Universidad de Valladolid

Detección e identificación automática de actrices y actores mediante el uso de algoritmos de Deep Learning.

Trabajo de Fin de Grado

Grado de Estadística

Universidad de Valladolid

Autor: Adrián Atienza Arroyo

Tutor: Agustín Mayo Íscar

Cotutor: Diego Calvo

Introducción.

Contexto, motivación y objetivos.

La evolución del Deep Learning en los últimos años, debido al desarrollo exponencial de la capacidad de computación de los ordenadores personales, hace que se haya convertido en una herramienta popular, al alcance de cualquiera con conocimientos suficientes sobre programación y estadística.

Reconocimiento de figuras en imágenes, procesamiento de lenguaje natural o clasificación de archivos de sonidos son tareas que pueden parecer triviales, sobre todo comparadas con cálculos de física o matemática avanzados, pero lo cierto es que, a diferencia de los segundos, que una computadora normal los puede resolver sin problemas, el primer grupo de tareas resultaba ser un grupo de problemas inabordables para una computadora hasta la llegada del Deep Learning.

Lo normal ahora es que ustedes vean la utilidad a que un ordenador pueda resolvernos complejos sistemas de ecuaciones en cuestión de décimas de segundo, pero que no le vean una clara finalidad a que nuestro ordenador nos diga si una foto que tenemos delante es un perro o un gato, o que si la música que suena es de género pop o rock. Tenemos que ir un poco más allá para ver las implicaciones que puede tener esto. Estamos haciendo que un ordenador sea capaz de percibir el mundo igual que lo percibimos nosotros, reconocer y clasificar estímulos que le llegan desde el exterior, por lo que el siguiente paso lógico es las computadoras sustituyan a humanos no solo en tareas puramente mecánicas como sacar dinero de una cuenta bancaria o cobrarte la compra en un supermercado, sino que también en tareas que requieren de más "instinto" como conducir o diagnosticar a pacientes.

En este último paso de mecanizar tareas humanas con tecnología Deep Learning es donde entra este Trabajo de Fin de Grado. Entré a formar parte de un proyecto de desarrollar una aplicación para la empresa *Bravent* que sea capaz de concretar cuanto tiempo aparecen en pantalla cada actor/actriz que forma parte del reparto de una película dada.

Hasta ahora, esta tarea era manual y la realizaba una persona con un cronómetro. La idea es que una herramienta computacional sea capaz de realizar esto de manera más precisa, en mucho menor tiempo que la manera manual y de manera automática. Para esto se va a crear una aplicación dotada con Inteligencia Artificial.

El objetivo de este proyecto es implementar y evaluar distintas arquitecturas neuronales de vanguardia de Deep Learning sobre una base de datos abierta.

El primer capítulo describo una serie de conceptos diferentes, cuyos significados se suelen confundir o solapar.

El protagonista del segundo capítulo son las redes neuronales. Este capítulo es un camino que parte desde la neurona, el componente básico de cualquier red neuronal, y acaba en las redes convolucionales, usadas en el procesamiento inteligente de imágenes.

Como ya hemos dicho, las redes neuronales convolucionales es un campo de expansión constante, en este tercer capítulo profundizamos en dos de las arquitecturas más famosas que usaremos para realizar las pruebas que motivan este proyecto.

La fase experimental está descrita en el cuarto capítulo, en él se describen los detalles de la implementación y entrenamiento de las redes neuronales con las bases arquitectónicas descritas en el anterior capítulo, y los informes del comportamiento del modelo después del entrenamiento.

Estas redes descritas hasta el capítulo cuatro no son suficientes para cubrir la tarea que debe llevar a cabo la aplicación. En este último capítulo se describe la tarea *Object Detection* y un novedoso algoritmo que es capaz de realizar dicha tarea.

Índice general

Capítulo 1.

1. Inteligencia Artificial, Machine Learning, Deep Learning y Computer Vision.

1.1 Inteligencia Artificial

La inteligencia artificial, como su propio nombre sugiere, es la ciencia que estudia la inteligencia de las cosas artificiales. Aunque desde un punto de vista filosófico la definición no es trivial, pues habría que dejar claro en que consiste la inteligencia, desde un punto de vista de ingeniería aplicada, que es el que nos interesa, se trata de la creación de elementos que puedan realizar tareas de una manera similar a como las realizaría un humano. De manera inteligente.

Se dice que la primera persona que pensó en si es posible que el ser humano sea capaz de construir inteligencia fue Allan Turing. Además de ser considerado uno de los padres de las computadoras modernas, participando en el diseño de los primeros computadores para el ejército inglés durante la segunda guerra mundial, publicó un artículo (Turing, 1950) donde el mismo formulaba la pregunta de si eran capaces las computadoras de pensar.

Sin embargo, el término de Inteligencia Artificial se le atribuye a John McCarthy. Integrante de la fundación Rockefeller, cuya investigación trataba de construir una computadora que fuera capaz de realizar operaciones inteligentes en lugar de operaciones prefijadas.

Resumiendo, la Inteligencia Artificial desde un punto de vista técnico, que es el que nos interesa, es la creación de aplicaciones que imiten el comportamiento humano en la realización de distintas tareas. Para ello se pueden utilizar diversas herramientas y algoritmos, entre ellas el *Deep Learning,* herramienta en la que se basa mi proyecto y de la que se hablará más tarde. Para mayor grado de detalle, consultar el libro que me ha servido para entender que es la Inteligencia Artificial. (Romero, Dafonte, Gómez, & Penousal, 2007)

1.2 Machine Learning.

El Machine Learning, como bien se puede deducir por su nombre, es una serie de técnicas y algoritmos cuyo fin es enseñarle a una máquina a hacer una determinada tarea, como puede ser de clasificación o de regresión, mediante la ingesta de datos y el aprendizaje que desarrolla con ellos, de manera que, en un futuro, debe ser capaz de predecir el resultado deseado para una nueva instancia, con el mejor rendimiento posible. Hay tres grandes tipos de aprendizaje supervisado, donde los datos están etiquetados en clases; el no supervisado, donde los datos no están etiquetados; y el aprendizaje de refuerzo, en el que no tenemos datos.

Hay diversos algoritmos que se usan en el desarrollo del Machine Learning, como Árboles de Decisión, Modelos de Regresión Lineal, Modelos de Regresión Logística o *Support Vector*

Machines, para más información sobre el funcionamiento de estos algoritmos y la teoría estadística que hay detrás de ellos, consultar (García & Fernández, 2016).

1.3 Deep Learning

1.3.1 Introducción al Deep Learning

El Deep Learning es una continuación del Machine Learning que viene dada por la aparición de una nueva herramienta, las redes neuronales, denominadas así por encontrar inspiración en los estudios de neurociencia sobre el comportamiento del cerebro humano, son capaces de llevar el Machine Learning a otro nivel mucho más profundo, de ahí el nombre.

Una red neuronal es una arquitectura dividida en capas de neuronas artificiales. Las neuronas son unidades simples, que reciben información, la procesan mediante una función matemática simple, y la transmiten a grupo de neuronas conectadas a ella. La idea es que al igual que el cerebro, las neuronas son unidades muy simples, pero al formar una estructura compleja con gran cantidad de ellas, pueden dar solución a problemas de gran envergadura.

Mientras que el rango de problemas que se podían resolver con Machine Learning era bastante limitado, en cuanto las instancias comenzaban a tener un número relativamente alto de atributos, los algoritmos empiezan a mostrar un nivel de rendimiento bajo, el Deep Learning suple esa falta de rendimiento con un mucho mayor coste computacional para que nos sea posible procesar una mucho mayor gama de objetos, como pueden ser imágenes, archivos de audio o textos enteros, con resultados por qué no decirlo, sorprendentemente buenos.

1.3.2 Historia

Cabe reseñar que la herramienta de la red neuronal no es una cosa novedosa. Ya fue formulada por Donald Hebb en 1940 (Hebb & Penfield, 1940), y en 1958 Frank Rosenblatt creó el primer perceptrón, la arquitectura de red neuronal más simple. Es lógico entonces preguntarnos las razones por las que no se ha oído hablar de esta herramienta hasta la fecha, y son las mismas que ya hemos expuesto en apartados anteriores. El gran coste computacional requerido para desarrollar, sobre todo, las labores de entrenamiento de esta herramienta, la han limitado hasta hace bien poco, a una mera teoría hipotética. Con el exponencial desarrollo en estos últimos años del hardware en general, y sobre todo de las GPU en particular, han posibilitado trasladar esta herramienta desde el mundo teórico al práctico, haciendo posible su implementación en casi cualquier ordenador personal, colocando a la Inteligencia Artificial en un componente más en nuestra vida cotidiana como una extensión que mejora el rendimiento y la experiencia de usuario de casi cualquier máquina que conectamos a la electricidad a lo largo de nuestro día.

1.3.3 Deep Learning en la actualidad

La explosión de esta tecnología ni mucho menos ha tocado techo. El desarrollo de la misma sigue en paralelo con el de la ciencia en la que encontró inspiración, la neurociencia, y la aplicación de distintos principios de la misma en el desarrollo de nuevas arquitecturas neuronales que mejoran el rendimiento de una tarea en específico, por lo que es un campo que día a día sigue innovando y evolucionando.

Para finalizar esta breve introducción al Deep Learning, me parece necesario mencionar, aunque no sea el tema a tratar de este proyecto, que la irrupción de la inteligencia artificial en nuestra vida cotidiana ha provocado necesariamente la aparición de otro campo de estudio e investigación, que es el de la ética asociada a la Inteligencia Artificial. Los problemas morales y éticos que han de ser llevados a revisión son tantos, que daría perfectamente para otro Trabajo de Fin de Grado.

1.4 Computer Vision

Computer Vision (Roberts, 1960) es un campo dentro de la Inteligencia Artificial. Se empezó a oír hablar de este campo cuando se llegó a la conclusión de que el mundo visual es simplificado en formas simétricas simples, y el objetivo para que una computadora procese imágenes de manera similar al humano era reconocer y reconstruir dichas formas.

En 1966 nace el primer proyecto de construir un sistema visual inteligente en la universidad MIT. La duración de dicho proyecto era de un verano. Más de cincuenta años más tarde se sigue sin haber resuelto al 100% el problema de la visión artificial.

Este es el campo dentro de la Inteligencia Artificial que quiero explorar en este trabajo usando técnicas de Deep Learning, es decir, el uso de redes neuronales para el procesamiento de imágenes. Requiere de una arquitectura especial de redes neuronales en la que entraremos en profundidad más adelante.

No hace falta decir nada sobre el número de sensores y cámaras que hay en el mundo. Se estima que cada 3 segundos se suben a la nube 15 horas nuevas de contenido audiovisual a la plataforma de Youtube, (CISCO, 2017) predice que la mayor parte del tráfico de Internet hasta 2022 será en forma de vídeo.

De toda esta producción masiva de datos visuales, nace la necesidad del desarrollo de algoritmos que sean capaces de entender y utilizar todos estos archivos. Por eso el desarrollo de este campo del Deep Learning está siendo exponencial, con avances casi diarios, y cada vez son más los diferentes empresas que utilizan procesamiento inteligente de imágenes, ya sea en el campo de la medicina, para procesar imágenes de radiografías o de células y ayudar al médico en su diagnóstico, en el campo de la economía para sacar información de distintas gráficas de valores de un producto a lo largo del tiempo o en el sector del automóvil para el desarrollo de coches inteligentes que conduzcan solos.

1.5 Conclusión del capítulo

Después de haber diferenciado conceptos cuyo significado se suele solapar o confundir con facilidad, y de haber explicado qué es visión computacional, la importancia del procesamiento inteligente de las imágenes, el porqué del desarrollo exponencial del campo en los últimos años y las aplicaciones actuales que se le dan hoy en día a dicha tarea vamos a adentrarnos en las partes que conforman el Deep Learning y las modificaciones que se le deben realizar a la arquitectura básica de redes neuronales, el perceptrón multicapa, para que sea capaz de realizar dicha tarea de la manera más eficiente posible.

Capítulo 2.

Redes Neuronales, de la neurona a redes neuronales convolucionales.

En este capítulo hablaremos de la herramienta básica del Deep Learning, las redes neuronales en paralelo a el funcionamiento del cerebro humano en la captación de estímulos y procesamiento de respuestas, que es donde encuentra inspiración, y acabaré explicando las redes neuronales convolucionales. Para entender estas últimas hay que hacer un repaso desde los conceptos más básicos de la red neuronal: la neurona y el funcionamiento de la red neuronal más simple.

Al igual que el cerebro humano, la unidad básica de procesamiento es la neurona, y la potencia de procesamiento del sistema se basa en conexiones neuronales, en las que una neurona recibe información de un grupo de neuronas y la transmite a otro grupo de las mismas.

2.1 La Neurona

En nuestro sistema neuronal, la neurona recibe información a través de las dendritas, la procesa en el núcleo y la transmite a otras neuronas a través de las terminaciones del axón.

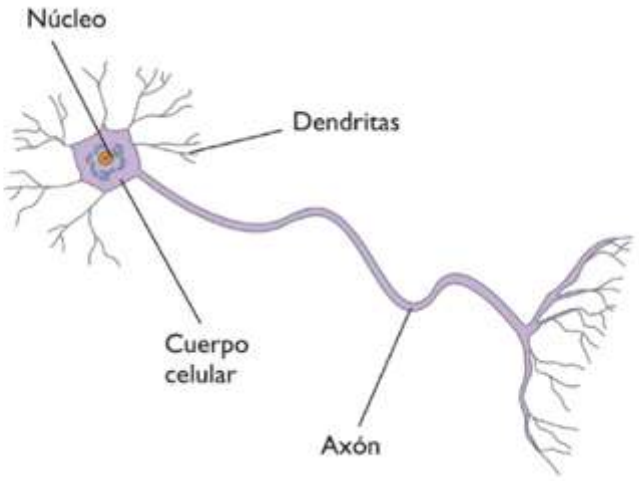


Ilustración 1: Esquema de una neurona biológica. Fuente: neuronas.*blogspot.com*

En 1943, McCulloch y Pitts (McCulloch & Pitts, 1943) formularon un modelado que explicaba el comportamiento de una neurona con base matemática, siendo este el resultado:

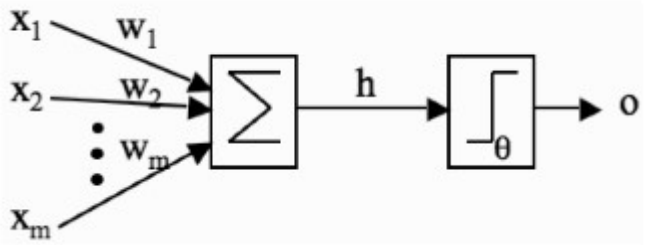


Ilustración 2: Esquema del modelo matemático de neurona artificial

- Un conjunto de entradas ponderadas con un peso cada una, que corresponden a la sinapsis.
- Un sumador de los valores de las entradas por sus pesos, más un valor de umbral b.
- Una función de activación, que toma el valor dado por el sumatorio y lo transforma mediante una función matemática.

El concepto clave, que es donde reside la fuerza de procesamiento de nuestro cerebro, y que se intenta emular mediante este sistema matemático es el de la sinapsis, las conexiones entre dos neuronas.

La neurociencia le dio el nombre de plasticidad a la capacidad de modificar la intensidad de las conexiones entre dos neuronas, o de crear nuevas de nuestro cerebro. Ni hoy en día está claro como el cerebro determina la fuerza de la intensidad de estas conexiones, pero en 1949 Donald Hebb expuso su teoría (Hebb D. , 1949) mediante la cual, si dos neuronas se activan a la vez constantemente, la intensidad de la conexión entre ellas deberá ser alta porque significa que responden a los mismos estímulos, y la conexión entre dos neuronas deberá ser nula en el caso contrario.

Este comportamiento es el que se quiere imitar mediante este modelo matemático, donde, mediante el algoritmo de aprendizaje de la red neuronal, del que hablaré más tarde, se pretende que la red configure los pesos de las sinopsis influidos por si las neuronas responden o no a los mismos estímulos.

También considero que hay que hacer mención a la función de activación, que transforma el sumatorio de los pesos por los valores, a el valor devuelto por la función. Lejos de paralelismos con nuestro sistema neuronal, una de las principales razones por las que está función de activación es indispensable es para evitar que una capa de neuronas sea combinación lineal de la anterior, lo que limitaría mucho el rango de acciones de la red y disminuiría mucho su capacidad. Las funciones de activación más utilizadas son la función Hiperbólica, la función ReLu o la Sigmoide.

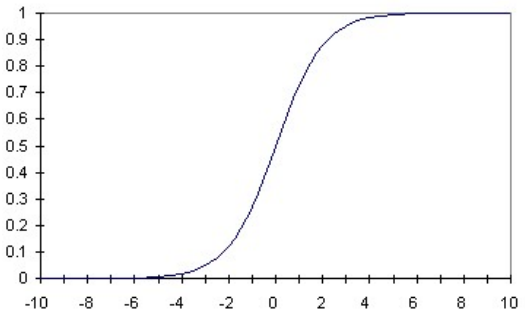
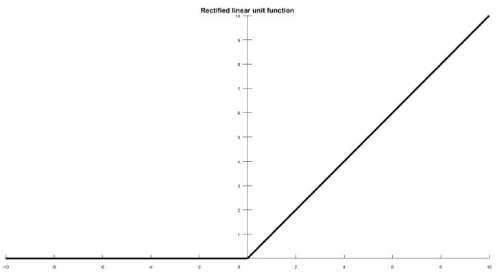
Nombre	Fórmula	Función
Sigmoide	$y = \frac{1}{1 + e^{-x}}$	
ReLu (Rectified Linear Unit Function)	$y = \max(x, 0)$	

Tabla 1: Comparativa entre la fórmula y la función de las funciones de activación Sigmoide y ReLu

Normalmente la más usada es la ReLu, ya que como explicaremos más adelante, el algoritmo de entrenamiento implica a la derivada de la función de activación. El valor de la derivada es la pendiente, y como podemos ver en la función sigmoide, la pendiente para valores grandes y pequeños es casi igual a cero, por lo que el valor de los pesos casi no se actualiza.

2.2 Perceptrón Simple

2.2.1 Introducción al Perceptrón Simple

La Figura muestra el esquema de la estructura más simple dentro de la arquitectura de redes neuronales.

- Cada circulo representa una red neuronal
- Se tienen tantas neuronas en la capa Input como entradas tiene el modelo
- Se tienen tantas neuronas en la capa Output como salidas posibles tiene el modelo
- Las flechas muestran como la información se transmite de izquierda a derecha.
- Los enlaces entre dos neuronas representan las conexiones sinápticas entre ellas.

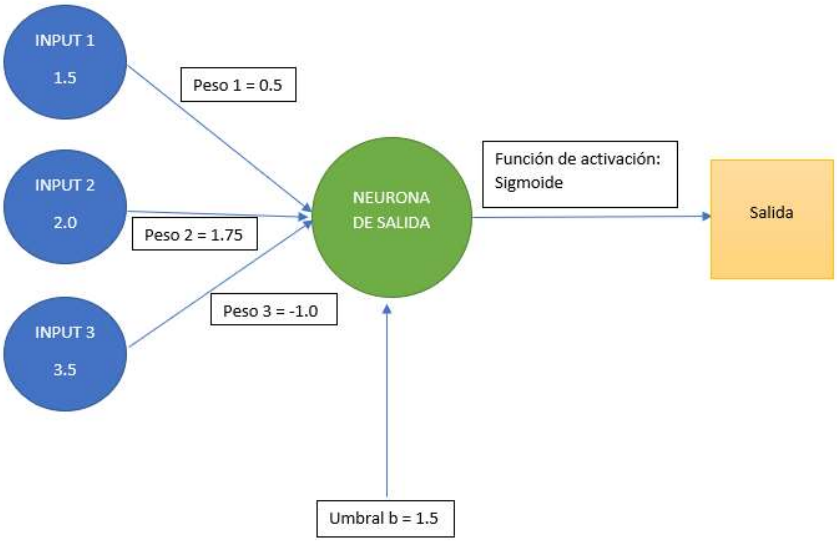


Ilustración 3: Ejemplo de perceptrón simple con valores

En este caso tenemos un perceptrón simple de una sola neurona, ya que se trataría de un modelo de clasificación binomial, es decir, de dos clases. Cuando son solo dos clases con una neurona nos basta, ya que la salida de la función activación sigmoide nos da valores comprendidos entre o y 1. Si dicho valor es mayor o igual a 0.5, la entrada será clasificada como 1. Por el contrario, si el valor es menor, el resultado será clasificado como 0. Podemos ver las similitudes entre este esquema de la neurona y el presentado en el *paper* donde se describía el modelo original, mostrado en la sección anterior.

2.2.2 Forward Propagation

El algoritmo que propaga la información recibida por la red neuronal mediante la entrada a través de la propia red neuronal se denomina *forward propagation.* Su funcionamiento es muy simple, como hemos explicado antes, la salida de dicha neurona se computaría multiplicando los pesos por las entradas, sumándole el valor del umbral y aplicándole la función de activación.

En este ejemplo:

- Lo primero es multiplicar los pesos por el valor de las entradas:

$$Input1 * W1 + Input2 * W2 + Input3 * W3 = \ 1.5 * 0.5 + 2 * 1.75 + (-1.0) * 3.5 = 0.75$$

- Hay que añadirle al resultado obtenido el valor del umbral:

$$Resul + Umbral = 0.75 + 1.5 = 2.25$$

- Por último, hay que pasarle el valor obtenido a la función de activación, en este caso sigmoide.

$$Sigmoid(7.75) = \frac{1}{1 + \ e^{-2.25}} = 0.90$$

Aquí concluiría el algoritmo de *forward propagation* para esta neurona, el resultado obtenido es 0.9 por lo que el modelo clasificaría la entrada como clase 1.

Por simplicidad en la explicación hemos considerado un modelo con dos clases. Si el modelo tuviera que clasificar entre más de dos clases, tendríamos más neuronas de salida, cada una obtendría su salida de una manera similar a como obtiene esta neurona su salida, pero cambiando su función de activación. La función de activación usada para la clasificación en modelos de más de dos clases se llama *softmax*.

2.2.3 Función pérdida

Los pesos se inicializan normalmente de manera aleatoria. Durante la llamada fase de entrenamiento, se intenta llegar a una configuración en los pesos que minimice la llamada "Función de pérdida" del modelo.

Como se puede deducir por el nombre, el valor de esta función es una métrica acerca de lo mal que está actuando el modelo sobre el conjunto de datos en el que se está probando. En nuestro caso, al tratarse de clasificación binaria, la función de pérdida sería la llamada "*binary cross-entropy":*

$$BCE(X) = -\frac{1}{N}\sum yi * \log(xhat\ i) + (1 - yi) * \log(1 - xhat\ i)$$

Siendo yi la clase real de la entrada introducida a la red neuronal i, y xhat i la saluda producida por nuestra red. Como podemos ver en la ecuación, para calcular el valor de dicha función se suman los valores de cada entrada y se hace la media. Imaginemos que el valor real de la entrada que le hemos dado a nuestra red es 0. El valor de dicha función para nuestra red sería:

$$-(0 * \log(0.9) + (1 - 0) * \log(1 - 0.9)) = 1$$

2.2.4 Back Propagation

Los pesos se suelen inicializar aleatoriamente. Durante la llamada fase de entrenamiento, se busca llegar a la configuración óptima de los pesos respecto a los datos de nuestro problema. Esta solución óptima es la que minimiza el valor de la función de pérdida. Aunque la función de pérdida escrita así parece que depende del valor de la salida producida por la red $Xhat$, y el valor real de la entrada y, lo cierto es la salida depende de 4 parámetros más: Los tres pesos y el valor del umbral.

Aunque la dimensión de nuestra función de pérdida es de 4 dimensiones, por simplicidad vamos a dibujar una función de tres dimensiones. Imaginemos que prescindimos del valor del umbral y esta es nuestra función de pérdida en tres dimensiones para cualquier combinación de $W1, W2\ y\ W3$:

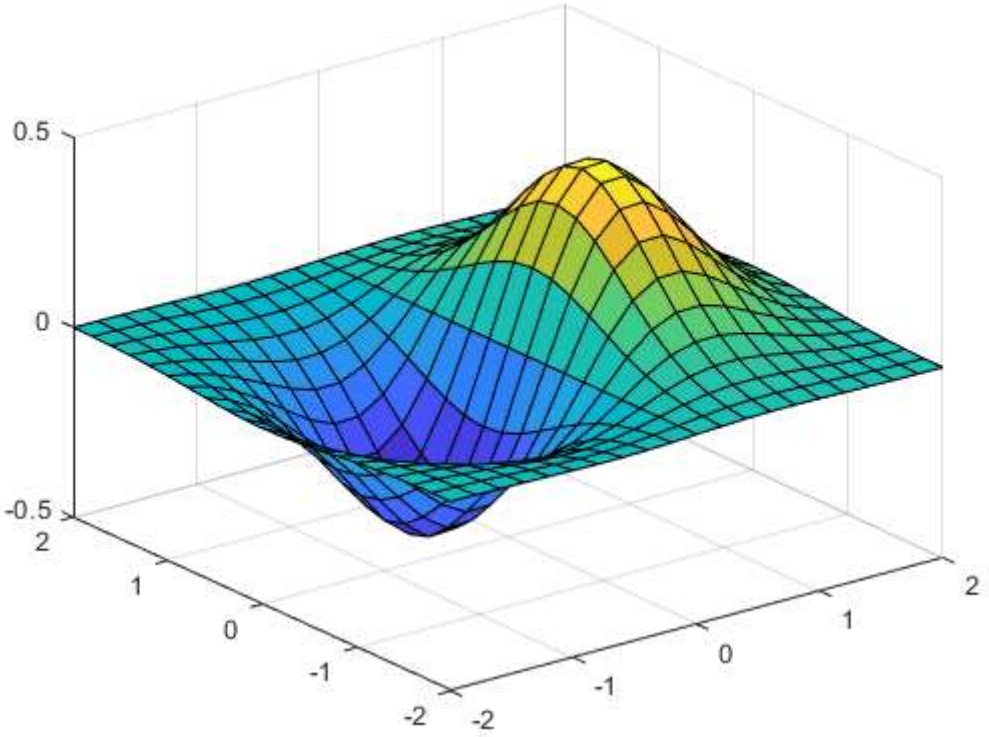


Ilustración 4: Ejemplo de función de tres variables independientes.

Como hemos dicho antes, el objetivo de la fase de entrenamiento es que los pesos alcancen la configuración donde el valor de la función es menor. Para ello utilizamos derivadas direccionales. Calculando la derivada de la función de pérdida respecto a cada peso, y restándole dicho valor al peso que teníamos antes lo que conseguimos es que vayamos descendiendo en la curva de la función de pérdida hasta llegar al mínimo. Recordamos que le valor de la derivada es la pendiente de una función, por lo que calculando dicha derivada estamos haciendo que el valor producto de una combinación de pesos vaya descendiendo por el camino más rápido posible.

Vamos a calcular el valor de la derivada para la actualización del peso $W1$.

- Como hemos dicho, el valor de esta actualización se consigue a partir de la derivada direccional, es decir, tendremos que calcular el valor de la derivada de la función pérdida (BCE) respecto a $W1$. A dicho valor lo llamaremos Ω.

$$\Omega = \frac{\partial \mathrm{BCE}}{\partial \mathrm{W1}}$$

- Para calcular esta derivada, hacemos uso de la regla de la cadena, es decir la derivada de la función de pérdida (BCE)con respecto a $W1$ es igual a la derivada de BCE respecto a la salida de la red $Xhat$ multiplicado por la derivada de esta salida respecto a $W1$.

$$\frac{\partial \mathrm{BCE}}{\partial \mathrm{W1}} = \frac{\partial \mathrm{BCE}}{\partial \mathrm{Xhat}} * \frac{\partial \mathrm{Xhat}}{\partial \mathrm{W1}}$$

- También tenemos que aplicar la regla de la cadena en el segundo producto de la multiplicación anterior, ya que la salida del sumatorio pasa por la función sigmoide. El resultado de la derivada para el peso $W1$ es el siguiente. El desarrollo de la derivada completa puede ser consultado en los apuntes de la asignatura (Calonge & Alonso, 2018).

$$\Omega = \frac{\partial \mathrm{BCE}}{\partial \mathrm{W1}} = \left((1 - yi) * \mathrm{X_{hat}} - \mathrm{y_i} * (1 - \mathrm{X_{hat}})\right) * \mathrm{Input\_1}$$

- Sustituyendo por los valores del ejemplo:

$$\left((1 - 0) * 0.9 - 0 * (1 - 0.9)\right) * 1.5 = 0.9$$

- Una vez calculado el valor de la derivada de cada peso, se le sustrae dicho valor, al valor del peso anterior, multiplicado por una constante llamada *learning rate,* los valores de esta constante suelen estar en el rango (0.1, 0.0001) y su valor determina la velocidad con la que el valor de la función coste desciende por la pendiente Suponiendo *learning rate* igual a 0.1:

$$W1_{new} = W1_{old} - lr * \Omega = 0.5 - 0.1 * 0.9 = 0.41$$

- Si se calculasen las actualizaciones de todos los pesos y la del umbral y se volviese a calcular el valor de la función de pérdida para la misma entrada, se podría comprobar que el valor ha descendido.

En la fase de entrenamiento de la red neuronal repetimos este proceso de manera iterativa hasta que se estabiliza el valor de la función de pérdida. En ese momento asumimos que se ha alcanzado un valor en la combinación de pesos muy cercano al óptimo y nuestra red ya debería ser capaz de realizar la tarea para la que ha sido entrenada.

2.3 Perceptrón Multicapa

El siguiente nivel de complejidad es el perceptrón multicapa. Una de las limitaciones más evidentes del Perceptrón Simple es que no deja de ser una Regresión Logística, por lo tanto, cuando el conjunto de clases no resulta linealmente separable, no se va a poder realizar una separación perfecta.

La solución para conseguir que el modelo sea capaz de separar las clases es introducir capas intermedias entre la entrada y la salida. Estas capas intermedias se denominan “capas ocultas” y se representan con círculos grises en la Ilustración 6. La suma de estas capas, añadida a la función de activación de cada neurona tiene como resultado que el modelo es capaz de separar clases que no son linealmente separables. En el momento en el que introducimos una de estas capas ocultas en nuestra red neuronal, a esta se la pasa a llamar *Deep neural network.*

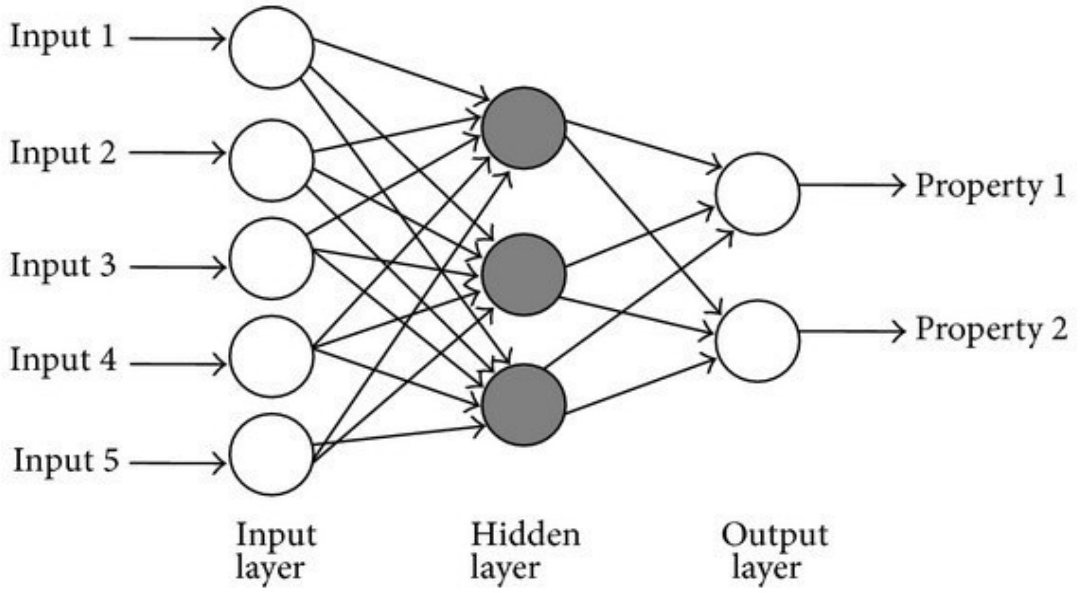


Ilustración 5: Esquema de Perceptrón Multicapa. Fuente: https://medium.com/pankajmathur/a-simple-multilayer-perceptron-with-tensorflow-3effe7bf3466

Como se puede intuir del último párrafo, el añadir capas ocultas o neuronas a cada capa propicia que el error de clasificación disminuya, pero como en cualquier otro modelo estadístico, esto suele significar que se está incurriendo en *overfitting*, es decir que nuestro modelo se adapta tanto a los datos con los que ha sido entrenado, que pierde cualquier tipo de capacidad de generalización de los datos de entrenamiento sobre el problema que estamos tratando, y por lo tanto pierde cualquier tipo de capacidad de predicción.

No existe una fórmula mágica que nos diga cuantas capas y cuantas neuronas por capa poner. Si bien la experiencia de haber resuelto problemas con redes neuronales nos puede ayudar a la hora de medir la complejidad del problema a tratar y estimar el número de capas ocultas y el número de neuronas por capa, llegaremos a la solución óptima probando distintas configuraciones y midiendo el comportamiento del modelo con la técnica de validación *Croos-Validation*, que se basa en dividir en conjunto de datos en datos de entrenamiento y datos de prueba. Hay varios tipos de *Croos-Validation* como pueden ser *K-fold Cross-Validation, o Leave-One-Out Cross-Validation*

El método de entrenamiento que utiliza el perceptrón multicapa durante su fase de entrenamiento es el mismo que usa el perceptrón simple, utiliza el algoritmo *back propagation*. Es decir, calcula el valor de la derivada de la función de pérdida con respecto al parámetro que se quiera actualizar, se multiplica ese valor por la constante *learning rate,* y se le sustrae dicho valor al valor antiguo del parámetro, hasta conseguir que el valor de la función pérdida se estabilice. Entonces supondremos que hemos alcanzado la combinación de valores para los parámetros óptima.

2.4 Redes Neuronales para la comprensión de imágenes.

2.4.1 Procesamiento de imágenes

El primer punto para entender este apartado es comprender como lee las imágenes un lenguaje de programación. Cada píxel de una imagen tiene tres valores, cada valor representa la intensidad de dicho píxel en los colores rojo, verde y azul. El valor de cada intensidad va desde 0 hasta 255. La mayoría de los lenguajes de programación lee las imágenes como una matriz tridimensional que tiene la siguiente forma:

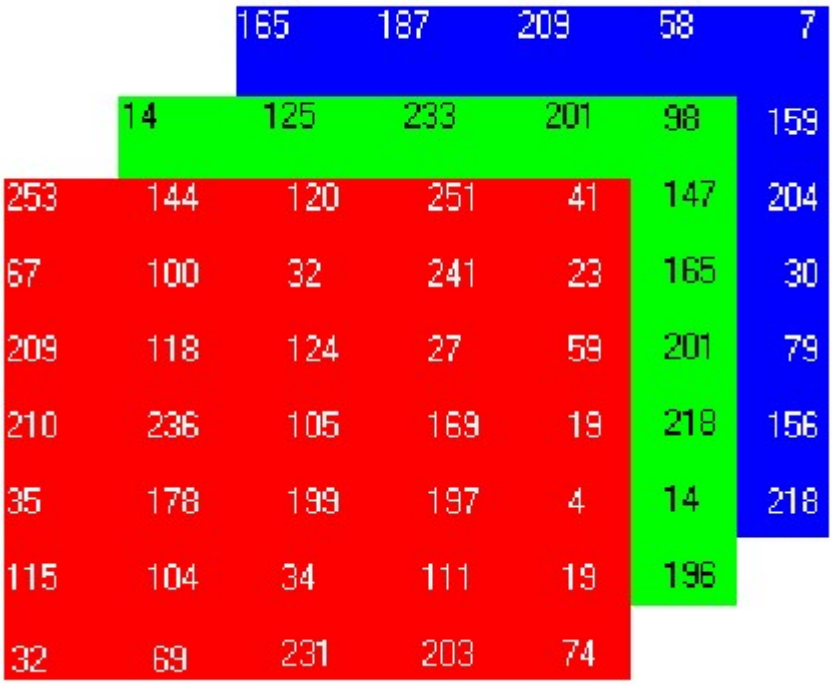


Ilustración 6: Representación de una imagen en forma de matriz tridimensional.

Las dimensiones de esta matriz son (número de pixeles de anchura x número de pixeles de altura x 3). En el caso de la Ilustración 6, la imagen tendría una dimensión de 7x6x3. Las imágenes de una resolución media tienen 224x224 píxeles, por lo que una imagen no tiene menos de 150.528 valores. Si queremos utilizar esta imagen como input en un perceptrón multicapa, tendríamos que tener una neurona de entrada por cada valor dentro de la matriz, es decir, 150.528 neuronas de entrada. Aunque el número es demasiado grande no es el principal inconveniente para no usar el perceptrón multicapa como arquitectura en el procesamiento de imágenes y desarrollar modelos que por características hacen esta tarea de un modo mucho más eficiente.

2.4.2. Limitaciones del perceptrón multicapa en el procesamiento de imágenes

Con la potencia computacional que existe hoy en día podemos pensar que 150.028 neuronas de entrada no son un impedimento para que nuestra red obtenga resultados satisfactorios, Y

estamos en lo cierto. Es por la propia naturaleza de la imagen por la que no podemos usar un perceptrón multicapa en el procesamiento inteligente de imágenes.

Imaginemos que tenemos un perceptrón multicapa donde cada neurona de entrada representa el valor de una intensidad de un pixel, para todas las tres intensidades distintas de todos los pixeles de una imagen. Queremos entrenar un modelo que clasifique si en una imagen hay perros o no los hay, y para entrenar el modelo utilizamos un conjunto de imágenes en las que los perros, en las imágenes en las que los hay, aparecen en la parte inferior de la imagen. Después de alcanzar una combinación de valores buena para los pesos, le damos al modelo una imagen en la que el perro aparece en la parte superior. El modelo fallará.

La razón por la que falla es que cada entrada en el modelo del perceptrón multicapa tiene un valor por si sola. Por ejemplo, en una herramienta de análisis de jugadores de baloncesto, la primera entrada puede representar los puntos encestados por partido, la segunda el número de minutos jugados… etc. No tiene ningún sentido extender esto a los píxeles, ya que la posición del píxel dentro de la foto no importa, no nos interesa si el perro está arriba o debajo de la imagen, lo que nos interesa es que está. Para más ejemplos consultar (National Research University Higher School of Economics).

2.4.3 ¿Qué tareas deberá llevar a cabo una computadora para procesar imágenes?

Antes de ponernos a implementar un algoritmo tenemos que tener claro que tarea debe realizar ese algoritmo. Esto es algo que puede parecer trivial en la mayoría de los casos, pero no en el campo de Computer Vision. Para resolver esta primera pregunta, los desarrolladores de algoritmos capaces de procesar imágenes de manera inteligente se ayudaron de una serie de trabajos ya publicados. Voy a hacer una mención de algunos de los trabajos que fueron clave a la hora de definir dichas tareas

La neurociencia dice que el cerebro humano reconoce objetos por composición. Primero se detectan los límites y bordes de las partes de los objetos que forman parte del estímulo visual, por composición de dichos bordes detectamos partes de los objetos, y por composición de dichas partes llegamos al objeto en sí (Marr, 1982).

Otra conclusión que tuvo influencia a la hora de formular el modelo matemático que usa el Deep Learning a la hora de procesar imágenes es que una persona puede ser representada mediante partes críticas de la misma y la distancia entre dichas partes (Fischler & Elschlager, 1973).

Otra aportación teórica determinante fue teoría la explicación de que los objetos representados en las imágenes son muy susceptibles de recibir cambios debido a, por ejemplo, el ángulo en el que se observan o la luz. Sin embargo, hay características que resultan invariantes inherentes a cada objeto. Se tendría que reconocer dichas características y enlazarlas con el objeto al que corresponden (Lowe, 1999).

Aunque todos los estudios anteriores son importantes a la hora de entender como el cerebro humano procesa imágenes, son demasiado ambiciosos teniendo en cuenta los medios con los que se contaba en la época en la que fueron formulados. La tarea del entendimiento artificial de una imagen era demasiado ambiciosa. Agrupar los píxeles de una imagen de manera que

cada grupo represente un objeto es una tarea más simple por la que empezar a trabajar. De esta idea nace la tarea *Image Segmentation* (Shi & Malik, 2000).

Del reconocimiento de alguna de las tareas que tendría que resolver el modelo, después de haber analizado desde un punto de vista neurocientífico los fundamentos del entendimiento humano a los estímulos visuales nace el modelo matemático que hoy utilizamos en Deep Learning para el procesamiento de imágenes. Las redes neuronales convolucionales.

2.4.4 Redes Neuronales Convolucionales.

Las redes neuronales (LeCun, Bengio, & Haffner, 1998) es un modelo matemático capaz de resolver la mayoría de las tareas expuestas en el apartado anterior. La primera vez que se aplicó este modelo en un *framework* de *Deep Learning* mejoró cualquier resultado en clasificación de imágenes obtenidos hasta la época (Krizhevsky, Sutskever, & Hinton, 2012),. En la actualidad, las redes neuronales convolucionales siguen siendo el modelo más eficaz a la hora de procesar imágenes de manera inteligente.

Su nombre viene dado por "Convolution of two signals". "En matemáticas, y en particular análisis funcional, una convolución es un operador matemático que transforma dos funciones *f* y *g* en una tercera función que en cierto sentido representa la magnitud en la que se superponen *f* y una versión trasladada e invertida de *g*. Una convolución es un tipo muy general de media móvil, como se puede observar si una de las funciones se toma como la función característica de un intervalo" (Wikipedia, 2019)

Cualquier arquitectura de redes neuronales convolucionales tiene dos capas importantes:

2.4.4.1 Capa Convolucional

Uno de los problemas propios de utilizar el perceptrón multicapa a la hora de tratar con imágenes es que teníamos que "aplanar" la estructura matricial tridimensional en un vector y utilizarlo como input de la red neuronal. Sin embargo, la capa convolucional preserva la estructura espacial.

Para preservar dicha estructura, la capa convolucional está formada por filtros. Los filtros son estructuras tridimensionales que tienen como dimensiones (altura del filtro x anchura del filtro x profundidad de la imagen). Los filtros suelen ser cuadrados por lo que la altura y la anchura suele ser la misma.

Los valores de esta matriz que forma el filtro son los pesos. El funcionamiento del filtro es ir cogiendo porciones de la imagen que corresponden con la altura y a anchura del filtro, y multiplicar dichos valores por su peso correspondiente, sumando todas las multiplicaciones. El resultado final de aplicar un filtro a una porción de la imagen es un número.

La Tabla 2 es un ejemplo de cómo un filtro produce una salida, para mayor simplicidad he supuesto que la profundidad es 1, por lo que la matriz resulta bidimensional. Normalmente la profundidad es mayor, pero el resultado no varía.

Entrada	Filtro	Resultado
1 0 1 1 -1 1 1 -1 -1	0 -0.5 1 1 3 0 4 0 -2	$1*0+0*(-0.5)+1*1+1*1+(-1)*3+1*0+1*4+(-1)*0+(-1)*(-2)=5$

Tabla 2: Ejemplo de computación matemática de la correlación entre un filtro y una ventana de la imagen entrada en una capa convolucional

En cada capa de convolución, cada filtro recorre la matriz de izquierda a derecha y de arriba hacia abajo produciendo salidas.

El número de celdas que se mueve después de cada multiplicación es introducido como parámetro con el nombre "*stride*", por lo que es configurable. En la siguiente tabla veremos cómo influye dicho valor en el comportamiento del modelo.

Para el ejemplo mostrado en la tabla 3, he supuesto, por simplicidad, un tamaño de la imagen de 7x7 bidimensional, y un tamaño de filtro de 3 x 3. Como la imagen inicial es bidimensional, el filtro también lo es.

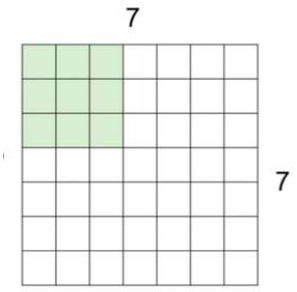
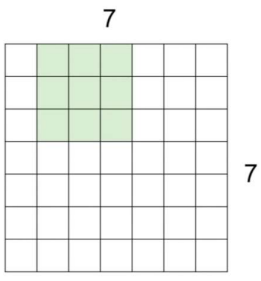
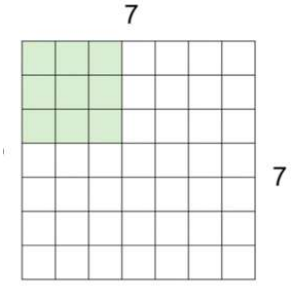
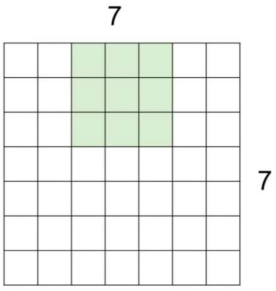
Valor Stride	Primer Paso	Segundo Paso
1		
2		

Tabla 3: Primeras iteraciones de una capa convolucional dependiendo de su valor de stride

Como he explicado antes, cada paso del filtro por la imagen produce un valor, que se guarda dentro de una matriz que formal la salida de la capa convolucional.

Como es lógico pensar, el tamaño la matriz de salida depende del valor del *stride*. Para el ejemplo anterior, con el valor del *stride* = 1, el tamaño de la salida será de 5 x 5. Con el valor *stride* = 2 el tamaño será 3 x 3. El tamaño de la matriz de salida viene dado por la siguiente fórmula:

$$tam\ salida = \frac{N - F}{stride} + 1$$

Donde *N* es el tamaño inicial, *F* es el tamaño del filtro y *stride* es el valor del *stride*.

Como hemos notado, el tamaño de la salida siempre será menor que el de la entrada, y eso no es conveniente en las capas convolucionales, ya que no se quiere que el tamaño de la imagen descienda demasiado rápido. Para paliar esto, se suele hacer antes del paso de los filtros una operación llamada *padding*, que consiste en añadir tantos marcos de 0's a la imagen entrante como sean necesarios para que el tamaño final de la salida sea el mismo que el de la entrada.

El valor que se añade en el marco por convenio suele ser el de 0, ya que por pruebas realizadas es el que mejor funciona, aunque se puede elegir cualquier valor, siempre y cuando sea el mismo para todo el marco.

Antes de padding

3	4	6	5	1	3
5	3	2	4	3	2
5	4	3	3	2	6
1	1	2	5	3	4
2	3	3	4	1	2
3	3	2	4	2	4

Después de padding

0	0	0	0	0	0	0	0
0	3	4	6	5	1	3	0
0	5	3	2	4	3	2	0
0	5	4	3	3	2	6	0
0	1	1	2	5	3	4	0
0	2	3	3	4	1	2	0
0	3	3	2	4	2	4	0
0	0	0	0	0	0	0	0

Ilustración 7: Comparativa entre el antes y el después de la aplicación del algoritmo de padding sobre una matriz

Como he mencionado antes, una capa tiene muchos filtros, y cada filtro produce una matriz bidimensional. Los resultados del paso de los diferentes filtros por la matriz se apilan, formando una matriz tridimensional de dimensiones (tam salida x tam salida x nº filtros). Como podemos observar, la salida producida por una capa convolucional es una matriz tridimensional, al igual que la entrada. Por lo que el propósito inicial de preservar la estructura espacial se consigue.

En el siguiente ejemplo veremos las diferentes salidas dependiendo del número de filtros que tengamos en la capa.

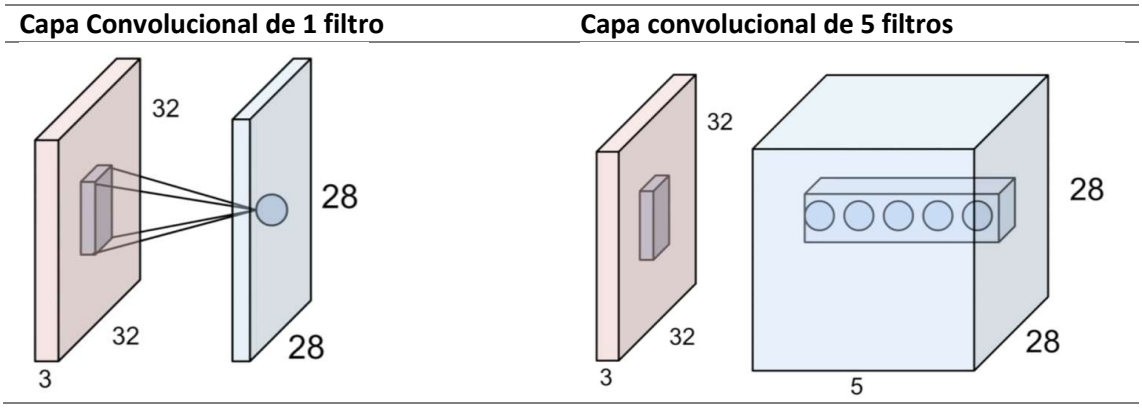


Ilustración 8: Comparativa entre una capa convolucional de 1 filtro con una de 5 filtros

Una vez que tenemos claro el funcionamiento de una capa convolucional y sus filtros, podemos hacernos la siguiente pregunta. ¿Qué significan los filtros y el resultado que producen? ¿De qué depende el número de filtros?

Un filtro configura sus pesos en función de la característica que está buscando en la foto. Se suele distinguir entre 3 distintos niveles de características dependiendo del nivel en el que nos encontremos de la red neuronal. Características de bajo, medio y alto nivel. Como ejemplo para ilustrarlo imaginemos que nuestra red está diseñada para reconocimiento facial. Características de bajo nivel serán bordes de diferentes formas y direcciones, de medio nivel ojos, bocas, narices. Características de alto nivel serán distintos trozos de la cara.

Cada filtro busca la correlación entre la característica que le corresponde y el trozo de la imagen que se le da. Cuanto más alto sea dicho valor, más se parecerá esa parte de la foto a lo que está buscando. Si nos fijamos en el anterior ejemplo, en la capa convolucional de 5 filtros, cada filtro está mirando a la misma zona de la imagen, pero está buscando cosas distintas, por lo que las 5 salidas serán distintas.

El número de filtros depende del número de características distintas que estemos buscando. Normalmente, cuanto más profundo sea el nivel en el que se encuentra la capa, más número de características querremos diferenciar, por lo que mayor será el número de filtros y en consecuente, mayor será la profundidad de la matriz de salida. Para ver la expliación original sobre la que me he basado para desarrollar este contenido (National Research University Higher School of Economics):

2.4.4.2 Capa Pooling

La segunda capa propia de las arquitecturas convolucionales es la capa de *pooling*.

Después de haber explicado lo que era el *padding,* es decir, enmarcar la entrada con ceros para no perder tamaño, y haber añadido que conforme avanza la imagen por la red, la profundidad aumenta, alguno se preguntará que si la entrada aumenta en tamaño siempre conforme avanza por las capas, cuando lo normal en las redes es lo contrario. La respuesta es que no, y es precisamente la labor de esta capa.

En este esquema podemos ver el comportamiento de la capa de *pooling*:

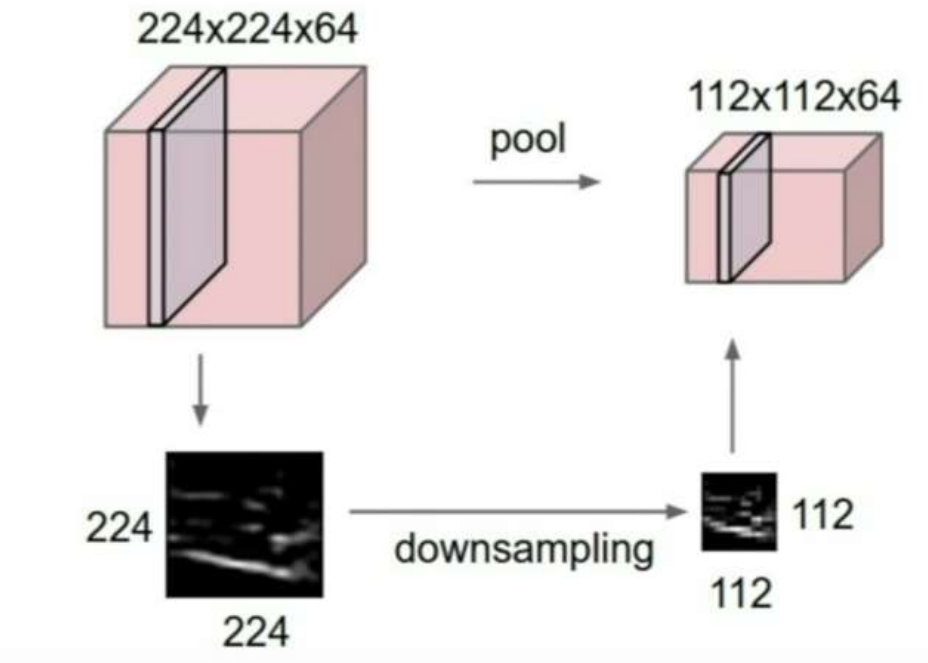


Ilustración 9: Descripción gráfica del algoritmo de pooling con comparativa en cuanto a la pérdida de información en la imagen

Como bien indica su nombre en inglés, *pool* significa agrupar por lo que capa de *pooling* será una capa de agrupamiento, es decir, de reducir el tamaño de la entrada. ¿Cómo reducimos el tamaño de la entrada? Lógicamente intentaremos reducir el tamaño de la entrada manteniendo la mayor cantidad de información posible. Para ello voy a comentar dos opciones:

- Average Pooling:

Consiste en coger una matriz de nxn de la entrada y devolver el valor medio de los 4 números. El valor de n dependerá de cuanto queramos reducir el tamaño. Lo normal es querer reducirlo a la mitad, para ello cogeremos matrices de 2x2. Si queremos reducir el tamaño a un tercio, cogeremos matrices de 3x3 y así sucesivamente.

- Max Pooling:

Aunque nos pueda parecer que la opción más sensata a la hora de no perder información es devolver el valor medio, lo cierto es que esta opción tiene mejores resultados. Esta capa lo que devuelve es el valor máximo de los valores que pertenecen a la matriz. Aunque desde un punto de vista estadístico esto pueda no parecer mucho sentido, si pensamos en lo que significan estos valores, que son la correlación que tiene una parte de la imagen con un filtro, nos interesará quedarnos con el mayor de los valores, es decir, la parte más significativa de la imagen.

En el siguiente ejemplo podemos ver el uso de la capa Max Pooling:

Antes de Max Pooling	Después de Max Pooling

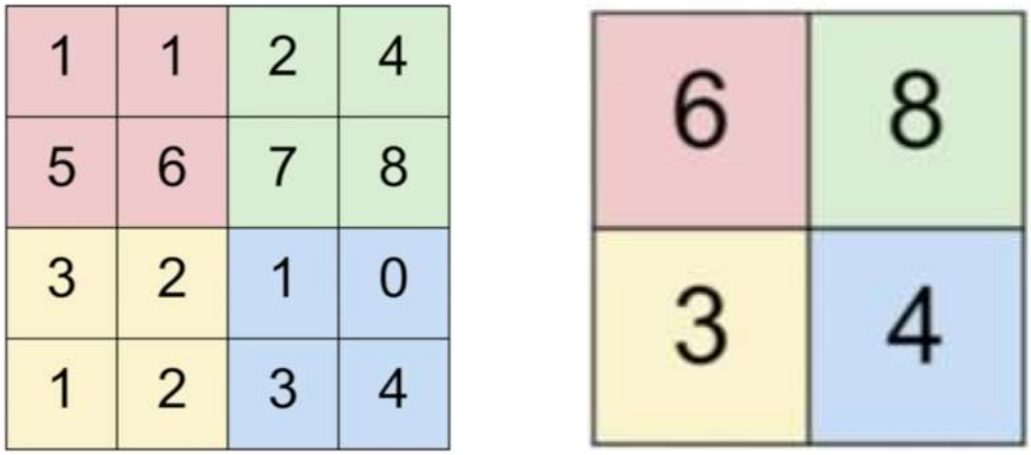


Ilustración 10: Comparación de una matriz antes y después de computar el algoritmo de MaxPooling sobre ella

Nos podemos dar cuenta de que el resultado de esta capa en cuanto a modificación del tamaño de la entrada es opuesto a la capa de convolución. Mientras que esta última, siempre y cuando apliquemos la operación de *padding,* solo modifica la profundidad de la entrada, la capa de *pooling* modifica la altura y la anchura de la matriz, manteniendo la profundidad.

2.4.5 Ejemplo de red neuronal convolucional. AlexNet

Para acabar este capítulo vamos a analizar esta arquitectura que ya hemos mencionado antes. AlexNet fue la primera arquitectura neuronal que usaba el modelo de capas convolucionales y mejoró notablemente los resultados obtenidos en un conjunto de 1000 grupos de imágenes, y la labor del modelo era la clasificación de cada imagen en su grupo correspondiente.

Como es de suponer, la red neuronal cuenta con los tres tipos de capas que hemos explicado hasta el momento: capas convolucionales, capas de *pooling* y perceptrón multicapa.

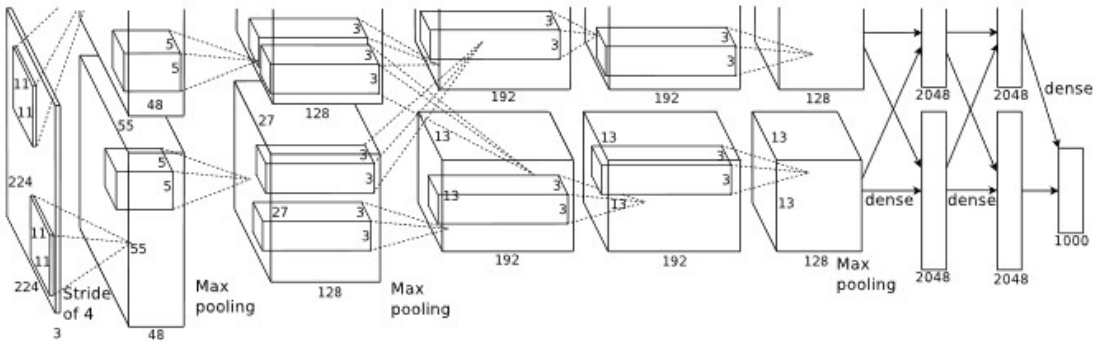


Ilustración 11: Esquema de la red neuronal AlexNet obtenida del paper original (Krizhevsky, Sutskever, & Hinton, 2012)

En el esquema de arriba se muestran las capas que conforman la red neuronal AlexNet, de izquierda a derecha:

- La primera figura tridimensional es la imagen de entrada, que tiene dimensiones 224x224x3. Como ya explicamos en el apartado 3.4.1 las imágenes tienen esta estructura.
- La primera capa por la que pasa esta imagen es una capa convolucional de 48 filtros de dimensión 11x11x3. Recordamos que la profundidad de los filtros tiene que ser igual a la de la entrada y además sabemos que tenemos 48 filtros porque la profundidad de la salida es de 48. El valor de *stride* de esta capa es de 4, sin *padding*, por lo que si aplicamos la fórmula del apartado 3.4.4.1: $tam\ salida = \frac{224-11}{4} + 1 \approx 55$.
- Lo siguiente que tenemos es una capa de Max Pooling de tamaño 2x2 por lo que reducimos tanto la altura como la anchura a la mitad, reduciendo el tamaño de la entrada de 55 a27 píxeles.
- Después tenemos una capa convolucional de 128 filtros de tamaño 3x3x48. Incluida esta capa, a partir de ahora tenemos capas de convolución con un valor de *stride* igual 1 y con operación de *padding*.
- Volvemos a tener una operación de Max Pooling 2x2 para reducir la altura y la anchura de la entrada de 27 pixeles a 13.
- La siguiente capa es convolucional de 192 filtros de tamaño 3x3x128.
- Seguida de esta última capa tenemos otra capa convolucional de 192 filtros de tamaño 3x3x192.
- La última capa convolucional de la red es una capa de 128 filtros de tamaño 3x3x192.
- El output producido por esta última capa es una matriz 13x13x128. Las siguientes capas forman un perceptrón multicapa. Este tipo de arquitectura necesita un input en forma de vector, por lo que hay que “aplanar” la matriz, es decir, convertirla a vector unidimensional de tamaño 21632.
- El perceptrón multicapa que va después de la parte convolucional de la red lo conforman una entrada de tamaño 21632, dos capas ocultas de 2048 neuronas y una capa de salida de tamaño 1000 debido a las 1000 clases distintas entre las que tiene que diferenciar el modelo entrenado.

Capítulo 3.

Arquitecturas Convolucionales para nuestro problema.

3.1 Introducción de arquitecturas convolucionales

Después del éxito obtenido por la red AlexNet al juntar capas convolucionales y capas de *maxpooling*, muchos investigadores del campo de Computer Vision empezaron a experimentar haciendo combinaciones de estas dos capas buscando una arquitectura óptima que mejorase los registros obtenidos por esta primera red. Todas estas redes tienen como principal función la de clasificación. Es por ello que el comportamiento de todos estos modelos siempre está medido con la base de datos ImageNet de 1000 clases diferentes.

ImageNet (Standford University; Princeton University, 2006) es una base de datos abierta que cuenta con más de 14 millones de imágenes para 1000 clases distintas. Debido a la gran cantidad de imágenes etiquetadas y la calidad de las mismas, es la base de datos sobre la que se prueba la eficacia de las arquitecturas convolucionales que tienen la tarea de clasificación.

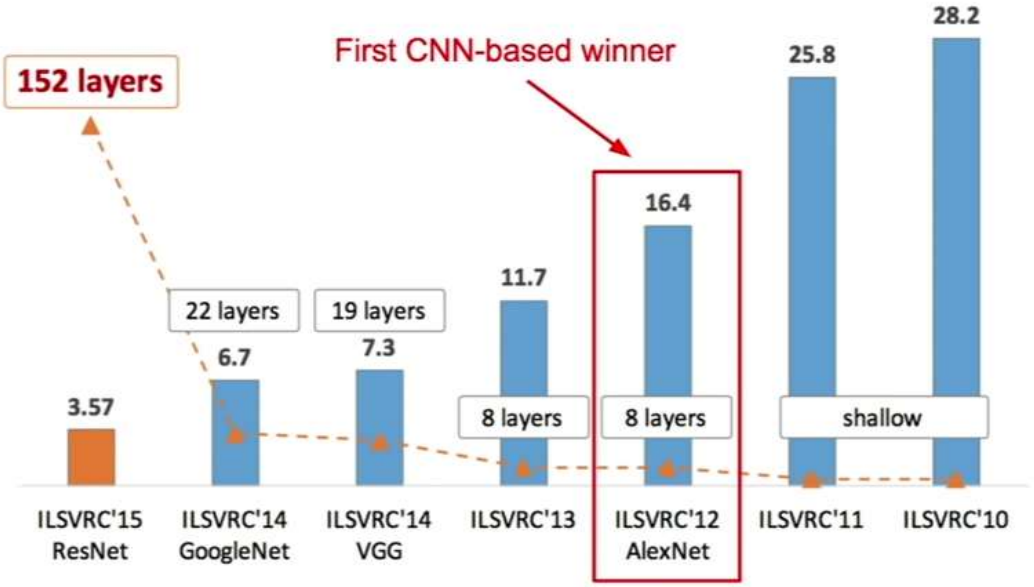


Ilustración 12: Comparativa del rendimiento de diferentes arquitecturas ganadoras del concurso ImageNet (Stanford University School of Engineering, 2017)

En este gráfico vemos las mejores arquitecturas para la tarea de clasificación sobre el archivo de imágenes ImageNet. Vemos como aparecieron nuevos modelos que mejoran el comportamiento de la primera red convolucional.

El objetivo de este capítulo es explicar dos arquitecturas diferentes que obtuvieron muy buenos resultados en la base de datos mencionada (VGG y ResNet), describir la base de datos que hemos utilizado para nuestro problema y hablar del Transfer Learning en Deep Learning. Después

entrenaremos las dos arquitecturas antes descritas con nuestra base de datos y veremos cuál de las dos produce mejores resultados.

3.2 VGG16

Como podemos ver en el gráfico anterior, VGG (Simonyan & Zisserma, 2014) ganó la competición de ImageNet en 2014, el año en el que fue presentada. Al igual que AlexNet, antes mencionada, está formada a base de una combinación de los tres componentes de redes neuronales convolucionales antes mencionados: capas convolucionales, capas de *maxpooling* y al final un perceptrón multicapa para las tareas de clasificación.

En el papel presentado por los dos investigadores desarrolladores de esta arquitectura, Karen Simonyan y Andrew Zisserman, se presentan hasta 5 configuraciones diferentes:

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Ilustración 13:Distintas configuraciones de la arquitectura VGG, imagen extraída del paper original (Simonyan & Zisserma, 2014)

En esta tabla presentada en el propio *paper*, están descritas las 5 configuraciones distintas por columnas. Para nuestras pruebas vamos a utilizar la configuración D (Las distintas configuraciones vienen presentadas en columnas). Si prestamos atención a dicha configuración, vemos que tenemos las 16 capas de pesos que describen el nombre de VGG16, divididas en cuatro bloques de 2,2,3 y 3 capas en cada uno respectivamente, separados por una capa de *pooling.* Los filtros que usa cada capa son siempre del mismo tamaño, 3x3 y la entrada al modelo es una imagen de 224x224x3. El número de características extraídas es de 64 para el primer bloque, 128 para el segundo y 512 para el tercero y el cuarto. Finalmente, al igual que en la arquitectura de AlexNet, tenemos un perceptrón multicapa con dos capas ocultas de 4096 neuronas cada una, y una capa de salida de 1000 neuronas debido a las 1000 clases que tiene la base de datos ImageNet.

Para poner en contraposición esta arquitectura con la arquitectura de AlexNet he adjuntado este esquema para hacer una comparación visual rápida:

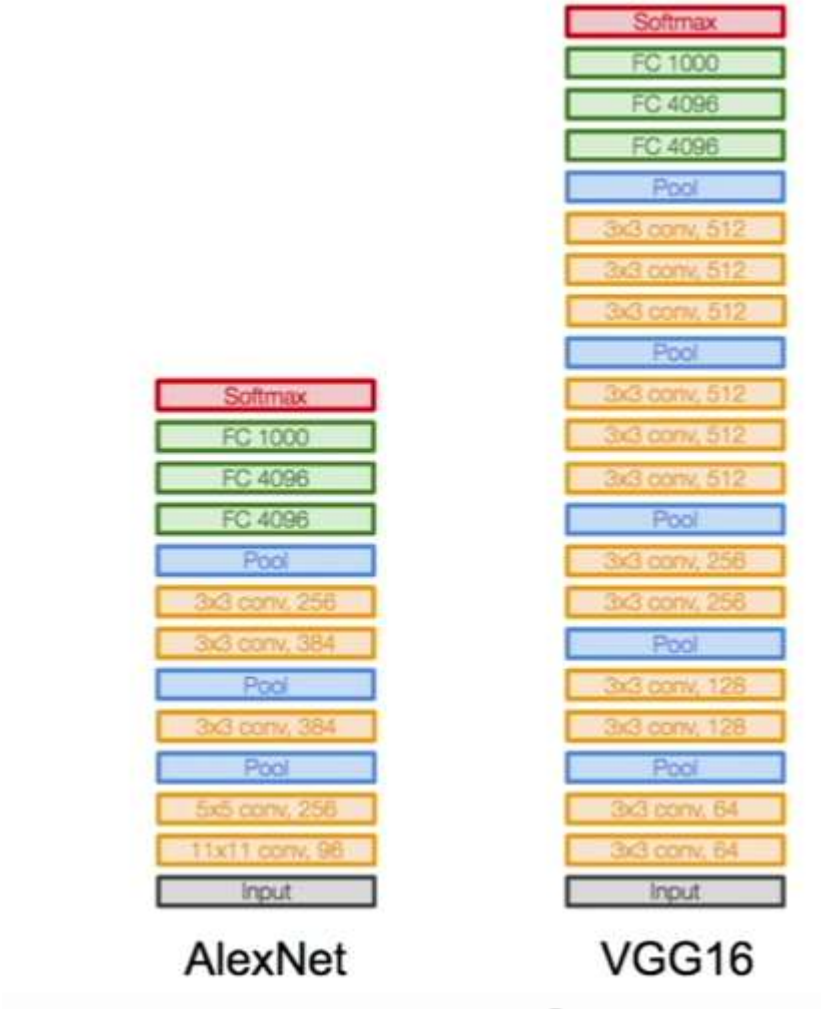


Ilustración 14: Comparativa entre la configuración de AlexNet y Vgg16 (Stanford University School of Engineering, 2017)

Parece que no es más que una ampliación de la anterior, más capas y más parámetros, pero ¿qué tiene de especial?, ¿por qué es novedosa?

Lo primero, vemos que a la base de esta arquitectura son las capas convolucionales 3x3, de hecho, es el único tamaño de filtros que usa. También vemos que entre las dos capas de pooling, tenemos dos o tres capas convolucionales apiladas de este tamaño de filtros. ¿Por qué?

Al explicar cómo funcionaban los filtros de las capas convolucionales, dijimos que para un filtro 3x3, lo que hace el modelo es multiplicar el valor de cada píxel por el peso correspondiente en el filtro y hacer un sumatorio de todo, es decir, estamos reduciendo la información contenida en nueve píxeles (3x3) en un valor. En este caso diremos que el tamaño receptivo del filtro es de 3x3. ¿Qué pasa si apilamos dos filtros 3x3? ¿Cuál será el tamaño del campo receptivo final?

Para visualizar mejor el problema nos ayudamos del siguiente esquema:

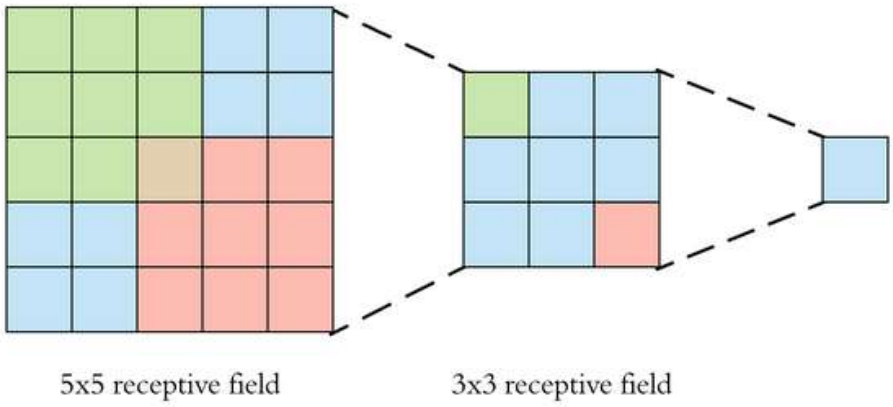


Ilustración 15: Esquema para facilitar la comprensión de los campos receptivos. Fuente: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Como vemos en el esquema, la información guardada en un valor después del paso de la entrada por dos capas convolucionales apiadas de tamaño de filtro 3x3 es de 5x5, es decir, el tamaño del campo receptivo final es de 5x5, cada píxel de la salida tendrá información de 5x5 píxeles de la entrada. Si nos fijamos en las capas altas de la red, vemos que se apilan 3 capas convolucionales del mismo tamaño de filtro. Haciendo uso de la lógica y ayudándonos del esquema anterior podemos inferir que después del apilamiento de tres filtros, el campo receptivo final será de 7x7, es decir, cada píxel de la capa de salida tendrá información de 49 píxeles de la entrada.

Podemos pensar que, si en el caso de que el campo receptivo final sea de 5x5, la potencia del apilamiento de dos capas convolucionales será equivalente a la de un filtro de tamaño 5x5, por lo que puede parecer que no tiene mucho sentido este modelo si por simplicidad podemos usar un filtro de tamaño 5x5. Lo cierto es que no es así.

Al poner una capa convolucional detrás de otra tenemos un efecto similar que al añadir una capa de neuronas normales en un perceptrón multicapa, en vez de añadir dicho número de neuronas a la capa ya existente. Es mucho más potente porque al pasar la salida por la función de activación, rompemos con la linealidad del problema, es decir la solución ya no es una combinación lineal de la entrada, por lo que el modelo es mucho más potente. Esto es lo que pasa también el aplicar dos capas convolucionales consecutivas, el resultado no es una combinación lineal de la entrada y la potencia de dicha parte de la red es mucho mayor.

Esto puede parecer ya suficiente para decantarnos por la solución de las capas convolucionales consecutivas, pero no es la única. Imaginemos que tenemos en una capa 10 filtros de tamaño 5x5 para una entrada con 10 características (profundidad 10). El número de parámetros que tiene esta capa es de $5 * 5 * 10 * 10 = 2500$. Si en vez del uso de un solo filtro de tamaño 5x5 usamos dos filtros consecutivos de tamaño 3x3 el número de parámetros será $2 * 3 * 3 * 10 * 10 = 1800$ parámetros. Cuantos menos parámetros tengamos más rápido se entrenará nuestra red neuronal por lo que es conveniente intentar reducir el número de los mismos al mínimo.

El uso de capas convolucionales consecutivas, además de la combinación de filtros de tamaño del campo receptivo de 5x5 y 7x7, que parece que son los tamaños que mejor generalizan la extracción de características para cualquier imagen dada, hizo que el rendimiento de la red neuronal mejorara, por eso VGG alcanzó mejores resultados que todas las redes usadas con anterioridad.

3.3 ResNet

3.3.1 Introducción a ResNet

La segunda arquitectura en la que vamos a profundizar se llama ResNet (Residual Net) (He, Zhang, Ren, & Sun, 2015). Si nos fijamos en el gráfico localizado en la introducción de este capítulo, es la arquitectura que mejores resultados consiguió en 2015. Como veremos después esta arquitectura no tiene nada que ver con los anteriores modelos que hemos explicado. Lo innovador de esta arquitectura se denominó *"Revolution of Depth",* es decir, revolución en la profundidad, y es debido a que se pasó de tener 22 capas convolucionales como máximo a 152. Pero, ¿por qué tantas capas?

Uno de los dilemas que se encuentra siempre un desarrollador de redes neuronales es qué profundidad darle a su red para un determinado problema, y que implica esta profundidad.

Si tomamos por ejemplo el caso de una regresión polinómica, conforme aumentamos el grado del polinomio, la recta se ajusta cada vez más a nuestros datos, acabando en sobreajuste. ¿Qué pasa cuando en nuestra red neuronal tenemos más profundidad, es decir un mayor número de capas, de la necesaria?

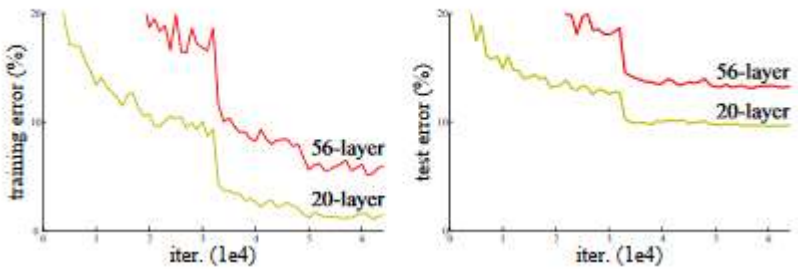


Ilustración 16: Comparativa entre rendimiento de red neuronal de 20 capas con red neuronal de 56 capas (He, Zhang, Ren, & Sun, 2015)

Por analogía a la regresión polinómica podríamos decir que podemos esperar un caso de sobreajuste, que las predicciones para nuestro conjunto de entrenamiento sean perfectas, pero muy malas para nuestro conjunto de test. Sin embargo, como muestra la gráfica de la Ilustración 19 publicada por los desarrolladores de esta arquitectura, las redes neuronales con mucha más profundidad de la necesaria funcionan peor incluso en el conjunto de entrenamiento.

Se cree que esto se debe a problemas de optimización, las redes neuronales profundas son mucho más difíciles de optimizar y por consiguiente se obtienen peores resultados en ambos conjuntos.

Lo ideal para el desarrollador que se encuentra ante la decisión de la elección de profundidad sería que la propia red decidiera que profundidad necesita para resolver el problema al que se enfrenta. Y esto precisamente es lo que intenta hacer la arquitectura ResNet.

La idea fundamental sobre la que se basa ResNet es que si tenemos una red neuronal A de profundidad P y una red neuronal B de profundidad P + 1, B tiene que funcionar por lo menos tan bien como A. La deducción lógica de esto es muy simple, si A tiene la profundidad óptima, la última capa de B tendrá que aprender a devolver lo mismo que le llega, que es la solución óptima, es decir, tendrá que ser una capa de identidad.

Como hemos comprobado que las capas por sí solas no aprender a devolver lo mismo que les llega, por eso el rendimiento empeora, las capas de ResNet no son simplemente un apilamiento de capas convolucionales, sino que esta arquitectura está compuesta por lo que se denominan bloques residuales.

En este esquema podemos ver la diferencia entre un bloque de apilamiento de dos capas convolucionales usado típicamete en la arquitectura VGG y un bloque residual:

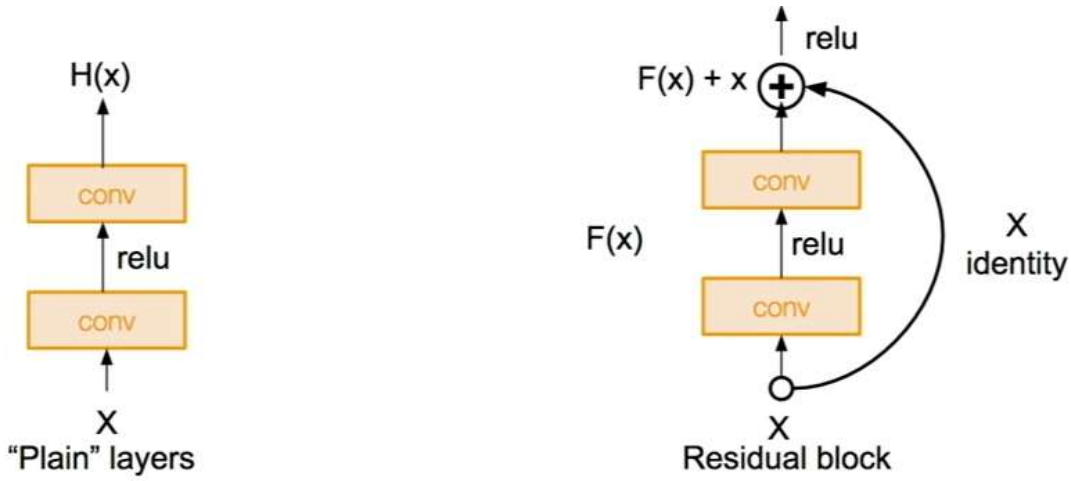


Ilustración 17: Comparativa entre esquema tradicional de capas convolucionales y bloque residual (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: ResNet, 2018)

La principal diferencia es que la entrada que le llega al bloque, además de pasar por las diferentes capas convolucionales, tiene un puente mediante el que se las puede saltar, y de ese modo, que la salida sea igual que la entrada y convertirse así en una capa de identidad.

La motivación de este puente entre la entrada y la salida es ayudar al bloque a aprender la función identidad, ya que como hemos dicho antes, cuando un bloque sobra no es capaz por si solo de aprender a devolver lo mismo que le llega para no estropear el modelo, al insertar este nuevo camino de los bloques, ayudamos a que ahora el bloque si sea capaz de aprender cuando

no es necesario y no modificar la entrada que le llega. La salida de estos bloques residuales será la aplicación de la función de activación ReLu a la suma de los dos caminos.

La idea es que para una capa de una red neuronal que no es necesaria es mucho más fácil aprender que sus pesos tienen que ser cero, de manera que el valor de su salida sea lo que pasa por el puente, es decir, el valor no transformado de la entrada, a modificar los pesos de manera que no modifiquen el valor de la entrada, que es lo que debería aprender una red que no tuviera estos puentes residuales.

La arquitectura descrita por los desarrolladores no solo usa los tres elementos descritos anteriormente para cualquier arquitectura convolucional, es decir: capas convolucionales, capas de *pooling* y MLP para el momento de clasificar la imagen. También usa una capa denominada *BatchNorm*. Para ver la explicación original sobre la que me he basado para desarrollar este contenido consultar (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: ResNet, 2018)

3.3.2 Capa BatchNorm

La capa *BatchNorm* (Ioffe & Szegedy, 2015) está diseñada para mejorar la eficacia de la etapa de entrenamiento de las redes neuronales. La principal función de esta capa es hacer que el intervalo de valores de los hiperparámetros que funcionan a la hora de diseñar nuestra red neuronal sea más amplio, y que el entrenamiento de la red neuronal a través del algoritmo de *back propagation* sea más efectivo.

Explicado en un ejemplo, imaginemos que tenemos la siguiente red neuronal con una capa de entrada, tres capas ocultas y una capa de salida.

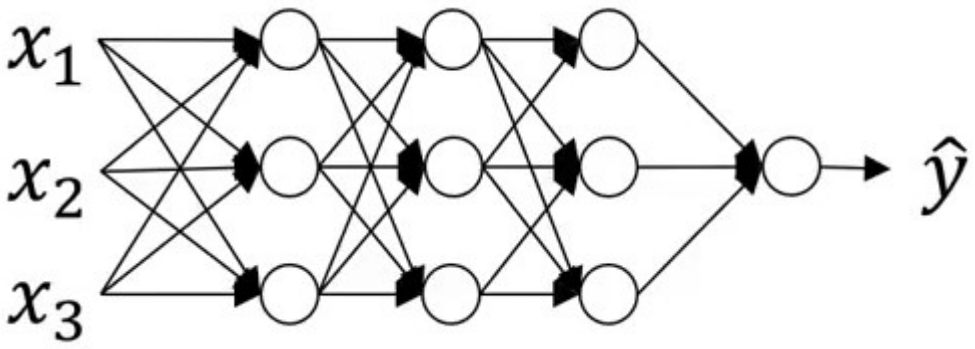


Ilustración 18: Imagen de ayoda para la explicación de la capa BatchNormalization (Deeplearning.ai, 2017)

El uso de una capa de *batchnorm* entre la segunda y la tercera capa oculta provocará que los valores de la segunda capa de salida se normalicen, y por lo tanto el entrenamiento de los parámetros de la tercera capa oculta sea más efectivo, es decir, se llegue antes a los valores óptimos.

¿Cómo se normaliza los valores de la entrada la capa de *batchnorm*? Lo primero es tipificar dichos valores, es decir, suponiendo que los valores sigan una distribución normal, calcular su

media y su varianza, restarle la media a cada valor y dividir el resultado por la varianza. Con esto logramos que los valores de la salida sigan una distribución normal de media 0 y varianza 1. A este valor final lo podemos llamar $Znorm$. Es interesante normalizar los valores de la salida de una capa, pero no tiene sentido forzar a que sigan una distribución $\mathrm{N}(0,1)$ por lo que introduciremos dos nuevos parámetros: α y β para que la propia red aprenda cuales son los parámetros óptimos que deberá seguir la entrada a una nueva capa con la finalidad de maximizar la efectividad de su entrenamiento. La salida de esta capa de *batchnorm* será:

$$\propto * Znorm + \beta$$

Y los valores seguirán una distribución normal de media β y de varianza α.

Normalizar los valores hace que los pesos de las capas más profundas de la red sean más robustos ante los cambios de los pesos de las primeras capas. Esto es importante en nuestra arquitectura porque queremos diferenciar cada parte de la red, y que su entrenamiento no afecte a otras partes, es decir que un cambio en los pesos encargados de la extracción de las características de medio nivel no afecte a los pesos encargados de coger esas características y extraer de ellas características de alto nivel. Para consultar la fuente en la que me he basado para desarrollar este contenido, consultar (Deeplearning.ai, 2017)

3.3.3 Los dos tipos de boques residuales.

Una vez que sabemos lo que es una capa *batchnorm* y la importancia que tiene su aplicación en redes neuronales profundas, como es el caso, podemos estudiar los bloques residuales con mayor profundidad. La arquitectura ResNet utiliza dos tipos de bloques:

- El primer tipo de bloque se denomina *"Identity Block"* y normalmente tiene esta estructura:

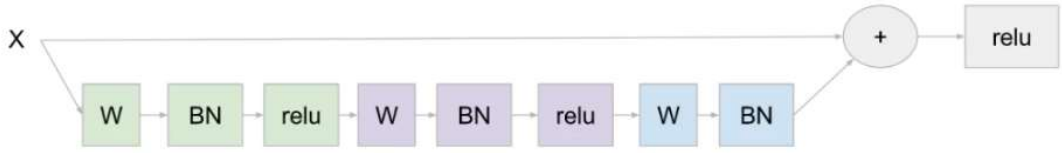


Ilustración 19: Esquema de Identity Block (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: ResNet, 2018)

Como hemos explicado antes, la entrada toma dos caminos, uno por el que no sufre modificaciones (el superior) y otro por el que pasa por dos bloques de capa convolucional + capa *batchnorm* + función de activación ReLu y otro solo compuesto por capa convolucional + capa *batchnorm.* Al final de este bloque residual se suman los valores de cada camino, por ello es importante que cada camino devuelva matrices tridimensionales con las mismas dimensiones (recordamos que los modelos convolucionales trabajan con matrices tridimensionales) y a esa matriz resultante se le aplica la función de activación ReLu. Este bloque es capaz de aprender a hacer lo que veníamos demandando con anterioridad, pasar directamente los valores de entrada a los de salida.

- El segundo tipo de bloque se denomina *"Convolution Block"* y tiene la siguiente estructura:

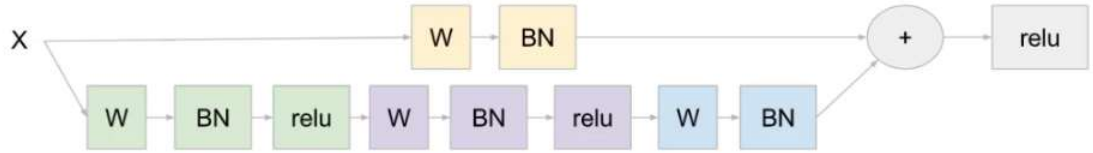


Ilustración 20:Esquema de Convolution Block (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: ResNet, 2018)

La única diferencia entre este bloque y el anterior es que, en éste, el camino corto obliga a pasar a la entrada por una capa convolucional + una capa *batchnorm,* en vez de llegar a la salida directamente.

3.3.4 Arquitectura ResNet completa.

Al igual que en la arquitectura VGG, en esta también hay diferentes configuraciones descritas en el *paper* publicado por los desarrolladores:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				

Ilustración 21: Tabla con las diferentes configuraciones de ResNet (He, Zhang, Ren, & Sun, 2015)

El modelo con el que se presentó al concurso de ImageNet es el de 152 capas, como podíamos ver en el gráfico de la introducción de este capítulo. El modelo que vamos a utilizar nosotros en nuestros experimentos con la base de datos que se adapta a nuestro problema es el de 50 capas.

Este modelo tiene 4 bloques de distintas dimensiones, el primero se repite 3 veces, el segundo 4, el tercero 6 y el cuarto 3. Entre un grupo de bloques y otro hay una capa de *maxpooling*. El

primer bloque de cada grupo, según se describe en el *paper* es un bloque residual convolucional y el resto de identidad. En total el modelo tiene 16 bloques distribuidos de la siguiente manera:

Ilustración 22: Configuración de arquitectura ResNet50 (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: ResNet, 2018)

3.4 Transfer Learning

Como se puede intuir después de haber explicado las dos arquitecturas convolucionales que vamos a aplicar y comparar en nuestro experimento, son redes neuronales muy complejas con un gran número de parámetros. Para que nos hagamos una idea, una capa convolucional de 128 filtros de tamaño 3x3 que recibe 128 características tendrá $128 * 3 * 3 * 128 = 49456$ parámetros. Y esto no es más que una capa estándar dentro de las 16 que tiene la arquitectura VGG16 o las 50 que tiene la arquitectura ResNet50.

El total de parámetros que tiene cada arquitectura viene descrito en ambos *papers:*

- En la arquitectura VGG16 se acumulan más de 138 millones de parámetros, y guardar todos los valores de las salidas de todas las capas gasta más o menos 98 MegaBytes de memoria en el ordenador.
- Aunque es mucho más profunda, la arquitectura ResNet50 tiene 25.6 millones de parámetros. Es bastante sorprendente que, a pesar de ser una red tan profunda, apenas tenga el 18% del número de parámetros que tiene la red VGG.

Después de estos datos en cuanto al número de parámetros podemos pensar que es inviable el uso de estas redes en un ordenador particular, o en cualquier ordenador que no sea una supercomputadora. Imaginemos que tenemos que entrenar esta red, no solo tendríamos problemas en cuanto a tiempo para encontrar la combinación óptima de pesos. ¿Cuánto nos llevaría semejante tarea? ¿semanas? ¿meses?

Como hemos dicho antes, guardar las salidas de todas las capas de la arquitectura VGG para una sola entrada nos consume 94 MB de memoria. Esto provoca que probablemente también tengamos problemas de memoria al entrenar nuestra arquitectura.

El último problema que nos podemos encontrar al querer entrenar nuestra propia arquitectura convolucional compleja es el tamaño de nuestra base de datos. ¿Cuántos ejemplos distintos son necesarios para encontrar los valores óptimos de los pesos de una red neuronal de 138 millones de parámetros? En teoría se recomienda que el conjunto de datos de ejemplos para entrenar sea mayor en número que el número de parámetros a entrenar en la red. ¿Tenemos un conjunto de datos mucho mayor de 138 millones de fotos?

Esta reflexión puede hacernos pensar que estas arquitecturas están muy bien de manera teórica, o para su uso con supercomputadoras en grandes centros de investigación, pero que la

mayoría de los interesados en el campo del Computer Vision jamás van a poder usar. Lo cierto es que esto no es verdad, cualquier persona puede usar estas arquitecturas en su ordenador personal, gracias a un truco denominado *Transfer Learning* (West, 2007).

Como hemos explicado antes, podemos dividir las funciones de toda una red neuronal en dos partes:

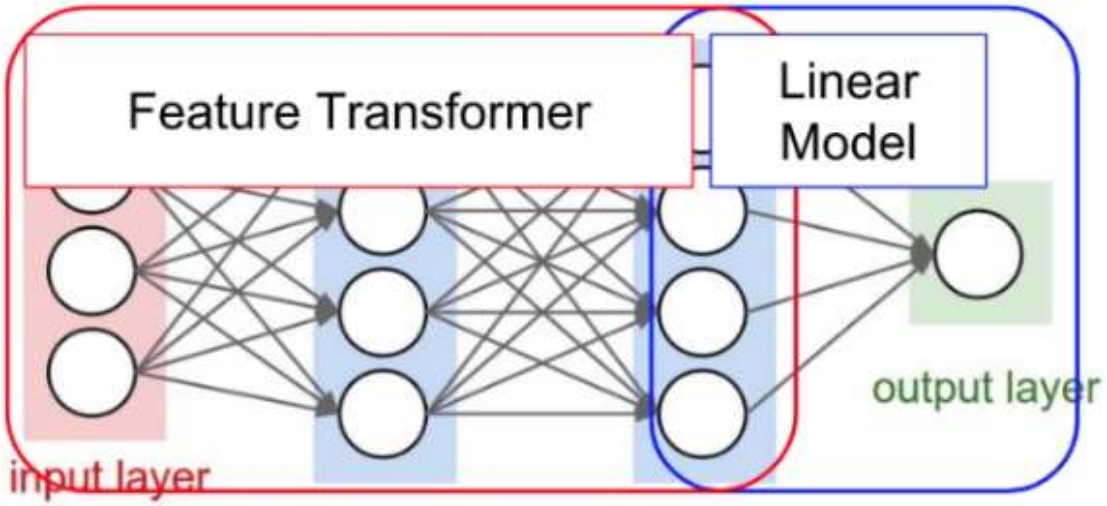


Ilustración 23: Esquema de la división de una red neuronal convolucional en dos partes: Feature Transformer y Linear Model (Lazy Programmer, 2018)

La primera parte, que engloba a todas las capas convolucionales y de *pooling* es la encargada de extraer las características de bajo, medio y alto nivel para cada entrada en forma de foto. La segunda parte es la encargada de ver qué características de alto nivel tiene una entrada, y clasificar a la entrada acorde a ello.

Si pensamos en la naturaleza de las imágenes, al final todas las imágenes son iguales, no es como un problema en el que las entradas son datos bancarios y otro en el que las entradas son datos biológicos, que no tienen nada que ver. Todas las imágenes tienen características similares, es por ello que podemos usar un extractor de características común. ¿Qué implica esto? Pues que podemos usar los pesos del extractor de características de una red neuronal ya entrenada. Y esto es precisamente lo que se hace.

Todas las arquitecturas neuronales famosas, incluida ResNet50 y VGG16 están entrenadas con la base de datos ImageNet, y con algoritmos de entrenamiento y potencia computacional suficiente para encontrar una combinación de pesos muy próxima al óptimo para dicha base de datos. Sin embargo gracias a la propiedad de *Transfer Learning* propia de este tipo de problemas con imágenes, y debido a la gran variedad de imágenes diferentes que contiene esa base de datos, las capas de la arquitectura están preparadas para la extracción de cualquier característica propia de nuestro problema y nosotros solo deberemos entrenar la segunda parte de nuestra red, la del clasificador lineal, para que aprenda dadas las características de alto nivel que son extraídas de las imágenes, aprender a clasificarlas en diferentes grupos.

Capítulo 4.

Comparación de diferentes arquitecturas convolucionales para nuestro problema

4.1 Descripción de la base de datos usada.

Como ya se introdujo en el capítulo 1, el fin de este proyecto es el desarrollo de una herramienta que sea capaz de reconocer actores en imágenes extraídas de archivos de vídeo. Al estar la base de datos que se usará para el entrenamiento del modelo usado por la aplicación en proceso de creación, se buscó en Internet una base de datos abierta, con el mayor número de actrices y actores posibles y con el mayor número de fotos por persona, de manera que la base de datos sea lo más similar posible a la que se usará en un futuro.

La base de datos que encontramos se llama *thumbnails_features_deduped_publish* y cuenta con imágenes para más de 1500 personas famosas. Es la mejor base de datos que se adapta al problema que estoy intentando resolver, aun así, las imágenes han sido sacadas con técnicas de *scrapeo* de red, por lo que no es la base de datos idónea para nuestro problema, veremos cómo se comporta.

Del total de personas en la base de datos se hizo una selección de 145 actrices y actores con los que vamos a entrenar nuestra red neuronal. Serán las 145 personas que será capaz de reconocer nuestro modelo.

Los actores y actrices incluidos en nuestros datos son:

- Aamir Khan, Aaron Eckhart, Abigail Breslin, Adam Brody, Adam Sandler, Adrian Grenier, Adrien Brody, Al Pacino, Amanda Bynes, Amber Heard, Amy Adams, Andy García, Angelica Bridges, Angelina Jolie, Anne Hathaway, Anthony Hopkins, Antonio Banderas, Arnold Schwarzenegger, Asthon Kutcher, Audrey Hepburn.
- Barbra Streisand, Ben Afflek, Ben Stiller, Brad Pit, Brendan Fraser, Bridget Fonda, Bruce Lee, Bruce Willis.
- Cameron Diaz, Channing Tatum, Charlie Chaplin, Charlie Sheen, Charlize Theron, Christian Bale, Chuck Norris, Clint Eastwood, Clive Owen, Cuba Gooding.
- Daniel Craig, Del Toro, Demi Moore, Denzel Washington, Drew Barrymore, Dustin Hoffman, Dwayne Johnson.
- Eddy Murphy, Edward Norton, Elijah Wood, Elsa Pataky, Emma Watson, Eva Mendes, Ewan McGregor.
- Gary Oldman, George Clooney, Gerarld Butler.
- Halle Berry, Harrison Ford, Heath Ledger, Hugh Grant, Huge Jackman, Humphrey Bogart.
- Ice Cube.

- Jack Black, Jack Nicholson, Jackie Chan, James Dean, James Franco, Jamie Foxx, Jane Fonda, Jared Leto, Jason Statham, Javier Bardem, Jennifer Aniston, Jessica Alba, Jet Li, Jim Carrey, Jodie Foster, John Travolta, John Wayne, Johnny Deep, Jude Law, Julia Roberts, Justin Tiimberlake.
- Kate Beckinsale, Keanu Reeves, Keira Knightley, Kevin Costner, Kevin Spacey, Kim Basinger, Kirsten Dunst.
- Leonardo Dicaprio, Liam Neeson, Lindsay Lohan, Liv Tayler, Lucy Liu.
- Macaulay Culkin, Mark Wahlberg, Marlon Brando, Matt Damon, Matthew McConaughey, Meg Ryan, Megan Fox, Mel Gibson, Meryl Streep, Michael Douglas, Michelle Rodriguez, Milla Jovovich, Morgan Freeman.
- Natalie Portman, Nick Nolte, Nicolas Cage, Nicole Kidman.
- Orlando Bloom, Owen Wilson.
- Pamela Anderson, Paul Newman, Paul Walker.
- Richard Gere, Robert Downey, Robert Pattison, Robert Redford, Robin Williams, Russel Crowe, Ryan Gosling, Ryan Reynolds.
- Salma Hayek, Sandra Bullock, Scarlett Johansson, Sean Connery, Sean Penn, Sharon Stone, Snoop Dogg, Steve Carell, Steve Martin, Steven Seagal, Sylvester Stallone.
- Tom Cruise, Tom Hanks.
- Viggo Mortensen, Vin Diesel.
- Wesley Snipes, Will Ferrel, Will Smith, Winona Ryder.
- Zac Efron.

Tenemos un total de 41525 imágenes para las 145 clases. De este total de imágenes se separó 3804 para formar el conjunto de prueba y ver que tal responde el modelo ante imágenes con las que no ha sido entrenado.

4.2 Hardware usado

A pesar de que vamos a usar el truco de *Transfer Learning,* entrenar una red neuronal convolucional es una tarea computacionalmente muy pesada. Se usó un ordenador personal para el entrenamiento de los modelos pertinentes que, además de contar con 16 GB de memoria RAM, cuenta con una tarjeta gráfica Nvidia GTX 1060.

Que disponga de tarjeta gráfica Nvidia es muy importante, ya que para esta marca existe la posibilidad de hacer computación distribuida proporcionada por la librería tensorflow, y mandar parte de la computación a la tarjeta RAM, simultáneamente con la computación que se realiza en la memoria principal, con lo que aceleramos bastante el proceso de entrenamiento.

4.3 VGG16: Detalles de implementación, entrenamiento y pruebas

4.3.1 Detalles de implementación

La librería Keras, que usa Tensorflow de *backend* y presenta un nivel de abstracción mayor que la misma tiene su modelo VGG16 entrenado con el paquete de imágenes ImageNet. La carga y uso de dicha arquitectura es muy simple, basta con esta línea de código:

```
model_vgg = VGG16(include_top = True, weights='imagenet', input_shape = (224,224,3))
```

Con esta línea, cargamos en la variable *model_vgg* el modelo ya entrenado. Al añadir el parámetro *include_top = True,* cargamos la arquitectura incluidas las tres capas de neuronas finales que constituyen el Perceptrón Multicapa que clasifica las características extraídas por las capas convolucionales de la red neuronal. Si en vez de eso, el valor del parámetro *include_top* fuera *False*, solo cargaríamos la parte convolucional.

No obstante, la implementación de la arquitectura en tensorflow nos aporta un mayor grado de libertad y personalización en nuestra red, por lo que he desarrollado en dicha librería mi propia clase con arquitectura VGG16, aquí están algunos de los detalles de la implementación.

La primera función que se ha programado se llama *init_filter.* Como se puede deducir por el nombre, su uso tiene lugar al inicializar los filtros de las capas convolucionales. Pese a que los filtros en la teoría tienen tres dimensiones, en la práctica es más interesantes que sean matrices de cuatro dimensiones, para que el paso de la entrada por el filtro se desarrolle como una multiplicación matricial.

```
def init_filter(d, mi, mo):
    return (np.random.randn(d, d, mi, mo) * np.sqrt(2.0 / (d * d * mi)))
```

Con esta función que toma como parámetros la altura y la anchura del filtro, (d), el número de características que tiene la entrada (*mi*) y el número de características que tiene la salida (*mo*), devuelve una matriz de cuatro dimensiones, (altura del filtro x anchura del filtro x nº características iniciales x nº características finales). Desarrolladores de este campo indican que es beneficioso inicializar los pesos de las variables en una red neuronal con valores aleatorios que siguen una distribución normal de parámetros media = cero y varianza = $\frac{2}{d*d*nº caracteres\_inicial}$, que es exactamente lo que hace esta función.

La siguiente clase que he desarrollado es la de Capa Convolucional:

```
class ConvLayer:

    def __init__(self, d, mi, mo, activation = tf.nn.relu, stride = 1, padding='SAME'):
        self.W = tf.Variable(init_filter(d, mi, mo), dtype = np.float32)
        self.b = tf.Variable(np.zeros(mo, dtype=np.float32))
        self.stride = stride
        self.padding = padding
        self.function = activation
        self.session = None

    def forward(self, X):
        X = tf.nn.conv2d(
            X,
            self.W,
            strides=[1, self.stride, self.stride, 1],
            padding=self.padding)
        X = self.function(X + self.b)
        return X

    def copyFromKerasModel(self, layer):
        W, b = layer.get_weights()
        op1 = self.W.assign(W)
        op2 = self.b.assign(b)
        self.session.run((op1, op2))

    def setTrainable (self, booolean):
        self.W.trainable = boolean
        self.b.trainable = boolean

    def setSession (self, session)
        self.session = session
```

Para crear un objeto que representa una capa convolucional solo hace falta introducir los parámetros de dimensiones del filtro. Por defecto la función de activación será a ReLu, el valor del *stride* será 1 y aplicaremos el algoritmo de *padding* para mantener las dimensiones de la entrada, aunque estos parámetros son modificables. Utilizamos la función de tensorflow *conv2d* en el método *forward* de la clase, para el cómputo del paso de la entrada por la capa de la manera más eficiente posible. Una de las ventajas de tensorflow es que no hace falta definir el algoritmo de *backpropagation,* tensorflow guarda la red neuronal en forma de grafo, e infiere este algoritmo por si solo. Es por eso que la función *back_propagation* no está implementada.

La función *copy_from_keras* está diseñada para aprovechar el truco del *Transfer Learning,* y darle el valor de los filtros del modelo entrenado de Keras a nuestra red, ante la imposibilidad de entrenar dicha red por nosotros mismos. Como hemos dicho, la única parte que vamos a querer entrenar de dicha red es el Perceptrón Multicapa final, por lo que tendremos que poder configurar que a una variable no se le aplique el algoritmo de *back_propagation.* Para eso está la función setTrainable. Como tensorflow funciona por sesiones, y queremos todo nuestro modelo dentro de la misma sesión, tenemos la última función: setSesion

El otro componente fundamental de las arquitecturas convolucionales es la capa de *MaxPooling*, aquí tenemos su implementación:

```
class MaxPoolingLayer:

    def __init__ (self, size, stride, padding = "VALID"):
        self.size = size
        self.stride = stride
        self.padding = padding

    def forward (self, X):
        X = tf.nn.max_pool(X,
                ksize = [1, self.size, self.size, 1],
                strides = [1, self.stride, self.stride, 1],
                padding = self.padding)
        return X
```

Para crear una instancia de dicha clase, necesitamos los valores *size,* que define el número por el que se divide las dimensiones de la matriz de entrada. Es decir, si tiene un valor de dos, las dimensiones de altura y anchura de la entrada serán la mitad en la salida. Los valores de *stride* y de *padding* también son configurables, aunque por defecto tendremos que *stride* será igual a *size,* en esta capa no nos interesa que los valores se solapen, y no queremos aplicar el algoritmo de *padding,* ya que el fin principal de esta capa es el de reducir las dimensiones de la matriz, perdiendo la mínima información posible.

Para el cómputo del algoritmo nos volvemos a ayudar de una funcióón de tensorflow, en este caso la clase *max_pool.* Como en esta capa no hay ninguna variable de pesos, no hace falta inicializar nada, ni una función que configure si queremos entrenar la variable o no.

Para la implementación del perceptrón multicapa final, necesitamos capas de neuronas normales. Aquí está el código de una capa neuronal básica:

```
class DenseLayer:

    def __init__(self, ninput, noutput, activation = tf.nn.relu):
        self.W = tf.Variable(np.random.randn(ninput, noutput) * np.sqrt(2.0 / ninput), dtype = np.float32)
        self.b = tf.Variable(np.zeros(noutput), dtype = np.float32)
        self.function = activation

    def forward (self, X):
        return self.function(tf.matmul(X, self.W) + self.b)
```

Es una clase muy simple, inicializamos las variables: matriz de pesos (W) con los valores recomendados que hemos explicado antes y la matriz de *bias* (b) con ceros, y creamos la función que computa el algoritmo *forward_propagation.* Gracias a las propiedades de la multiplicación matricial, podemos definir el paso de un conjunto de entradas por una capa neuronal como una multiplicación matricial entre la matriz con las entradas y la matriz con los pesos, y sumarle después el valor de *bias*. Como es tan sencilla y eficiente, en esta clase no nos tenemos que ayudar de ninguna función para computar este algoritmo.

La última clase que necesitamos para formar nuestra arquitectura VGG16 es la capa *Flatten.* Esta capa es necesaria para, como indica su propio nombre, “aplanar” la matriz final, resultado del paso de la entrada por las capas de convolución, a un vector de dimensión 1, que es la entrada

que necesita el perceptrón multicapa. Al igual que la capa de *pooling,* esta capa tampoco tiene variables. Aquí podemos ver su implementación.

```
class Flatten :
    def __init__(self):
        None
    def forward (self, X):
        return tf.reshape(X, shape = [4, -1])
```

Una vez que tenemos implementadas todas las partes de nuestra arquitectura, podemos definir la clase de la misma.

```
class VGG16:

    def __init__ (self, noutput):

        self.X_input = tf.placeholder(shape = (None, 224, 224, 3), dtype = np.float32)
        self.layers = []
        self.session = tf.InteractiveSession()


        self.layers.append(ConvLayer(3,3,64))
        self.layers[0].setSession(self.session)
        self.layers.append(ConvLayer(3,64,64))
        self.layers[1].setSession(self.session)
        self.layers.append(MaxPoolingLayer(2,2))

        self.layers.append(ConvLayer(3,64,128))
        self.layers[3].setSession(self.session)
        self.layers.append(ConvLayer(3,128,128))
        self.layers[4].setSession(self.session)
        self.layers.append(MaxPoolingLayer(2,2))

        self.layers.append(ConvLayer(3,128,256))
        self.layers[6].setSession(self.session)
        self.layers.append(ConvLayer(3,256,256))
        self.layers[7].setSession(self.session)
        self.layers.append(ConvLayer(3,256,256))
        self.layers[8].setSession(self.session)
        self.layers.append(MaxPoolingLayer(2,2))

        self.layers.append(ConvLayer(3,256,512))
        self.layers[10].setSession(self.session)
        self.layers.append(ConvLayer(3,512,512))
        self.layers[11].setSession(self.session)
        self.layers.append(ConvLayer(3,512,512))
        self.layers[12].setSession(self.session)
        self.layers.append(MaxPoolingLayer(2,2))

        self.layers.append(ConvLayer(3,512,512))
        self.layers.append(ConvLayer(3,512,512))
        self.layers.append(ConvLayer(3,512,512))
        self.layers.append(MaxPoolingLayer(2,2))
        self.layers.append(Flatten())

        self.layers.append(DenseLayer (25088, 4096))
        self.layers.append(DenseLayer (4096, 4096))
        self.layers.append(DenseLayer (4096, noutput, activation = tf.nn.sigmoid))
```

El único parámetro que necesitamos darle a esta clase para crear una instancia del mismo, una red neuronal con una arquitectura VGG16, es el número de salidas, es decir, el número de clases entre las que tiene que clasificar nuestra red. Vemos que, para implementar la red, he creado un vector de capas, y a este le he ido añadiendo cada capa indicada en el *paper* original de este modelo.

También se inicia una sesión de tensoflow, en el momento de la creación de la instancia, y se configura la sesión de cada capa, para que todas las variables del modelo estén en la misma sesión.

Como tenemos todas las capas en un vector de capas, y para cada capa la función que computa el algoritmo de *forward propagation* se llama de la misma manera, la función que computa dicho algoritmo para toda la arquitectura es muy sencillo:

```
def forward (self, X):
    for i in self.layers:
        X = i.forward(X)
    return X
```

El último método que compone nuestra clase VGG16 es la siguiente:

```
def CopyFromKeras(self):
    keras_model = keras.applications.vgg16.VGG16(include_top=False, weights='imagenet')
    self.layers[0].copyFromKerasModel(keras_model.layers [1])
    self.layers[0].setTrainable(False)
    self.layers[1].copyFromKerasModel(keras_model.layers [2])
    self.layers[1].setTrainable(False)

    self.layers[3].copyFromKerasModel(keras_model.layers [4])
    self.layers[3].setTrainable(False)
    self.layers[4].copyFromKerasModel(keras_model.layers [5])
    self.layers[4].setTrainable(False)

    self.layers[6].copyFromKerasModel(keras_model.layers [7])
    self.layers[6].setTrainable(False)
    self.layers[7].copyFromKerasModel(keras_model.layers [8])
    self.layers[7].setTrainable(False)
    self.layers[8].copyFromKerasModel(keras_model.layers [9])
    self.layers[8].setTrainable(False)

    self.layers[10].copyFromKerasModel(keras_model.layers [11])
    self.layers[10].setTrainable(False)
    self.layers[11].copyFromKerasModel(keras_model.layers [12])
    self.layers[11].setTrainable(False)
    self.layers[12].copyFromKerasModel(keras_model.layers [13])
    self.layers[12].setTrainable(False)

    self.layers[14].copyFromKerasModel(keras_model.layers [15])
    self.layers[14].setTrainable(False)
    self.layers[15].copyFromKerasModel(keras_model.layers [16])
    self.layers[15].setTrainable(False)
    self.layers[16].copyFromKerasModel(keras_model.layers [17])
    self.layers[16].setTrainable(False)
```

Este método descarga el modelo VGG16 de Keras, y le da los valores de cada filtro de cada capa, a nuestros propios filtros y aplicar así, el truco del *Transfer Learning.*

Aunque la clase funciona, y se ha probado que da las mismas salidas que la clase VGG16 de Keras, para nuestra herramienta vamos a utilizar esta segunda, debido a que el entrenamiento va a ser tan computacionalmente costoso, que nos vamos a aprovechar de la definición de este mismo, de una manera en la que aprovecha los recursos del ordenador eficazmente.

4.3.2 Entrenamiento.

Algo de lo que no se ha comentado nada hasta ahora es el hecho de que la entrada a nuestra red neuronal tenga que ser una imagen de dimensiones 224x224x3. Es muy complicado, por no decir imposible, que nuestro archivo de imágenes que forma la base de datos con la que vamos a entrenar y a usar este modelo, sea un archivo con todas las imágenes de esta dimensión. Entonces, ¿qué hacemos?, ¿solo podemos usar las imágenes que cumplen dicho requisito?

Lo cierto es que no, lógicamente podemos ajustar las imágenes entrantes para que se adapten a estas medidas de entrada. Programar esto no es fácil, ya que nuestra manera de interpretar imágenes en programación es mediante una matriz tridimensional. Por suerte nos podemos ayudar de un generador que tiene implementado Keras, para hacer esta tarea.

Un generador, es una especie de función que se puede definir en lenguajes como Python, que se usa para generar datos en tiempo de ejecución.

El generador definido por Keras no solo nos redimensiona las imágenes, sino que también nos ofrece un abanico de posibilidades que vamos a usar, para que el aprendizaje de nuestra red neuronal sea más efectivo.

Cuando nuestro archivo de imágenes no es lo suficientemente grande, conviene modificar un poco cada imagen, cada vez que se usa para entrenar a nuestra red. Con modificar, nos referimos a rotar la imagen hacia ambos lados, aplicarle zoom, o deformarla estirándola o contrayéndola tanto vertical como horizontalmente. De esta manera forzamos a la red a tener una capacidad mayor de generalización. Este es el código donde usamos el Generador de Keras con estas opciones.

```
from keras.preprocessing.image import ImageDataGenerator
from keras.applications.resnet50 import preprocess_input


gen = ImageDataGenerator(

    rotation_range=20,
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode = "nearest"
)



train_generator = gen.flow_from_directory(
    "Data",
    target_size=(224, 224),
    batch_size=64,
    class_mode='categorical')
```

Para que este generador funcione, el directorio donde se encuentran nuestras fotos, en este caso el directorio se llama “Data”, tiene que estar formado por carpetas con el nombre de la clase que tengan en su interior las imágenes pertenecientes a la misma. El generador nos devolverá un *batch* con 64 imágenes seleccionadas aleatoriamente de entre todas las imágenes que conforman el directorio, y una matriz con la clase a la que pertenece cada imagen, en una codificación *one-hot*. Esta codificación es muy propia de los algoritmos de clasificación, y se trata

de un vector de ceros, exceptuando una posición que su valor será 1, y corresponde a la clase a la que pertenece.

Redimensionar las imágenes no es la única solución que podemos aplicar al problema de los distintos tamaños en las imágenes. Como la configuración de la red es indicar el tamaño de los filtros, que van a ser del mismo tamaño, sea la imagen del tamaño que sea, podemos dejar las imágenes cada una con su tamaño. La imagen pasará por las capas convolucionales y de *pooling* sin problema. El tamaño de la matriz de salida de la última capa convolucional dependerá del tamaño de la imagen de entrada. El problema es que esta matriz es la entrada de nuestro Perceptrón Multicapa, y a entrada de este sí que tiene que tener un número fijo, por lo que la solución a este problema es introducir después de la última capa de convolución, una capa *GolbalMaxPooling,* que lo que hace es aplicar el algoritmo de *pooling* a las dimensiones de la imagen para quedarnos con un solo valor, el máximo, y de esa manera forzar a que la salida siempre sea del mismo tamaño.

Inicialmente esta solución nos parecía más efectiva que la otra, aunque los resultados de las pruebas iniciales me mostraron que para nuestro problema iba a ser más efectiva la redimensión de las imágenes de entrada.

La idea inicial era copiar los pesos de las capas convolucionales, iniciar las dos capas del Perceptrón Multicapa aleatoriamente y entrenar solo estas últimas capas, de manera que la configuración de la red neuronal quedaría así:

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_2 (Flatten)	(None, 25088)	0
dense_29 (Dense)	(None, 4096)	102764544
dense_30 (Dense)	(None, 4096)	16781312
dense_31 (Dense)	(None, 145)	594065

Total params: 134,854,609
Trainable params: 120,139,921
Non-trainable params: 14,714,688

Ilustración 24: Resumen de la configuración de VGG y el número de parámetros ofrecida por Keras

Aquí es donde está el principal problema, después de la descripción por capas, tenemos el número de parámetros que tiene nuestra red neuronal. Entre las ultimas 3 capas neuronales normales que forman el perceptrón de clasificación suman más de 120 millones de parámetros.

Hacer llegar el algoritmo de *back propagation* a tantos millones de parámetros es inviable con el hardware disponible, por lo que se ha probado distintas alternativas para implementar el modelo:

- La primera fue reducir el número de neuronas de una de las capas finales y prescindir de la otra. Este es el resumen de las últimas capas de la configuración de la nueva red neuronal, que es donde cambia de la anterior:

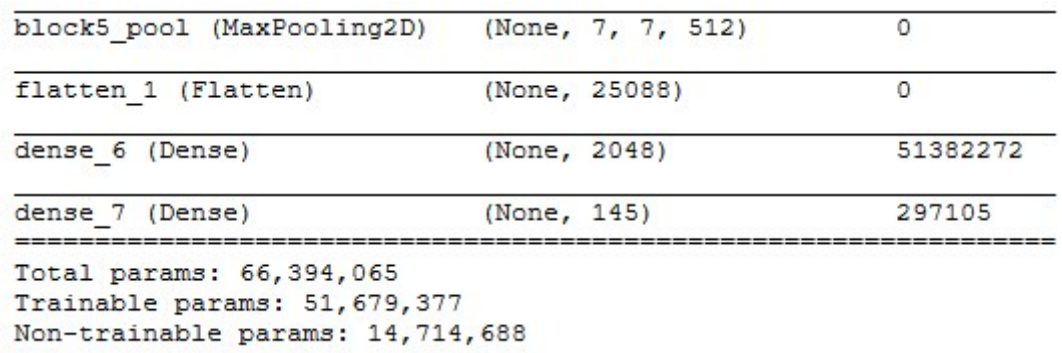


Ilustración 25: Resumen de la configuración de VGG y el número de parámetros ofrecida por Keras

Los resultados de esta configuración no son los esperados, por lo que se siguieron buscando más opciones.

- La segunda aproximación fue introducir una capa *GlobalMaxPooling,* de manera que se reduzca drásticamente el tamaño de la salida del bloque convolucional, ello provoque que se reduzca también el tamaño de la entrada al perceptrón multicapa, y, por consiguiente, el número de parámetros total. La configuración quedaría de la siguiente manera:

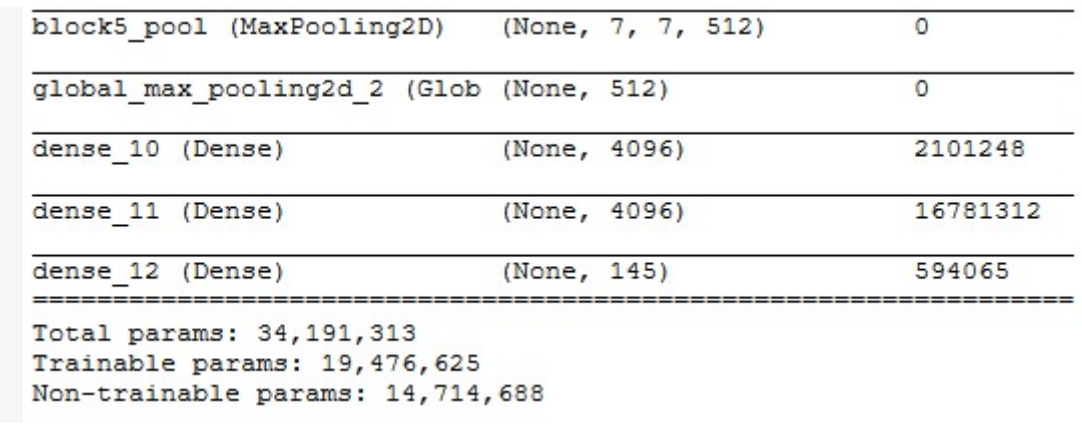


Ilustración 26: Resumen de la configuración de VGG y el número de parámetros ofrecida por Keras

Los resultados de esta configuración no son buenos tampoco. Volver a releer la teoría del modelo nos hizo pensar que igual se está utilizando una red neuronal con un número de parámetros mucho mayor del necesario, y por eso la red neuronal no funcionaba. Esto nos hizo pensar en la siguiente configuración.

- En esta tercera aproximación, decido prescindir de las dos capas intermedias de 4096 neuronas, y colocar directamente la capa de salida después del bloque convolucional. El resumen de la configuración de este modelo es el siguiente:

```
block5_pool (MaxPooling2D)   (None, 7, 7, 512)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_13 (Dense)             (None, 145)               3637905
=================================================================
Total params: 18,352,593
Trainable params: 3,637,905
Non-trainable params: 14,714,688
```

Ilustración 27: Resumen de la configuración de VGG y el número de parámetros ofrecida por Keras

Vemos que el número de parámetros ha descendido. Sin embargo, la red neuronal sigue sin funcionar. El problema no era ese.

- Después de buscar en foros diversas configuraciones y soluciones a problemas similares, encontré una que me gustó. La idea es cargar también los pesos de las dos capas del perceptrón del modelo entrenado. En la fuente consultada solo entrenaba la capa de salida de la red neuronal, yo he decidido entrenar también la penúltima capa, modificando los pesos de la arquitectura entrenada. Este es el esquema de la configuración. Para consultar la fuente de la que se extrajo está idea: (Deeplizard, 2017)

```
block5_pool (MaxPooling2D)   (None, 7, 7, 512)        0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
dense_1 (Dense)              (None, 145)               594065
=================================================================
Total params: 134,854,609
Trainable params: 17,375,377
Non-trainable params: 117,479,232
```

Ilustración 28: Resumen de la configuración de VGG y el número de parámetros ofrecida por Keras

Finalmente se obtienen resultados buenos y se entrena esta configuración a fondo. Este es el código de la orden de iniciar la computación del entrenamiento de la red neuronal:

```
model.fit_generator(train_generator, steps_per_epoch = 254, epochs = 180)
```

Keras tiene diversas ordenes que ejecutan diferentes modos de entrenamiento. Se elegió este ya que es el que contempla la opción de que el conjunto de datos de entrada sea proporcionado por un generador, y así usamos la herramienta de Keras para formar el conjunto de datos de entrenamiento.

El entrenamiento está dividido en 180 épocas de 254 etapas en cada época. En cada etapa el modelo computa la salida de 64 imágenes (tamaño del *batch*) y aplica la función de coste "*caregorica crossentrophy"* para modelos de clasificación introducida en la sección 3.2, sobre las salidas obtenidas y las salidas deseadas y actualiza los pesos conforme a ello. Si hacemos la

operación $180 * 254 * 64 = 2926080$ es el número de imágenes totales que se computa en la fase de entrenamiento para actualizar los pesos. Es una operación larga y costosa de la que vamos a ver un pequeño resumen.

```
Epoch 1/180
254/254 [==============================] - 151s 596ms/step - loss: 4.3518 - acc: 0.0872
Epoch 2/180
254/254 [==============================] - 157s 620ms/step - loss: 3.9227 - acc: 0.1283
Epoch 3/180
254/254 [==============================] - 156s 613ms/step - loss: 3.6479 - acc: 0.1674
Epoch 4/180
254/254 [==============================] - 156s 616ms/step - loss: 3.5304 - acc: 0.1847
Epoch 5/180
254/254 [==============================] - 156s 612ms/step - loss: 3.4344 - acc: 0.2019
```

Ilustración 29: Resumen del principio de la etapa de entrenamiento ofrecido por Keras

Aquí vemos el resumen de las primeras 5 etapas de entrenamiento. En un comienzo el valor de la función de coste para cada batch era de 4.35. Otra medida que nos da para ver el progreso es la precisión (*accuracy*). Después de la primera etapa el modelo era capaz de acertar solo en 8 de cada 100 imágenes del conjunto de entrenamiento.

Cada etapa tarda en computarse algo más de dos minutos y medio, unos 155 segundos. El tiempo total de computación llega casi a las 8 horas de tiempo de ejecución (7 horas y 45 minutos). Tras ese tiempo vamos a ver como nuestra red neuronal ha cambiado.

```
Epoch 176/180
254/254 [==============================] - 155s 611ms/step - loss: 0.2952 - acc: 0.9112
Epoch 177/180
254/254 [==============================] - 156s 613ms/step - loss: 0.2892 - acc: 0.9143
Epoch 178/180
254/254 [==============================] - 156s 612ms/step - loss: 0.3053 - acc: 0.9072
Epoch 179/180
254/254 [==============================] - 155s 611ms/step - loss: 0.2957 - acc: 0.9119
Epoch 180/180
254/254 [==============================] - 158s 621ms/step - loss: 0.3106 - acc: 0.9070
```

Ilustración 30:Resumen del final de la etapa de entrenamiento ofrecido por Keras

Este es el resumen de las últimas etapas de entrenamiento, vemos como ya la red se ha estabilizado y no mejora (el valor de las métricas de las últimas etapas es muy similar). El valor de la función de coste ha descendido a valores próximos a 0.3 y el valor de la precisión de nuestra red ronda el 0.91, es decir, que nuestro modelo acierta en 91 de cada 100 imágenes de nuestro conjunto de datos de entrenamiento.

4.3.3 Pruebas

Una vez que tenemos entrenado el modelo es el momento de probarlo con el conjunto de imágenes apartadas de la fase de entrenamiento. Como nos sigue interesando la manera en la que el generador de Keras nos forma conjuntos de datos listos para ser la entrada de la red neuronal, volvemos a utilizar esta herramienta para comprobar la eficiencia de nuestro modelo.

```
gen2 = ImageDataGenerator(
    fill_mode = "nearest"
)

test_generator = gen2.flow_from_directory(
    "Test",
    target_size = (224, 224),
    batch_size = 64,
    class_mode = 'categorical')
```

En este caso prescindo de las opciones de rotar la imagen, o invertirla vertical u horizontalmente. Para usar el generador con fines de testear el modelo utilizo el siguiente bucle:

```
contador = 0
for i in range(254):
    test, real = next(test_generator)
    respuestas = model.predict(test)
    for j in range(real.shape[0]):
        if (np.argmax(respuestas[j]) == np.argmax(real[j])):
            contador = contador + 1
print ("Eficacia del modelo: " + str(contador/(254*64) * 100) + " %")
```

```
Eficacia del modelo: 52.436023622047244 %
```

Como no podemos ordenarle a este generador que coja todas las fotos y las convierta en una matriz que encaje con la entrada de nuestro modelo, creamos 254 conjuntos de 64 imágenes cada uno y lo usamos para ver qué tal se comporta nuestro modelo. Nuestro conjunto de imágenes de test lo componen 3804 imágenes, y de esta manera estamos pasando a nuestro modelo 16256 imágenes, por lo que el resultado es representativo. Para comprobar la eficiencia del modelo simplemente tenemos una variable “contador” que aumenta su valor cada vez que el modelo acierta. Después del paso por el bucle, se divide entre el número total de fotos para ver el porcentaje de acierto.

El resultado es que se acierta en algo más de la mitad de las fotos, un 52.43%. Es un resultado muy pobre, que comparado con la tasa de acierto del conjunto de entrenamiento nos puede indicar que estamos incurriendo en algún caso de *overfitting.* Vamos a mostrar alguna imagen de algún famoso y quien predice el modelo que es.

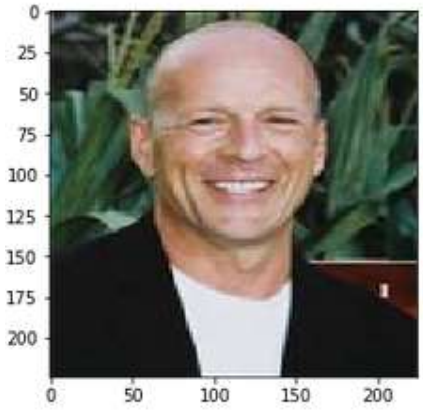
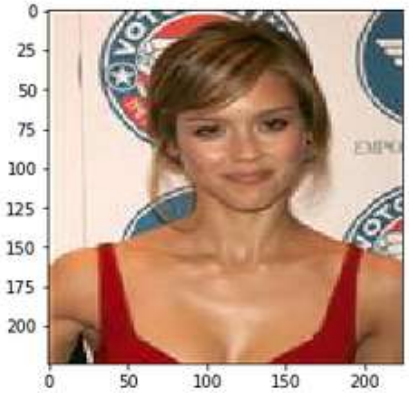
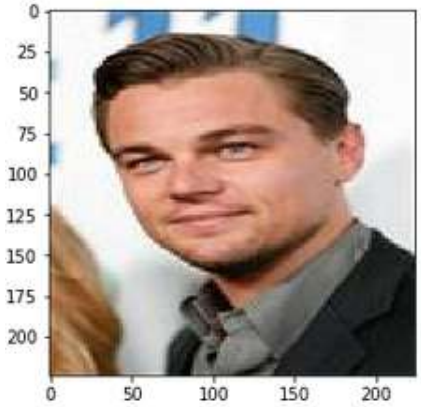
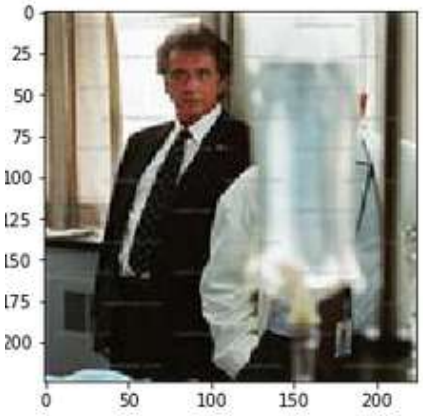
Imagen (Entrada)	Predicción	Real
	Bruce Willis	Bruce Willis
	Jessica Alba	Jessica Alba
	Brendan Fraser	Leonardo DiCaprio
	Richard Gere	Al Pacino

Tabla 4: Entrada resultado predicho por el modelo y resultado real

Vemos dos casos en los que el modelo acierta, y dos en los que el modelo falla. Como podemos ver, las imágenes no son de la mayor calidad posible, la dimensión de la mayor parte de ellas era menor que 224x224, por lo que se pierde calidad al ampliarla. Es discutible si se parecen los actores de la foto a los que confunde la red, pero por lo menos no parece que cometa errores de confundir un actor por una actriz, o actores de diferentes etnias.

4.4 ResNet: Detalles de implementación, entrenamiento y pruebas.

4.4.1 Detalles de implementación:

Lo que le hace diferente a la arquitectura ResNet son los bloques residuales. Para construir un bloque residual, necesitamos capas convolucionales, cuya implementación es idéntica a la del apartado anterior, y capas *batchnorm*. Aquí vemos el código de la implementación de esta capa que nos falta:

```
class BatchNormLayer:
    def __init__(self, D):
        self.running_mean = tf.Variable(np.zeros(D, dtype=np.float32), trainable=False)
        self.running_var  = tf.Variable(np.ones(D, dtype=np.float32), trainable=False)
        self.gamma        = tf.Variable(np.ones(D, dtype=np.float32))
        self.beta         = tf.Variable(np.zeros(D, dtype=np.float32))

    def forward(self, X):
        return tf.nn.batch_normalization(
                    X,
                    self.running_mean,
                    self.running_var,
                    self.beta,
                    self.gamma,
                    1e-3)
```

Como en anteriores ocasiones, nos hemos ayudado de la función de tensorflow correspondiente a la computación del paso de la entrada por una capa *batchnorm*, en este caso la función usada es *tf.nn.batch_normalization.* Si vemos la documentación de esta capa en la página de tensorflow, y tenemos en mente como funciona esta capa (Sección 4.3.2) vemos que la función necesita como parámetros in inicializador para los valores de beta y gamma, y variables que representen a las mismas. Son las cuatro variables que inicializamos cuando creamos una instancia de esta clase, y son los cuatro parámetros que le pasamos a la función cuando la invocamos en la función de la clase *forward* que representa la computación del algoritmo de *forward propagation* a través de esta capa.

Una vez que tenemos todos los componentes de los bloques implementados, procedemos a implementar el primer tipo de bloque residual, el que hemos denominado "*Identity Block",* que es capaz de aprender a pasar la entrada sin que sufra ningún tipo de modificación.

```
class IdentityBlock:
    def __init__(self, mi, fm_sizes, activation = tf.nn.relu):

        assert(len(fm_sizes) == 3)
        self.conv1 = ConvLayer(1, mi, fm_sizes[0], 1)
        self.bn1   = BatchNormLayer(fm_sizes[0])
        self.conv2 = ConvLayer(3, fm_sizes[0], fm_sizes[1], 1, 'SAME')
        self.bn2   = BatchNormLayer(fm_sizes[1])
        self.conv3 = ConvLayer(1, fm_sizes[1], fm_sizes[2], 1)
        self.bn3   = BatchNormLayer(fm_sizes[2])


    def forward(self, X):

        FX = self.conv1.forward(X)
        FX = self.bn1.forward(FX)
        FX = tf.nn.relu(FX)
        FX = self.conv2.forward(FX)
        FX = self.bn2.forward(FX)
        FX = tf.nn.relu(FX)
        FX = self.conv3.forward(FX)
        FX = self.bn3.forward(FX)

        return activation(FX + X)
```

Si nos fijamos en las especificaciones presentadas en el *paper* por los desarrolladores, (Sección 4.3.4) el tamaño de los filtros en el camino largo siempre es de: 1x1, 3x3 y 1x1 respectivamente, por lo que el tamaño de los filtros no será un parámetro de clase. La información que necesita la clase para ser implementada es: el número de características iniciales (mi), el número de características finales después de cada capa de convolución. Es imprescindible que el vector de características finales tenga longitud 3, debido a que son 3 filtros. Esto lo asegura la primera línea de código.

En el método *forward,* que como siempre implementa el algoritmo de *forward propagation,* simplemente computamos el paso de la entrada por el camino largo, a través de las capas de convolución, *batchnorm,* y funciones de activación antes definidas, y finalmente se le suma la salida de este camino largo a la entrada y se le hace pasar de nuevo por otra función de activación.

La clase también incluye un método para copiar los pesos desde el modelo entrenado de Keras, pero al ser idéntica a la de la implementación VGG16 no la he incluido.

El otro tipo de bloque residual es el que hemos denominado "*Convolutional Block".* Su implementación es muy similar a la del "*Identity Block":*

```
class ConvBlock:
    def __init__(self, mi, fm_sizes, stride=2, activation=tf.nn.relu):
        assert(len(fm_sizes) == 3)

        # Capas del camino largo
        self.conv1 = ConvLayer(1, mi, fm_sizes[0], stride)
        self.bn1   = BatchNormLayer(fm_sizes[0])
        self.conv2 = ConvLayer(3, fm_sizes[0], fm_sizes[1], 1, 'SAME')
        self.bn2   = BatchNormLayer(fm_sizes[1])
        self.conv3 = ConvLayer(1, fm_sizes[1], fm_sizes[2], 1)
        self.bn3   = BatchNormLayer(fm_sizes[2])

        # Capas del camino corto/puente
        self.convs = ConvLayer(1, mi, fm_sizes[2], stride)
        self.bns   = BatchNormLayer(fm_sizes[2])

    def forward(self, X):
        # Camino Largo
        FX = self.conv1.forward(X)
        FX = self.bn1.forward(FX)
        FX = tf.nn.relu(FX)
        FX = self.conv2.forward(FX)
        FX = self.bn2.forward(FX)
        FX = tf.nn.relu(FX)
        FX = self.conv3.forward(FX)
        FX = self.bn3.forward(FX)

        # shortcut branch
        SX = self.convs.forward(X)
        SX = self.bns.forward(SX)

        return activation(FX + SX)
```

La única diferencia es que hay que incluir las capas del camino corto, que ya no pasa el valor de entrada directamente, sino que lo modifica a través de una capa convolucional y una capa *batchnorm.* Además, en los detalles de implementación presentados en el *paper,* se especifica que tanto la primera capa de convolución del camino largo, como la capa de convolución del camino corto tengan un valor de *stride* = 2. (Para recordar lo que es significa el calor de *stride* ir a la sección 2.4.4.1).

4.4.2 Entrenamiento

Al igual que el anterior modelo, utilizamos el modelo de Keras debido a la codificación de las funciones de entrenamiento. Volvemos a usar el generador de Keras sobre la misma base de datos. Utilizamos la configuración original del modelo Res50, es decir, un *GlobalAveragePooling* tras el último bloque residual, y después de eso, la capa de salida de la red neuronal con 145 neuronas, debido a nuestras 145 clases. Los pesos de esta última capa serán los únicos modificables. Así queda la configuración de nuestra red:

```
res5c_branch2c (Conv2D)          (None, 7, 7, 2048)    1050624     activation_97[0][0]
bn5c_branch2c (BatchNormalizati (None, 7, 7, 2048)    8192        res5c_branch2c[0][0]
add_32 (Add)                     (None, 7, 7, 2048)    0           bn5c_branch2c[0][0]
                                                                   activation_95[0][0]
activation_98 (Activation)       (None, 7, 7, 2048)    0           add_32[0][0]
global_average_pooling2d_1 (Glo (None, 2048)          0           activation_98[0][0]
dense_2 (Dense)                  (None, 145)           297105      global_average_pooling2d_1[0][0]
==================================================================================================
Total params: 23,884,817
Trainable params: 297,105
Non-trainable params: 23,587,712
```

Ilustración 31: Resumen de la configuración de la red ResNet ofrecido por Keras

Aunque 297.105 parámetros modificables es una cifra a tener en cuenta, resultan muy pocos comparados con los más de 17 millones que teníamos en la arquitectura VGG. En este caso se podría entrenar la red entera utilizando los pesos del modelo entrenado como pesos iniciales. En ningún artículo o curso que se ha realizado para desarrollar este proyecto decían que eso era una idea recomendable, y los resultados que se obtuvieron después de intentarlo así lo confirman.

Una de las ventajas que tiene este modelo dado el archivo de imágenes que se está utilizando es el hecho de que la entrada de la red es otro parámetro configurable, no es necesario que sea una entrada 224x224 como en el caso de la arquitectura anterior. Sin embargo, en contra de nuestro pronóstico, entrenar la red con imágenes de resolución 100x100 da peores resultados que entrenarla con la resolución 224x224.

Como se trata de muchos menos parámetros que el modelo anterior, entreno la red durante menos tiempo, en vez de 180 épocas de 254 etapas cada época, la entreno durante 100 etapas del mismo número de épocas. Aquí podemos ver un resumen de como fue el entrenamiento:

```
model.fit_generator(train_generator, steps_per_epoch = 254, epochs = 100)
```

```
Epoch 1/100
254/254 [==============================] - 150s 589ms/step - loss: 4.3709 - acc: 0.0888
Epoch 2/100
254/254 [==============================] - 151s 593ms/step - loss: 3.9191 - acc: 0.1348
Epoch 3/100
254/254 [==============================] - 151s 594ms/step - loss: 3.6501 - acc: 0.1754
Epoch 4/100
254/254 [==============================] - 151s 593ms/step - loss: 3.5858 - acc: 0.1817
Epoch 5/100
254/254 [==============================] - 151s 595ms/step - loss: 3.3658 - acc: 0.2175
```

Ilustración 32: Resumen de la primera fase de la etapa de entrenamiento de la red ResNet, ofrecido por Keras

Durante las primeras épocas el desarrollo en cuanto a las dos métricas que obtenemos es muy similar. También es muy parecido el tiempo de computación de cada época, por lo que esperamos obtener resultados similares. Sin embargo:

```
Epoch 97/100
254/254 [==============================] - 229s 902ms/step - loss: 2.1482 - acc: 0.4424
Epoch 98/100
254/254 [==============================] - 233s 916ms/step - loss: 2.1280 - acc: 0.4439
Epoch 99/100
254/254 [==============================] - 229s 902ms/step - loss: 2.1608 - acc: 0.4329
Epoch 100/100
254/254 [==============================] - 227s 896ms/step - loss: 2.0958 - acc: 0.4445
```

Ilustración 33: Resumen de la etapa final de entrenamiento de la red resNet, ofrecido por Keras

La red se estanca en un valor para la función de pérdida de 2.1 y una precisión para el conjunto de entrenamiento de 0.44, es decir, acierta en 44 de cada 100 fotos. Acierta menos para el conjunto de entrenamiento que la anterior red para el conjunto de test. Debido a esto no esperamos tener unos buenos resultados en el apartado de pruebas.

4.4.3 Pruebas

Al igual que en la parte de pruebas del apartado anterior, se utiliza el mismo bucle para determinar la proporción de imágenes que clasifica correctamente en el conjunto de imágenes de test. Aquí tenemos los resultados:

```
contador = 0
for i in range(254):
    test, real = next(test_generator)
    respuestas = model.predict(test)
    for j in range(real.shape[0]):
        if (np.argmax(respuestas[j]) == np.argmax(real[j])):
            contador = contador + 1
print ("Eficacia del modelo: " + str(contador/(254*64) * 100) + " %")
```

```
Eficacia del modelo: 9.356545275590552 %
```

El modelo actúa de una manera bastante decepcionante, no llega a acertar ni el 10% de las imágenes de este conjunto de prueba. Al igual que en el apartado de pruebas del anterior modelo vamos a ver cómo clasifica el modelo imágenes específicas.

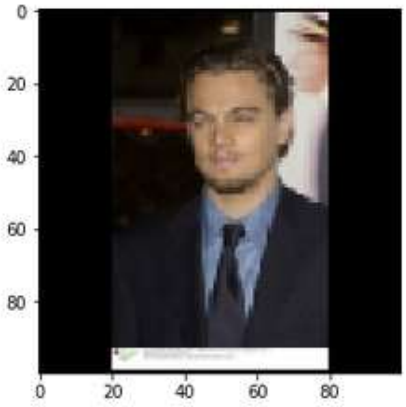
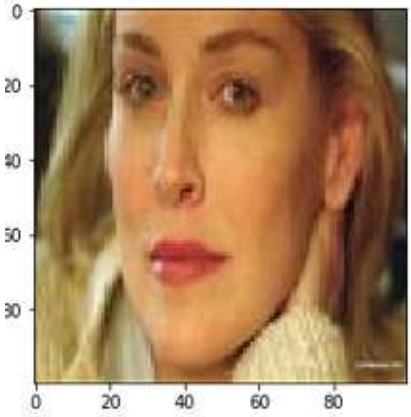
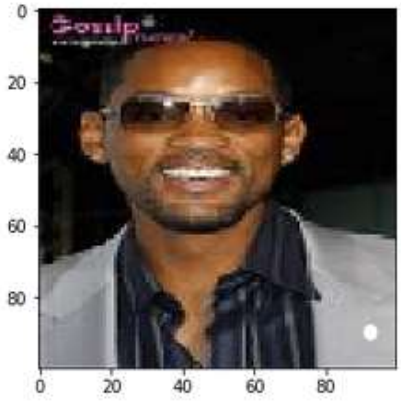
Imagen (Entrada)	Predicción	Real
	Tom Cruise	Leonardo DiCaprio
	Adam Sandler	Chuck Norris
	Lindsay Lohan	Sharon Stone
	Will Smith	Will Smith

Tabla 5: Resultados predichos por el modelo ResNet, y resultados reales

Como podemos ver en estos cuatro ejemplos, el modelo actúa peor que el anterior, desde nuestro punto de vista, las clases predichas erróneamente tienen menos parecido a las correctas que en el modelo VGG16. Sin embargo, al igual que el anterior, no confunde entre artistas de distinto sexo o etnia.

4.5 Conclusiones del capítulo

Ya podemos estar utilizando la arquitectura más novedosa, que, si no tenemos buenos datos, el resultado va a ser malo. Aunque nos quedaríamos con el modelo VGG16, porque los resultados son mucho mejores, sigue siendo insuficiente para implementar un modelo que sea la base de una aplicación que queremos que tenga un uso real. Si bien esta parte ha servido para comprobar el funcionamiento de ambas arquitecturas en la tarea de reconocimiento facial, se espera que con la base de datos que se va a utilizar para entrenar a nuestro modelo, este se comporte de una manera mucho más precisa.

Capítulo 5

Object Detection.

5.1 Introducción al problema Object Detection.

Hemos desarrollado una arquitectura que en mejor o peor medida puede clasificar una imagen de un actor o actriz dada, es decir, es capaz de reconocer quien es el famoso que sale en la imagen. El problema es que el propósito inicial de nuestra herramienta era el que sea capaz de determinar cuánto tiempo aparece en pantalla cada actor en la película.

Como bien es sabido, hay una gran cantidad de escenas en las que aparecen dos o más personas, y escenas en las que no aparece ninguno. Un clasificador no está programado para ello, ya que la premisa de un clasificador es que cada instancia pertenezca a un grupo, por lo tanto, clasifica cada entrada en un solo grupo, ya haya 10 actores en la imagen o ninguno.

Aunque la naturaleza del problema es distinta, ya no es clasificación de imágenes sino detección de objetos familiares en una imagen, la base de la solución del problema sigue siendo las capas convolucionales, solo que implementadas en algoritmos más complejos que explicaré a continuación. Prueba de ello es como la eficiencia de los algoritmos que pretenden dar solución está creciendo exponencialmente desde la llegada de las capas convolucionales y su implementación en estos algoritmos. En este gráfico vemos a lo que nos referimos:

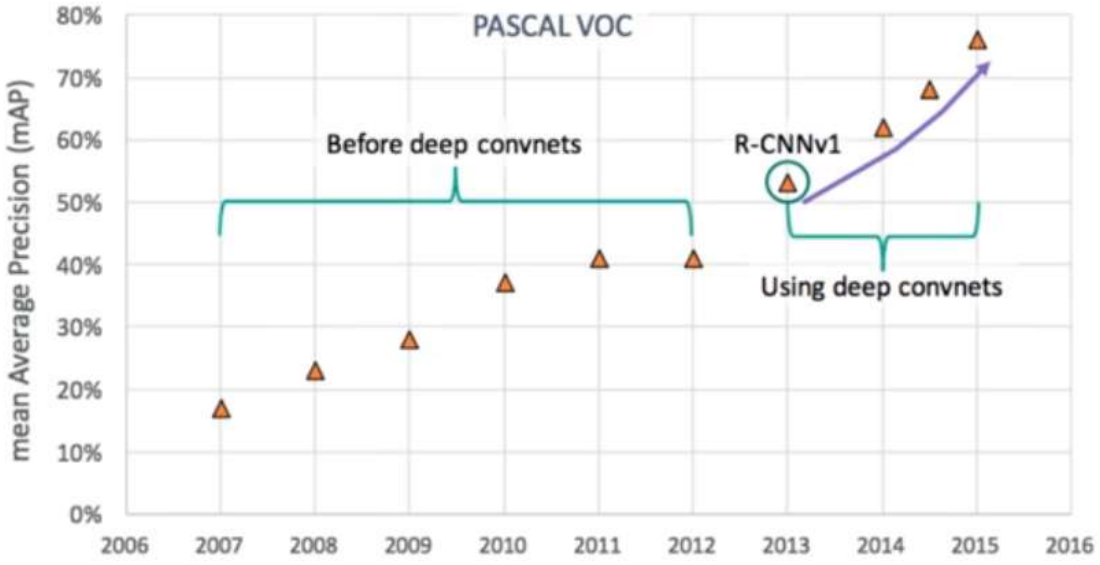


Ilustración 34: Comparativa entre el comportamiento de los algoritmos de detección de objetos antes y después de la llegada de las redes convolucionales (Lazy Programmer, 2018)

Las aplicaciones de este problema son muchas, en particular en el sector automovilístico. Es esencial para un coche que pretenda conducir solo, detectar todos los objetos relevantes en su

campo de visión, y hacer esta tarea en tiempo real. Actualmente el algoritmo que consigue esto se llama SSD, y se explicará cómo funciona en los siguientes apartados de este capítulo.

Por último, añadir que este capítulo es puramente teórico. Si en el mejor de los casos, la arquitectura que hemos conseguido acierta en poco más del cincuenta por ciento, con estos datos no se puede pretender entrenar una arquitectura que vaya más allá de clasificar una cara, detectar y clasificar todas las caras conocidas en una imagen dada.

5.2 Object Localization

Como su nombre sugiere, esta tarea no solo se trata de decir que objeto aparece en la imagen, que es la tarea propia de *Image Classification.* Además, tiene el añadido adicional de distinguir dónde está el objeto. Aunque no es nuestro propósito final, esta tarea es una aproximación intermedia entre la clasificación de imágenes y la detección de todos los objetos de una imagen dada.

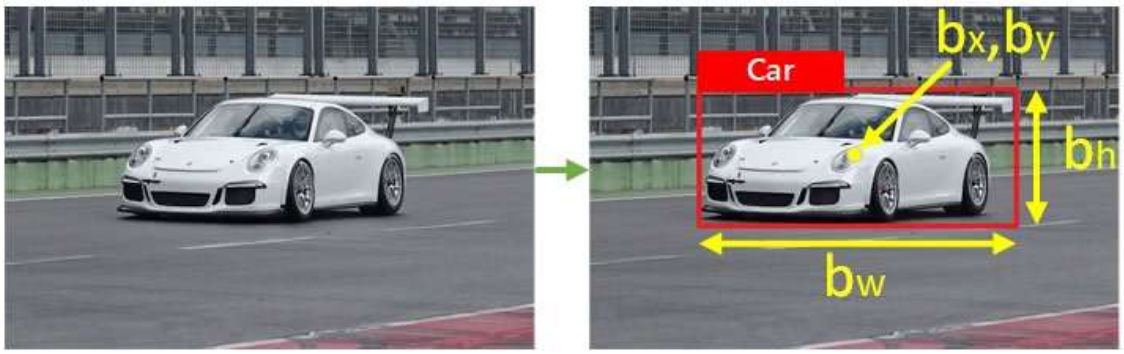


Ilustración 35: Comparativa de una imagen, antes y después de aplicar el algoritmo de Object Localization. Fuente: https://www.dlology.com/blog/gentle-guide-on-how-yolo-object-localization-works-with-keras/

Los modelos tratados hasta este punto del proyecto tenían como resultado una sola salida, la clase a la que pertenecía la imagen. Este resultado venía dado de hacer regresión logística a las características extraídas de la imagen por las capas convolucionales. Sin embargo, aunque hasta el momento el único uso que le hemos dado a la red neuronal es el de clasificación, una red neuronal puede tener como salidas números continuos, usando una regresión lineal en la última capa sobre con las características extraídas por las capas anteriores. Si se quiere consultar el curso en el que me he basado para desarrollar este contenido: (Lazy Programmer, 2018).

Para esta tarea tenemos una mezcla. Se usará una regresión logística en la última capa para determinar qué clase de objeto es el que aparece en la foto, y cuatro regresiones lineales para obtener cuatro valores continuos: los valores de las coordenadas x e y del centro de la imagen, la altura y la anchura del objeto. Con estos cuatro valores podemos dibujar un rectángulo como el mostrado en la imagen anterior, localizando al objeto cuya clase es “coche”.

La composición de la última capa de la red neuronal serán tantas neuronas como clases tenga el archivo de datos más cuatro neuronas que tendrán como salida los cuatro valores que hemos especificado antes. Estos dos grupos de neuronas son muy diferentes entre sí, ya que es distinta la función que cumplen. Como todas las neuronas de clasificación, la función final de activación

será la función *softmax* y la función de coste será la función *cross-entrophy* (Sección 2.2). Las cuatro neuronas que hacen una regresión lineal, utilizaran una función de activación lineal o ReLu, dependiendo si se contemplan valores negativos en la salida o no, y la función de coste propia de las regresiones lineales. MSE (Mean Square Error)

Como todas estas neuronas forman parte de la misma red, solo podemos tener una función de coste para ejecutar el algoritmo de *back propagation*. Está función de pérdida será una suma ponderada entre la función de pérdida de clasificar mal el objeto y la función de pérdida de localizarlo mal. El peso de las ponderaciones dependerá de la importancia que le demos a cada tarea.

5.3 Object Detection

La tarea de *Object Localization* aunque está más próxima de lo que andamos buscando, tiene una aplicación limitada. En un caso general, ya no solo en nuestro problema particular, sino para cualquier problema que implique el procesamiento inteligente de imágenes, identificar que objeto es el que aparece en la foto y localizarlo no parece resultar suficiente.

En un caso genera, pensemos por ejemplo en las imágenes captadas por la cámara de un coche, o en nuestro caso, imágenes de una película, el rango de objetos o personas a identificar va desde 0 hasta muchos, y en un caso ideal queremos que nos reconozca y nos localice a todos, y en el caso de que no haya ninguno, que no localice a ninguno. A esta tarea se la conoce como *Object Detection.* Para consultar el curso del que he sacado la información para este apartado: (Lazy Programmer, 2018).

5.3.1 Primera aproximación a la solución del problema

Si pensamos que tenemos un modelo que realiza la tarea *Object Localization* a la perfección, y recordamos que la salida de la función de activación *softmax* son decimales entre 0 y 1, dependiendo la probabilidad que estima el modelo de que el objeto esté en la foto para cada neurona de salida, una solución al problema puede ser un bucle en el que:

- Se le da al modelo la imagen de la que queremos detectar todos los objetos reconocibles en ella.
- Nos quedamos con la clase cuya salida es la que tiene mayor probabilidad de aparecer en la foto, y el marco en el que la ha localizado nuestro modelo.
- Comprobamos que la probabilidad de que aparezca la imagen es mayor a un límite que establecemos con anterioridad.
- Guardamos la salida como objeto detectado y pintamos el marco en el que ha localizado el modelo a la imagen de negro, para que el modelo no vuelva a localizar el mismo objeto.
- Le damos al modelo la imagen sin los objetos que ha detectado ya.
- Cuando la probabilidad de la clase ganadora no supere el límite que hemos establecido, supondremos que el modelo ya no es capaz de detectar ningún otro objeto y, por lo tanto, que ya no hay más objetos detectables.

5.3.2 Segunda aproximación al problema. *Sliding Windows*

La segunda manera de la que podemos afrontar nuestro problema es establecer un marco de manera que nos quedamos solo con un recorte de nuestra imagen original, vamos moviendo ese marco a través de la imagen, y con cada recorte que hacemos, se lo pasamos al modelo entrenado de *Object Localization* a ver qué es lo que encuentra y que probabilidad estima.

Al igual que en la aproximación anterior, nos quedaremos solo con las predicciones que superen cierto límite, pues el modelo siempre hace una predicción y es posible que no haya nada que localizar en muchos de los recortes.

El problema de este algoritmo salta a la vista. Aunque es capaz de detectar los objetos en nuestra imagen es demasiado lento. Para una sola imagen tendrá que hacer tantas predicciones como recortes hagamos en nuestra foto, y será incapaz de hacer predicciones a tiempo real.

Aunque esta solución no es práctica, la idea de ir identificando que hay en ciertas partes de la imagen nos tiene que haber resultado familiar. Es la idea que llevamos aplicando todo el proyecto, y es la tarea que está implementada con capas convolucionales de manera tan eficiente. (Ver sección 2.4.4.1)

5.3.3 Modificando la red neuronal detección de objetos.

Las arquitecturas con las que hemos trabajado en anteriores capítulos con demasiado complejas para esta explicación teórica. Por simplicidad imaginemos que tenemos la siguiente red convolucional:

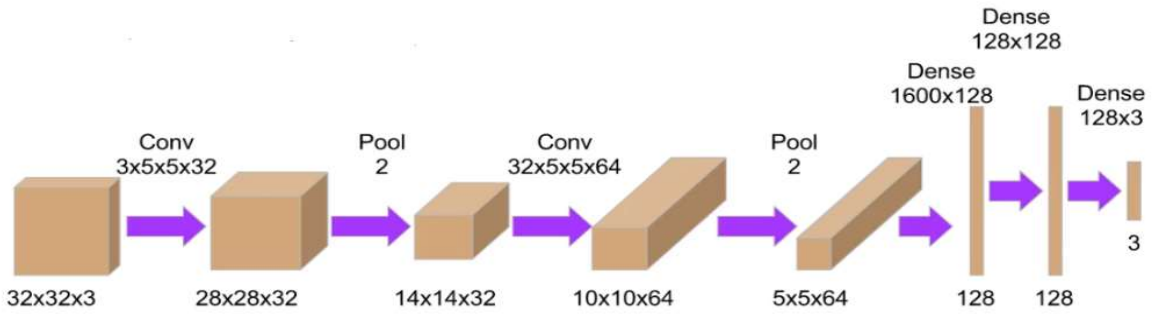


Ilustración 36: Esquema de la red convolucional que vamos a utilizar como ejemplo para la explicación de SSD (Lazy Programmer, 2018)

En el esquema anterior tenemos una red neuronal convolucional típica, esta vez con tamaños de filtros 5x5, intercalados con capas de *pooling* de tamaño dos, hasta llegar a las capas de redes neuronales normales que clasifican en tres clases distintas (el tamaño de la capa de salida es tres) imágenes de dimensiones 32x32. Hasta aquí todo normal.

El primer cambio que haríamos sería sustituir las capas de neuronas normales del final de nuestra red por capas de convolución de tamaño de filtro 1x1, como se muestra en la Ilustración 39. Si el tamaño de la capa anterior era de 128 neuronas, ahora tendremos 128 filtros de tamaño 1x1, es decir, el mismo número de parámetros y el mismo tamaño de la salida. Si la capa de salida se configura con una función de activación *softmax,* esta red será capaz de clasificar en 3 clases de una manera similar a como lo hacía la anterior.

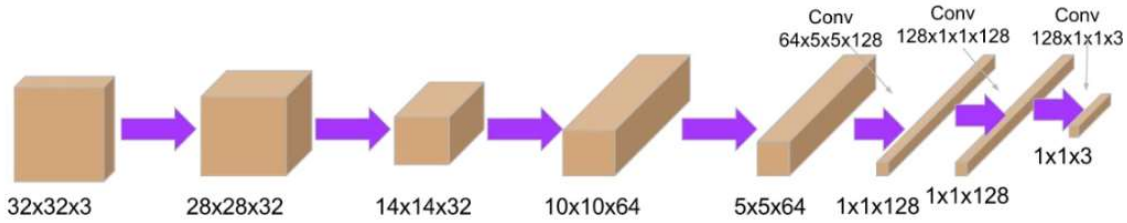


Ilustración 37: Esquema del ejemplo después de la modificación de las últimas capas de neuronas normales por capas de neuronas convolucionales (Lazy Programmer, 2018)

La pregunta es clara. ¿Por qué hacemos este cambio? Si recordamos el problema tratado en la sección 4.2.2 sobre la alternativa a redimensionar la imagen de entrada, decía que las capas convolucionales aceptan cualquier tamaño de entrada, como el tamaño de la propia red es independiente al tamaño del tamaño de la matriz de valores que le llega, puede funcionar con cualquier tamaño de entrada, la salida dependerá de ello, pero la capa funcionará igual.

El problema que nos encontrábamos en esa sección es que, a diferencia de las capas convolucionales, las capas de neuronas normales sí que son dependientes del tamaño de la entrada, cada valor de la entrada tiene que tener su neurona. Es por esta limitación por lo que decidimos prescindir de las capas de neuronas normales, en detrimento de más capas convolucionales 1x1.

¿Qué pasará entonces, si a nuestro nuevo modelo le metemos una imagen de mayores dimensiones? Vemos que pasa cuando le damos al modelo una entrada de dimensiones 36x36:

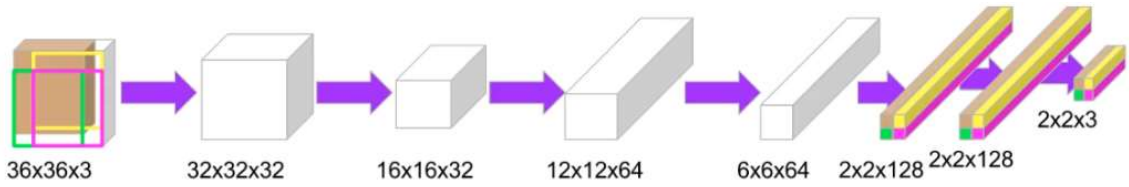


Ilustración 38: Ejemplo del paso de una entrada de dimensiones mayor en una red neuronal solo compuesta por capas convolucionales (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: Object Localization, 2018)

Como podemos ver en las dimensiones de las salidas según la entrada va pasando por el modelo, la salida final es una matriz 2x2x3. Debido a las características de los campos receptivos, cada una de las 4 submatrices 1x1x3 de la matriz final corresponderá a las probabilidades de cada clase de estar en cada una de las cuatro porciones de imágenes recortadas de la imagen inicial, que se muestran en el esquema con recuadros de distintos colores. Por lo tanto, el resultado será el mismo que aplicar cuatro veces el algoritmo *forward propagation* explicado en el

apartado anterior, una por cada marco 32x32, pero esta vez obtenemos las salidas de una sola ejecución del algoritmo. En esta propiedad se basa el algoritmo SSD, que voy a explicar a continuación.

5.4 Algoritmo SSD

Aunque ya hemos dado un paso muy grande en nuestro objetivo de detección de objetos en una imagen, no tenemos todavía la solución definitiva, el algoritmo SSD (Liu, y otros, 2016) afronta los siguientes dos problemas y los da solución:

5.4.1 Problemas en la escala.

Al observar esta imagen nos damos cuenta de que vamos a encontrar dificultades al encontrar un tamaño de ventana que nos permita detectar tanto al pájaro como a la rana, ya que, si elegimos un tamaño de ventana grande, la rana no será reconocible, y si elegimos uno pequeño, el modelo no verá al pájaro completo y no lo podrá detectar.

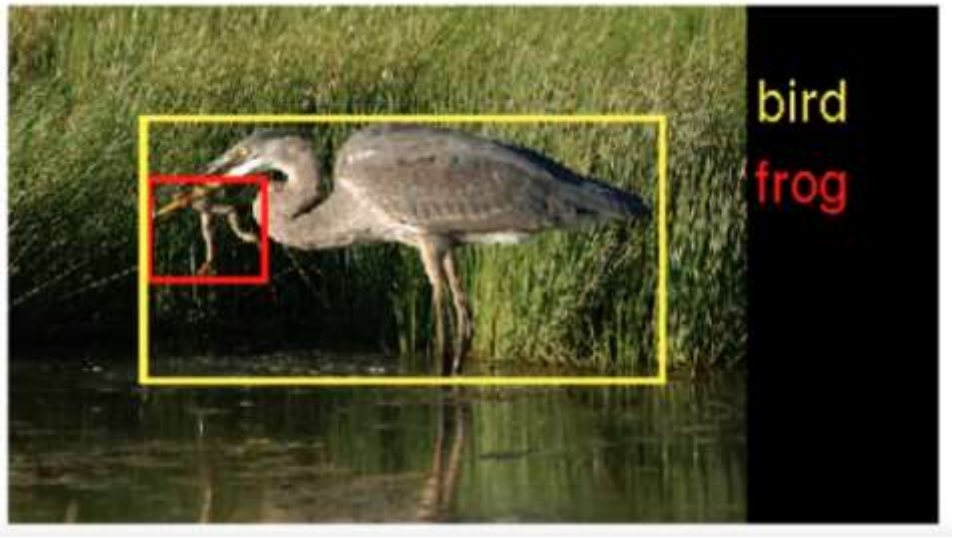


Ilustración 39: Diferencia entre tamaños de los marcos de localización entre la rana y el pájaro (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Sección 5: How would you find an object in an image?, 2018)

El problema es importante, un coche que pretenda conducir solo debe ser capaz de detectar los coches más cercanos, pero también los más lejanos a la cámara. Entonces, ¿cómo nos aseguramos que el enfoque de las ventanas es el correcto?

La idea que se describe en el *paper* del algoritmo SSD es seguir aprovechándonos de las propiedades de las redes convolucionales. La naturaleza de estas redes es ir de agrupando características de menor nivel, para ir extrayendo características de un nivel mayor, es decir, va desde ventanas muy pequeñas dentro de la imagen a ventanas mayores.

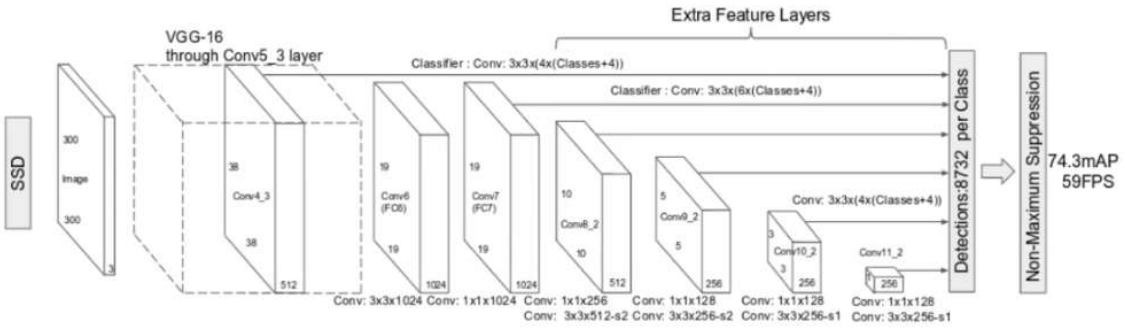


Ilustración 40: Esquema de implementación del algoritmo SSD sobre una red neuronal convolucional (Liu, y otros, 2016)

La solución esta presentada en este esquema, se trata de en vez de ver a la red neuronal entera como un extractor de características, tratar a cada capa como un extractor independiente y, además de pasarle a la siguiente capa las salidas de la capa anterior, usar la salida directamente en la parte final de nuestra red, es decir, tratar de distinguir un objeto en estas pequeñas ventanas antes de que se agrupen entre ellas en siguientes capas convolucionales.

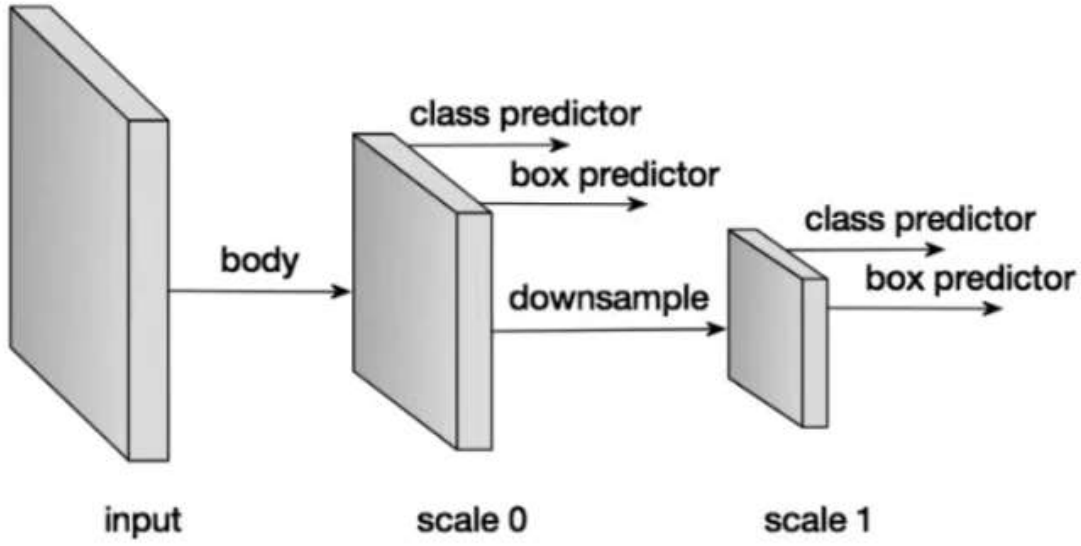


Ilustración 41: Esquema de una parte de una red neuronal convolucional en la que se ha implementado el algoritmo SSD (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Sección 5: How would you find an object in an image?, 2018)

Este esquema podría ser una parte de nuestra arquitectura SSD, vemos que sigue el modelo clásico de una red convolucional, pero que además de transmitir la salida a las siguientes capas, se trata de predecir la clase y los límites de cada objeto con dicha salida. Es decir, no es necesario llegar al final de la red para obtener la clasificación, conforme la entrada avanza por la red se van obteniendo clasificadores. Si el valor de la probabilidad de la clase ganadora en cualquiera de estas capas es mayor a un límite, habremos detectado un objeto en la imagen, y como lo hacemos en múltiples capas, estaremos trabajando cada vez a una escala distinta por lo que siempre tendremos una escala correcta para cada tamaño de objeto dentro de la foto.

Como lo único que necesitamos es salidas de capas convolucionales, que vayan desde un tamaño mayor de entrada a uno menor, que es básicamente lo que tienen en común todas las arquitecturas con las que hemos trabajado hasta ahora, no hay una arquitectura SSD propiamente dicha, sino que se puede modificar cualquier arquitectura de las antes estudiadas para que sea capaz de detectar objetos, y aprovecharnos así de arquitecturas que ya sabemos previamente que funcionan mejor con nuestro problema, como sería en nuestro caso la VGG, o trucos como el de *Transfer Learning* para no tener que entrenar nuestra red desde cero.

5.4.2 Problema con la forma

El último problema que está contemplado en el *paper* del SSD es el siguiente. Hasta ahora hemos trabajado con filtros cuadrados, ya sean 2x2, 3x3 o 5x5. Mirando esta imagen, ¿estamos trabajando con la forma adecuada en dichos filtros?

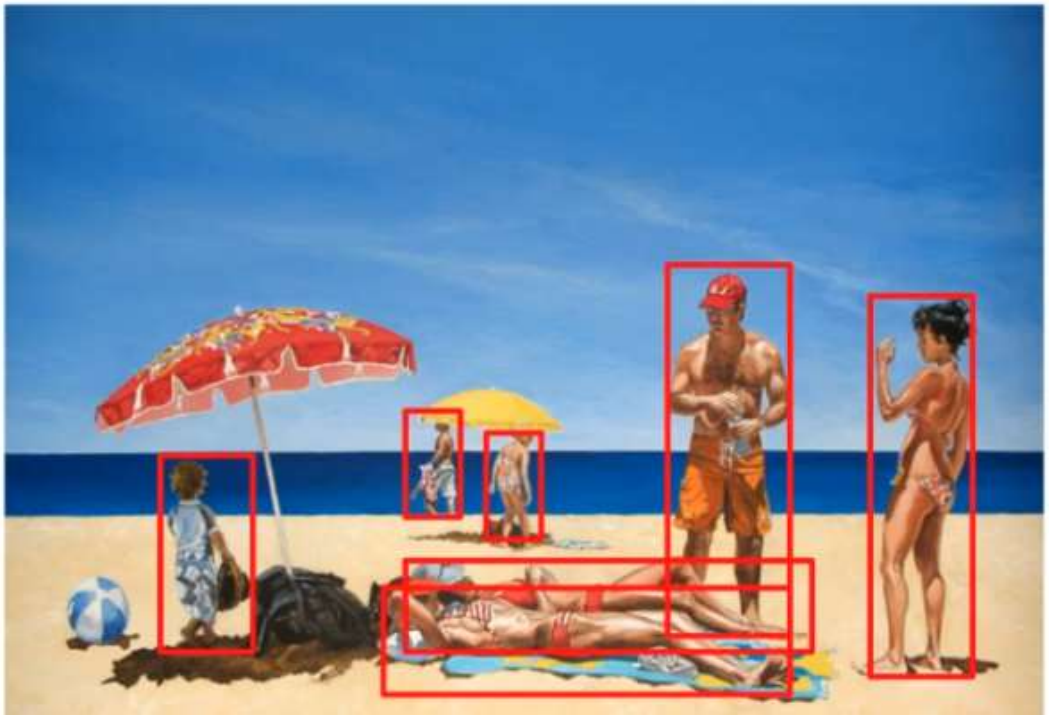


Ilustración 42: Imagen que ilustra el problema de la forma del filtro (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Sección 5: How would you find an object in an image?, 2018)

Si queremos buscar personas en la imagen anterior, vemos que la forma que ocupan estas personas puede ser rectangular vertical, o rectangular horizontal, pero en ningún caso cuadrada. Que filtro elegimos, ¿el vertical o el horizontal? Lógicamente, al igual que el problema de las escalas, lo ideal sería no tener que elegir, sino tener una forma de filtro ideal para cada objeto, ya sea cuadrada, vertical u horizontal.

La solución que proponen los desarrolladores del algoritmo es no utilizar una forma de filtro definida, sino cuatro. Para ser más exactos, los cuatro que se muestran en la Ilustración 45:

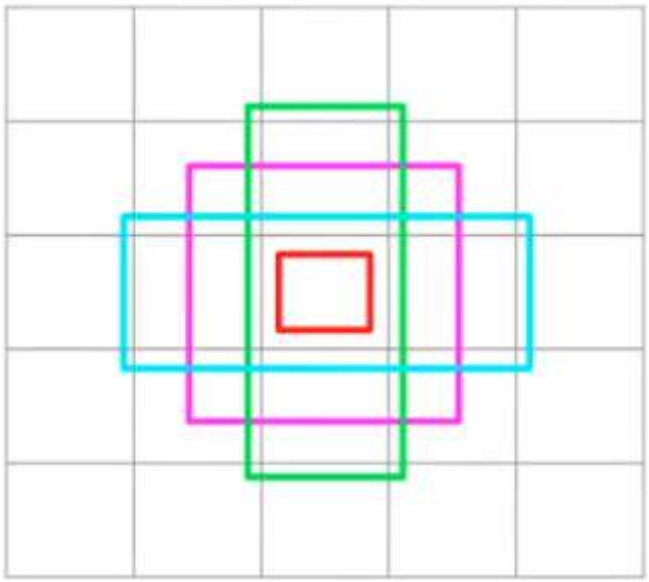


Ilustración 43: Distintos tipos de cajas propuestas por los desarrolladores del algoritmo SSD (Lazy Programmer, Udemy. Deep Learning: Advanced Computer Vision. Sección 5: How would you find an object in an image?, 2018)

Aquí podemos ver las cuatro diferentes formas de filtros que se proponen en el *paper,* una vertical, una horizontal y dos cuadradas de diferentes tamaños. La idea es que, para cada ventana, comprobar si tenemos un objeto dentro de estas cuatro cajas predefinidas. Puntualizando, sería el comportamiento contrario a la solución propuesta a la tarea de localización de objetos tratada antes en este capítulo. En ella, encontrado un objeto, tratamos de sacar el marco en el que está localizado mediante regresión lineal. En este caso, le damos al modelo el marco y este tendrá que decidir si hay un objeto o no dentro del mismo. De esta manera, con estos marcos predefinidos, siempre habrá un marco dentro de una escala que se ajuste casi a la perfección al objeto que estamos tratando de detectar. Si se quiere consultar el curso en el que me he basado para el desarrollo del contenido de este capítulo: (Lazy Programmer, 2018)

Capítulo 6.

Conclusiones y Trabajo Futuro

6.1 Relación con los contenidos del grado.

A pesar del nombre tan rocambolesco de “redes neuronales” hemos podido comprobar que no es un modelo matemático que se basa en la sucesión de combinaciones lineales y funciones de activación. Por semejanza, dominar metodologías de análisis de regresión como regresión logística y regresión lineal ayudan a entender cómo funciona una red neuronal.

Evidentemente este proyecto forma parte de la rama de la estadística de Análisis de Datos. Para desarrollar un buen modelo es necesaria la metodología aprendida en las asignaturas de esta rama sobre como validar un modelo, distintos algoritmos de validación, que errores incluir en las métricas y la diferencia entre ellos, que problema está asociado a cada tipo de error y conceptos como el de *overfitting.*

6.2 Objetivos alcanzados.

Al final de la ejecución de este proyecto puedo confirmar el cumplimiento de una serie de los objetivos iniciales:

- Se ha verificado la viabilidad de desarrollar una aplicación que realice la tarea de la detección y reconocimiento de actores en imágenes extraídas de archivos de vídeo, usando herramientas de Deep Learning.
- Se ha entrenado dos tipos de arquitecturas complejas convolucionales, y se ha concluido que una de ellas se comporta mucho mejor que la otra en el reconocimiento facial.
- Se ha llegado a la conclusión de que el algoritmo SSD ofrece la mejor solución a nuestra tarea de detección de actores.
- Se ha establecido como base una arquitectura VGG16 con el fin de implementar el algoritmo SSD sobre ella.

6.3 Trabajo futuro.

Para las redes neuronales, como en cualquier modelo estadístico que queremos desarrollar, la parte central son los datos. Simultáneamente al desarrollo de este proyecto sobre la investigación de la viabilidad de utilizar herramientas de Deep Learning para resolver el problema mencionado, la empresa interesada en el desarrollo de dicha aplicación, Copyright Management & Solutions ha estado recopilando imágenes de los actores que se quiere reconocer de las películas. Disponiendo de este mayor número de imágenes por persona, e imágenes de más calidad, es decir, mejores datos estamos seguros de poder entrenar un modelo con mejores resultados, capaz de soportar la implementación del algoritmo SSD sobre él.

Además, para el desarrollo de la herramienta se contará con la posibilidad de realizar computación en la red, pudiendo entrenar la red neuronal de una manera más rápida y exhaustiva que la que he podido desarrollar en mi ordenador personal.

La continuación natural de este proyecto es el desarrollo de la aplicación.

Bibliografía

Calonge, T., & Alonso, C. J. (2018). *Apuntes de la asignatura: "Técnicas de Aprendizaje Automático".* Valladolid: Departamento de Informática, Escuela de ingeniería Informática.

CISCO. (27 de Febrero de 2017). *Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper*. Obtenido de https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

Deeplearning.ai. (2017). *Coursera. Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization. Week 3. Batch Normalization.* Obtenido de https://www.coursera.org/learn/deep-neural-network/lecture/4ptp2/normalizing-activations-in-a-network

Deeplizard. (22 de noviembre de 2017). *Youtube. Fine-tune VGG-16 Image Classifier with Keras* . Obtenido de https://www.youtube.com/watch?v=oDHpqu52soI&t=21s

Fischler, M. A., & Elschlager, R. A. (1973). *The Representation and Matching of Pictorial Structures.* IEEE Computer Society Washington, DC, USA.

García, L. Á., & Fernández, M. A. (2016). *Apuntes de la asignatura "Análisis de datos".* Valladolid: Departamento de estadística e Investigación Operativa.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Resiudal Learning for Image Recognition .

Hebb, D. (1949). The Organization of Behavior.

Hebb, D. O., & Penfield, W. (1940). Human behaviour after extensive bilateral removal from the frontal lobes.

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks.

Lazy Programmer. (2018). *Udemy. Deep Learning: Advanced Computer Vision. Sección 3: Transfer Learning.* Obtenido de https://www.udemy.com/advanced-computer-vision/learn/lecture/9339958#announcements

Lazy Programmer. (2018). *Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: Object Detection.* Obtenido de https://www.udemy.com/advanced-computer-vision/learn/lecture/9346970#content

Lazy Programmer. (2018). *Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: Object Localization.* Obtenido de https://www.udemy.com/advanced-computer-vision/learn/lecture/9346968#content

Lazy Programmer. (2018). *Udemy. Deep Learning: Advanced Computer Vision. Seccion 4: ResNet.* Obtenido de https://www.udemy.com/advanced-computer-vision/learn/lecture/9339954#questions

Lazy Programmer. (2018). *Udemy. Deep Learning: Advanced Computer Vision. Sección 5: How would you find an object in an image?* Obtenido de https://www.udemy.com/advanced-computer-vision/learn/lecture/9346974#content

LeCun, Y., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition.

Liu, W., Anguelov, D., Dumitru, E., Christian, S., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector.

Lowe, D. G. (1999). *Object Recognition from Local Scale-Invariant Features.* Computer Science Department, University of British Columbia.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* MIT.

McCulloch, W. S., & Pitts, W. H. (1943). A logical calculus of the ideas immanent in nervous activity.

National Research University Higher School of Economics. (s.f.). *Coursera. introduction to Deep Learning. Week 3: Motivation for convolutional layers.* Obtenido de https://www.coursera.org/learn/intro-to-deep-learning/home/welcome

National Research University Higher School of Economics. (s.f.). *Coursera. Introduction to Deep Learning. Week 3: Our first CNN architecture.* Obtenido de https://www.coursera.org/learn/intro-to-deep-learning/lecture/9y2lf/our-first-cnn-architecture

Roberts, L. (1960). Blocks World.

Romero, J. J., Dafonte, C., Gómez, Á., & Penousal, F. J. (2007). *Inteligencia Artificial y Computación Avanzada.* Santiago de Compostela: Publicaciones Fundación Alfredo Brañas.

Shi, J., & Malik, J. (2000). Normalized Cuts and Image Segmentation.

Simonyan, K., & Zisserma, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition.

Standford University; Princeton University. (2006). *ImageNet*. Obtenido de http://www.image-net.org/

Stanford University School of Engineering. (2017). *Youtube. Lecture Collection. Convolutional Neural Network for Visual Recognition. CNN Architectures.* Obtenido de https://www.youtube.com/watch?v=DAOcjicFr1Y&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=9

Stanford University School of Engineering. (2017). *Youtube. Lecture Collection. Convolutional Neural Network for Visual Recognition. Introduction to Convolutional Neural Network for Visual Recognition.* Obtenido de

https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

Turing, A. (1950). Computing machinery and intelligence.

West, J. (2007). A Theoretical Foundation for Inductive Transfer.

Wikipedia. (25 de abril de 2019). *Wikipedia. Convoluxión.* Obtenido de https://es.wikipedia.org/wiki/Convoluci%C3%B3n