http://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/>
- Explicación en profundidad del algoritmo Deep Q Learning, sus conceptos teóricos, su implementación en código y sus aplicaciones en la actualidad: [https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/deep\\_q\\_learning.html](https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/deep_q_learning.html), <https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8>