

UNIVERSIDAD DE SEVILLA

TRABAJO FIN DE MÁSTER

Reconocimiento de peces usando aprendizaje profundo

Autor:

Marco HERRERO SERNA

Tutor:

Álvaro ROMERO JIMÉNEZ

Máster Universitario de Lógica, Computación e Inteligencia Artificial

Departamento de Ciencias de la Computación e Inteligencia Artificial

4 de septiembre de 2017

Resumen

En este trabajo se detallan las soluciones presentadas a la competición *The Nature Conservancy Fisheries Monitoring* de la plataforma de competiciones predictivas *Kaggle*. La competición consiste en detectar si aparecen y clasificar diferentes especies de grandes peces en imágenes de barcos pesqueros con el objetivo de analizar el comportamiento de estos.

Las soluciones exploran el uso de arquitecturas de aprendizaje profundo, usando redes neuronales convolucionales preentrenadas para otras competiciones de reconocimiento de imágenes mucho más generalistas junto con arquitecturas diseñadas para este problema.

La solución final presentada en este trabajo consigue una medalla de bronce en la competición, clasificándose en el mejor 10 % de los participantes.

Índice general

Resumen	III
1. Introducción	1
1.1. Aprendizaje automático	2
1.2. <i>Kaggle</i>	2
1.3. Reconocimiento de imágenes	3
1.4. Reconocimiento de imágenes	3
2. Conceptos básicos	5
2.1. Redes neuronales	5
2.1.1. Propagación hacia atrás	5
2.1.2. Codificación <i>onehot</i>	6
2.2. Imágenes digitales	7
2.3. Filtros convolucionales	8
2.4. Redes neuronales convolucionales	9
2.4.1. Estructura de una capa convolucional	10
Capa convolucional - CONV (<i>Convolutional layer</i>)	10
Capa de activación - RELU (<i>Rectifier Linear layer</i>)	11
Capa de muestreo - POOL (<i>Pooling layer</i>)	11
Capa densa - FC (<i>Fully connected layer</i>)	12
2.4.2. Arquitectura	12
3. Definición del problema	13
3.1. <i>The Nature Conservancy Fisheries Monitoring</i>	13

3.2. Definición del problema	13
3.2.1. Datos	14
3.2.2. Envío de la solución y evaluación	15
3.2.3. Tablas de clasificación y fases de la competición	15
3.2.4. Conjunto de entrenamiento	16
3.2.5. Conjuntos de test	17
4. Soluciones presentadas	19
4.1. Metodología	19
4.1.1. Partición del conjuntos de datos	19
4.1.2. Evaluación del modelo	20
4.1.3. Software	20
4.2. Arquitectura de los modelos	20
4.2.1. Red convolucional	21
4.2.2. Ajuste fino	22
4.3. Modelo completamente conectado	24
4.3.1. Análisis del sobreajuste	25
4.3.2. <i>Dropout</i>	26
4.3.3. Normalización por lotes	28
4.3.4. Aumento de datos	30
4.4. Modelo convolucional	33
4.4.1. Visualización del modelo	34
4.5. Modelo con entrada múltiple	36
4.5.1. Fuga de datos	39
4.6. Salida múltiple	39
4.6.1. Arquitectura del modelo	40
4.6.2. Visualización de la localización de peces	43
4.7. Modelos <i>ensemble</i>	43
4.7.1. Resultados	44

4.8. Modelos finales presentados	45
5. Conclusiones	47
5.1. Kaggle	48
5.1.1. Puntuación final	48
5.2. Siguientes pasos	48
A. Infraestructura	49
A.1. Requisitos	49
A.2. Hardware	49
B. Keras	51
B.1. Instalación	51
B.2. Uso	51
B.2.1. Arquitectura	52
B.2.2. Carga de datos	53
B.2.3. Entrenamiento y evaluación	53
Bibliografía	55

Índice de figuras

1.1.	Mapa que muestra casos de contagio del Córula, representando cada contagio con una barra en el edificio donde se produjo	1
2.1.	Representación de una red neuronal con dos entradas, una capa oculta con dos neuronas y una salida.	6
2.2.	Imagen parcial en blanco y negro de un dígito escrito a mano	7
2.3.	Matriz que representa los valores de cada punto en la imagen de la figura 2.2	7
2.4.	Imagen de un dígito escrito a mano, sacado de la base de datos MNIST	8
2.5.	Aplicación de diferentes filtros a la imagen de la figura 2.4	9
2.6.	Ejemplo de dos representaciones de la misma red convolucional. Abajo el modelo resumido con cajas.	10
2.7.	Evolución de un entrenamiento para una red convolucional de cuatro capas usando ReLUs (línea sólida) vs usando tanh (línea discontinua) (Krizhevsky, 2012)	11
2.8.	<i>Max-pooling</i> sobre una imagen de 4×4	12
2.9.	Ejemplo de arquitectura de una red convolucional para clasificación, (Russakovsky, 2014)	12
3.1.	Ejemplo de imágenes tomadas en barcos pesqueros para su posterior identificación	13
3.2.	Especies de peces a clasificar en el reto	14
3.3.	Cantidad de imágenes pertenecientes a cada categoría	16
3.4.	Dos imágenes del conjunto de entrenamiento casi idénticas	17
4.1.	Arquitectura de una red convolucional profunda. Consta de una acumulación de capas convolucionales destinadas a extraer características de la entrada y una serie de capas densas para clasificar la entrada usando dichas características.	21
4.2.	Arquitectura de VGG	22

4.3. Estructura del modelo final. Como recibe como entrada imágenes de 244 píxeles, pasa por las diferentes redes y devuelve el vector de ocho categorías.	23
4.4. Arquitectura del modelo completamente conectado	25
4.5. A la izquierda una red neuronal estándar con dos capas ocultas. A la derecha la misma red aplicando un <i>dropout</i> en cada una de las capas ocultas. Las neuronas tachadas han perdido su activación.	26
4.6. Evolución de la puntuación de un modelo usando <i>dropout</i>	27
4.7. Imagen del conjunto de datos	31
4.8. Cuatro aumentos diferentes de la imagen, reescalada a 224×224 píxeles	31
4.9. Arquitectura de la red completamente convolucional	34
4.10. Una imagen correspondiente a la clase ALB: <i>Thunnus alalunga</i>	35
4.11. Aplicación del mapa de calor reescalado a la imagen de la figura 4.10 .	35
4.12. Una imagen correspondiente a la clase ALB: <i>Thunnus alalunga</i>	36
4.13. Aplicación del mapa de calor reescalado a la imagen 4.10	37
4.14. Arquitectura del modelo con entrada múltiple	38
4.15. Representación de un rectángulo del conjunto de datos auxiliar que ubica cada pez en su imagen	41
4.16. Arquitectura del modelo denso con salida múltiple	41
4.17. En rojo, el rectángulo correspondiente a la localización del pez en el conjunto de datos alternativo. En verde, el rectángulo generado por el modelo con salida múltiple.	43

Capítulo 1

Introducción

En 1850 se creía que el Córera, una enfermedad intestinal contagiosa y mortal, era propagado por aire. Esta creencia se refutó en 1854 cuando un médico inglés, John Snow, dibujó en un mapa todos los casos de contagio ocurridos en el *Soho* de Londres (figura 1.1). El mapa mostraba que la mayoría de casos ocurrían en las cercanías de un pozo de agua específico. La epidemia cesó cuando, por recomendación de Snow, se cerró este pozo (Snow, 1855).

Este es uno de los primeros ejemplos de un problema resuelto gracias al análisis de datos. Una de las ventajas de las nuevas tecnologías es la gran cantidad de datos que son capaces de generar para problemas específicos. Esto abre nuevas posibilidades para resolver problemas que antes eran impensables. Sin embargo, de la misma manera que alguien tuvo que pintar todos esos puntos en el mapa de John Snow, los datos deben ser tratados para poder extraer conocimiento de ellos.

Una gran cantidad de datos requiere una gran cantidad de recursos humanos para extraer conocimiento, sobre todo cuando los datos son tan complejos como reconocer una voz en un fichero de audio, entender la intención de un texto o detectar un objeto en una imagen.

FIGURA 1.1: Mapa que muestra casos de contagio del Córera, representando cada contagio con una barra en el edificio donde se produjo.



1.1. Aprendizaje automático

El aprendizaje automático es un campo de las ciencias de la computación cuyo objetivo es crear sistemas que sean capaces de realizar determinadas tareas mejorando mediante la experiencia, sin necesitar que se le diga explícitamente como realizarlas (Mitchell, 1997).

Usando este tipo de sistemas es posible enseñar a una máquina a extraer conocimiento de diferentes datos si es capaz de ver ejemplos anteriores. Por ejemplo, puede ser capaz de reconocer peces en una imagen si se le provee de ejemplos resueltos.

Conseguir sistemas de este tipo hace que se puedan resolver problemas cuya solución no estaba al alcance de nadie que no tuviese grandes cantidades de recursos.

1.2. *Kaggle*

En el año 2007, *Netflix* creó una competición para mejorar su motor de recomendación de películas. Publicó un conjunto de datos con las películas elegidas por los diferentes usuarios y pidió a los participantes realizar un modelo predictivo usando este conjunto de datos. Evaluaron los modelos predictivos presentados usando un conjunto de datos diferente y otorgaron el premio al que menos error presentaba. El premio para el mejor modelo predictivo era de un millón de dólares.

En 2010 el economista australiano Anthony Goldbloom, inspirándose en la idea de la competición de *Netflix*, creó *Kaggle*. Esta es una plataforma de competiciones de análisis y modelización predictiva de datos.

La plataforma funciona de la siguiente manera: un equipo promotor contacta con *Kaggle* y prepara un conjunto de datos para la competición. Un subconjunto de esos datos son publicados en la web para el uso de los participantes. El otro subconjunto, oculto, se usa para determinar el mejor modelo presentado y declarar un ganador.

El conjunto de datos presentado contiene las respuestas de cada elemento (permitiendo así que los modelos usen aprendizaje supervisado). Junto a los datos se especifica una métrica que servirá para medir la bondad del modelo.

Para presentar el modelo el participante debe presentar las predicciones del conjunto de test en la web de *Kaggle*. La puntuación obtenida según la métrica sobre una parte del conjunto de test se hará pública en la tabla de posiciones pública. Cuando la competición termina se genera una tabla de posiciones privada, donde se publica la puntuación del modelo usando el conjunto de test completo y es con esta puntuación con la que se determina el ganador. Se hace de esta manera para evitar que los participantes sobreajusten el modelo al conjunto de test.

Existen varios tipos de competiciones a las que los usuarios se pueden presentar:

- **Getting started:** Competiciones sencillas organizadas por la misma plataforma con la finalidad de que los usuarios se familiaricen con los modelos predictivos y

la estructura de las competiciones. Suelen tener un conjunto de datos ordenado y sencillo, y existe mucha documentación y ayuda en los foros.

- **Playground:** Competiciones que no resuelven problemas reales, diseñadas para que los usuarios experimenten y prueben diferentes ideas. No tienen ningún premio.
- **Research:** El objetivo de estas competiciones es la investigación sobre diferentes técnicas o de problemas de bien común. Muchas de estas competiciones tienen como premio la invitación a conferencias o a publicar los resultados en algún medio. Suelen requerir que las soluciones sean publicadas como código abierto.
- **Recruiting:** Los organizadores de estas competiciones buscan contratar a los mejores participantes.
- **Featured:** Las competiciones con más dificultad y mejores premios. Buscan resolver un problema real y posiblemente comercial. Aquí entraría la competición de *Netflix*.

A principios de 2017 *Kaggle* fue comprado por *Google*, aunque por ahora sigue siendo una plataforma independiente funcionando bajo su propio nombre.

1.3. Reconocimiento de imágenes

Uno de los grandes problemas dentro del aprendizaje automático es el reconocimiento de objetos en imágenes. Detectar diferentes objetos y situaciones de una imagen tal y como lo hace un humano permitiría automatizar muchísimos procesos, desde la conducción autónoma hasta la digitalización de libros.

El desarrollo de redes neuronales convolucionales en arquitecturas de aprendizaje profundo está consiguiendo precisiones similares a la humana en problemas de reconocimiento visual (Taigman y col., 2014). Esto hace que sea cada vez más interesante aplicar este tipo de soluciones a problemas de clasificación visual donde antes no era posible.

1.4. Reconocimiento de imágenes

La competición de *Kaggle* que se quiere resolver aquí tiene como objetivo clasificar peces dentro de imágenes de barcos pesqueros. Esto permitirá en un futuro predecir patrones migratorios, estudiar patrones de pesca o vigilar que la pesca sea una actividad que no dañe al medio ambiente. Sin embargo es necesario encontrar un sistema que permita distinguir diferentes peces tal y como lo haría un humano.

En este trabajo se intentarán aplicar técnicas de aprendizaje profundo para resolver este problema, variando arquitecturas y usando redes preentrenadas.

Capítulo 2

Conceptos básicos

2.1. Redes neuronales

Las redes neuronales son modelos computacionales basados en las conexiones neuronales que ocurren dentro del cerebro. Están compuestas de neuronas artificiales (también llamadas nodos o unidades) conectadas a través de conexiones dirigidas. Cada conexión tiene un peso numérico que determina la fuerza de la conexión.

Las neuronas artificiales, al igual que las biológicas, se activan cuando reciben un determinado estímulo. Para determinar si una neurona artificial se activa primero hace la suma de sus entradas ponderadas con los pesos de las conexiones y luego aplica una función de activación para producir la salida.

Existen dos categorías principales de estructuras de redes neuronales: **redes con alimentación-hacia-delante** o **redes recurrentes**, dependiendo de si el grafo que representa las conexiones de la red es acíclico o cíclico.

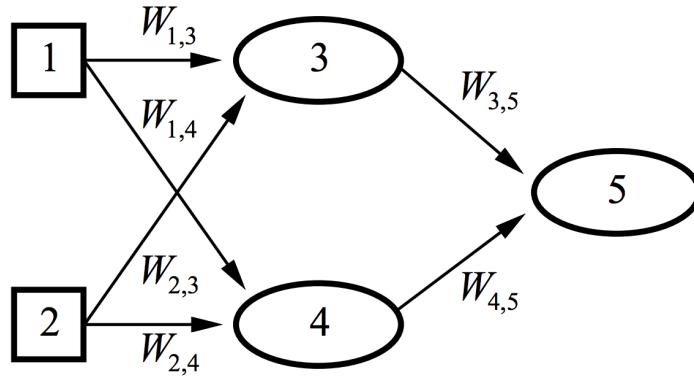
En este trabajo se van a usar redes con alimentación-hacia-delante. Este tipo de redes representa una función de sus estados actuales, por lo que no tiene otro estado interno que no sea el de sus propios pesos. Están organizadas en capas de forma que cada neurona recibe entradas únicamente de las neuronas de la capa que la precede. En la figura 2.1 se puede apreciar una red neuronal con dos entradas, una capa oculta (que no tiene neuronas de entrada ni de salida) con dos neuronas y una salida, así como los pesos para cada conexión.

La idea de las redes neuronales es que sean capaces de aprender funciones generalizando ejemplos de entradas y salidas de la función. Un algoritmo de aprendizaje para redes neuronales deberá ajustar los pesos de la red de tal manera que se minimice alguna medida del error producido sobre el conjunto de entrenamiento (Russell y Norving, 2004).

2.1.1. Propagación hacia atrás

El algoritmo usado para entrenar la red se llama propagación hacia atrás. Se trata de actualizar los pesos de la red para reducir el error, entendido como la diferencia entre la salida de la red con la salida esperada.

FIGURA 2.1: Representación de una red neuronal con dos entradas, una capa oculta con dos neuronas y una salida.



Para actualizar los pesos se usa el descenso del gradiente. Cada peso que lleva a una neurona de salida se actualiza de la siguiente manera:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

Siendo W_j el peso, α la tasa de aprendizaje, Err el error cometido en la salida, g' la derivada parcial de la función de activación y x_j la entrada. Si el error es positivo, la salida de la red es demasiado pequeña y por ello los pesos se incrementan para las entradas positivas y se decrementan para las entradas negativas. Cuando el error es negativo ocurre lo contrario.

Para actualizar los pesos de las neuronas ocultas se define una función de actualización de pesos. Es la misma con la que se actualizan los pesos de la capa final pero ahora se entiende que cada nodo oculto es responsable de una fracción del error en cada nodo de salida, proporcional al peso que tenga la conexión entre ambos nodos. Así que los valores se propagan hacia atrás con la fuerza de la conexión entre el nodo de salida y el nodo oculto.

2.1.2. Codificación *onehot*

La codificación *onehot* es la representación de variables categóricas como un vector binario. Cada elemento del vector representa la pertenencia a una de las categorías disponibles en el problema. El vector final tendrá el mismo tamaño que la cantidad de categorías y estará completo de ceros, excepto en el lugar perteneciente a la categoría del elemento, donde contendrá un uno.

Esta codificación es útil para redes neuronales, ya que al trabajar con varias categorías es más eficiente tener una neurona de salida para cada categoría disponible, en vez de trabajar con combinaciones de menos neuronas.

FIGURA 2.2: Imagen parcial en blanco y negro de un dígito escrito a mano

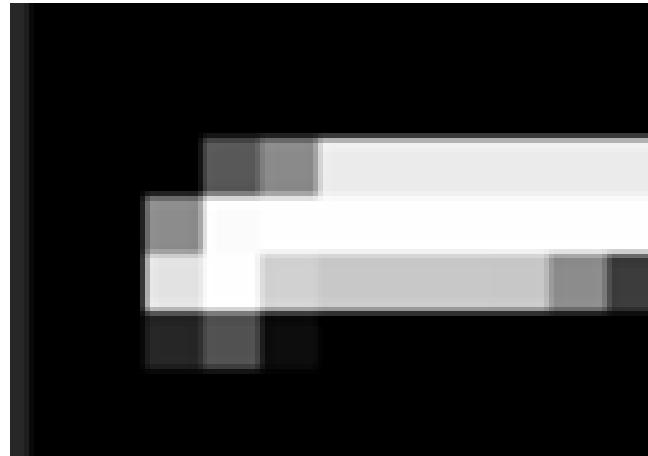


FIGURA 2.3: Matriz que representa los valores de cada punto en la imagen de la figura 2.2

```
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0.3529, 0.5412, 0.9216, 0.9216, 0.9216, 0.9216, 0.9216 ]
[ 0. , 0. , 0.549 , 0.9843, 0.9961, 0.9961, 0.9961, 0.9961, 0.9961, 0.9961 ]
[ 0. , 0. , 0.8863, 0.9961, 0.8157, 0.7804, 0.7804, 0.7804, 0.7804, 0.5451 ]
[ 0. , 0. , 0.149 , 0.3216, 0.051 , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
[ 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]
```

2.2. Imágenes digitales

Una imagen en el ámbito digital se entiende como una matriz de puntos. Cada uno de estos puntos puede ser interpretado como un número real entre 0 y 1, que representa la localización de este punto en la escala de grises. En esta representación usaremos valores menores para los puntos más oscuros y valores mayores para puntos más claros.

Un ejemplo de esa representación es la matriz de la figura 2.3, que representa la imagen de la figura 2.2.

En imágenes en color se suele usar el sistema *RGB* (*Red, Green, Blue*) para representar diferentes colores en cada punto de la imagen. En vez de usar un solo valor por punto se usan tres valores, cada uno representando la intensidad del color rojo, verde y azul en ese punto. Superponiendo estos tres valores se consigue el color final. Por lo tanto, las imágenes *RGB* constan de tres matrices, cada una con los valores reales entre 0 y 1 para cada color.

Para simplificar, algunos ejemplos de esta memoria van a usar una escala de grises, pero son aplicables a imágenes en color aplicando las operaciones a las tres diferentes matrices al mismo tiempo.

FIGURA 2.4: Imagen de un dígito escrito a mano, sacado de la base de datos MNIST



2.3. Filtros convolucionales

Al trabajar con esta interpretación de lo que es una imagen, se pueden usar operaciones sobre la matriz de la imagen para transformarla de diferentes maneras.

Consideremos por ejemplo la siguiente matriz 3×3 (llamada matriz filtro de convolución):

$$F = \begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Se puede usar F como un filtro para una imagen de la siguiente manera: primero, superponemos la matriz en algún punto de la imagen. Esto modificará el pixel donde ha quedado colocado el valor central de la matriz F . Para ello multiplicamos cada uno de los valores superpuestos, sumamos los resultados y los sustituimos en el valor central. Esto se hace para cada píxel de la imagen original (superponer el filtro en ese píxel y sustituir el valor por la operación).

En los bordes de la matriz existen menos píxeles adyacentes a cada píxel: seis en los bordes y tres en las esquinas. Para que la operación se haga con la misma cantidad de elementos se presupone que todos los valores que no existen son 0.

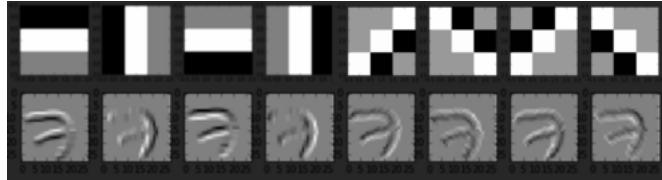
En el caso de la matriz F , la fila superior son todos valores negativos, la intermedia son todos 1 y la inferior todos ceros. Si aplicamos entonces la operación descrita sobre una imagen, los píxeles más brillantes (aquellos con mayor valor) en la imagen resultante serán los que su fila superior es cero, eliminando los valores negativos y su fila intermedia es 1.

Esto ocurrirá con más frecuencia en los bordes superiores de objetos claros con fondo oscuro.

Para mostar la utilidad de los filtros, los aplicaremos a la imagen de un dígito escrito a mano, sacado de la base de datos MNIST (LeCun y Cortes, 2010), que incluye 70000 imágenes de dígitos alfanuméricos escritos a mano.

Si aplicamos el filtro F a la imagen de la figura 2.4 podemos observar cómo resalta en blanco los bordes superiores y en negro los inferiores. Véase la primera columna de

FIGURA 2.5: Aplicación de diferentes filtros a la imagen de la figura 2.4



la figura 2.5. Filtros similares, rotando los valores del filtro F , son capaces de resaltar bordes laterales u oblicuos (resto de columnas de la figura 2.5).

Lo interesante de este método es que hemos conseguido resaltar características del objeto representado en la imagen usando solo operaciones matriciales.

Las matrices de convolución pueden ser de mayor tamaño, permitiendo capturar características más complejas. La matriz 3×3 es la menor matriz que permite definir en su totalidad el concepto de espacio, pudiendo extraer características espaciales en dos dimensiones.

A la hora de trabajar con imágenes en color es necesario aplicar cada filtro a cada capa del modelo RGB por separado, permitiendo de esta manera detectar diferentes características de la imagen que ocurran solo en uno de los colores.

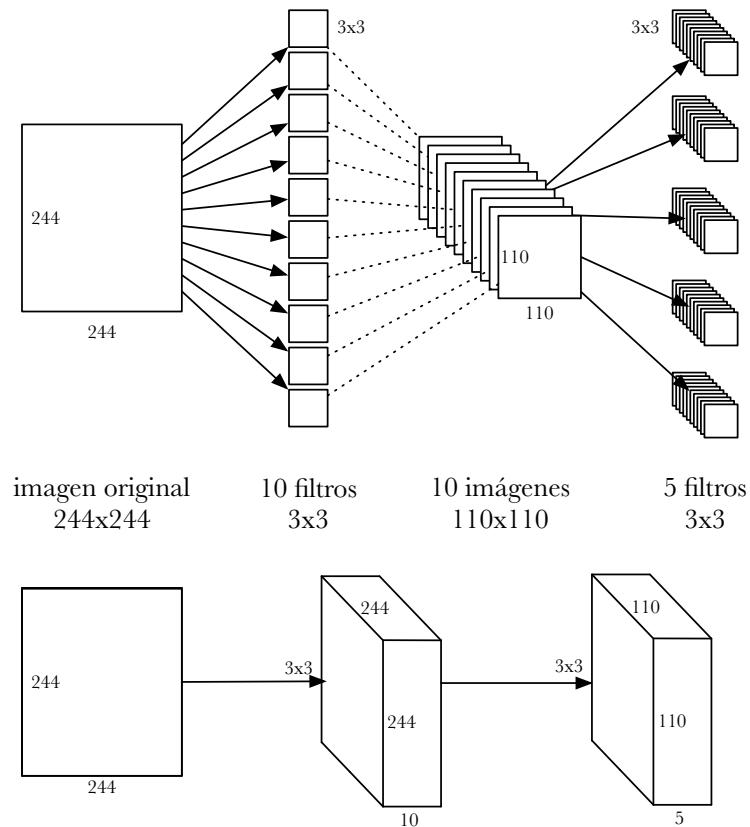
Para analizar cómo afectan diferentes filtros a una imagen, en la página (Powell, 2015) se pueden probar ejemplos con filtros personalizados, haciendo el concepto mucho más sencillo de comprender.

2.4. Redes neuronales convolucionales

Hemos explorado la idea de que determinados filtros sean capaces de extraer información localizada sobre características de la imagen. En el ejemplo del apartado anterior, dada una imagen podíamos saber si había bordes superiores y dónde se podían encontrar. Esto puede ser de gran utilidad en el campo del reconocimiento de imágenes, ya que podemos usar esa información localizada para categorizar o aplicar otro tipo de técnicas en esas áreas señaladas. El problema radica en encontrar los mejores filtros para que extraigan las características más relevantes de una imagen.

Analizando cómo funcionan los filtros convolucionales vemos su parecido con las redes neuronales. Al igual que estas, los filtros se pueden entender como funciones que al aplicarse a los datos de entrada producen unos datos de salida que serán más o menos relevantes para el objetivo buscado. El entrenamiento de una red neuronal va modificando los pesos de las diferentes capas hasta que produce una salida relevante para cada ejemplo del conjunto de entrenamiento. Si entendemos los pesos como matrices convolucionales, podemos hacer que sea la propia red la que busque el mejor filtro para nuestro problema de clasificación. De hecho, ya que las redes neuronales son capaces de componer diferentes funciones en diferentes capas de la red para aprender funciones no lineales, podemos aplicar la misma idea a las redes con filtros: componer diferentes filtros para poder extraer características más complejas.

FIGURA 2.6: Ejemplo de dos representaciones de la misma red convolucional. Abajo el modelo resumido con cajas.



En esta idea se basan las redes convolucionales que trabajan con imágenes. Cada capa de la red constará de varios filtros que se aplicarán a las imágenes que reciba, proporcionando los resultados a la siguiente capa. Ya que lo que importa es la composición de filtros, cada capa deberá entrenar varios filtros al mismo tiempo, permitiendo así aumentar la combinatoria final.

De esta manera, los pesos de cada capa vendrían dados en una matriz tridimensional, que se puede entender como d (profundidad) filtros de dimensión $w \times h$ (anchura y altura), como se indica en la figura 2.6.

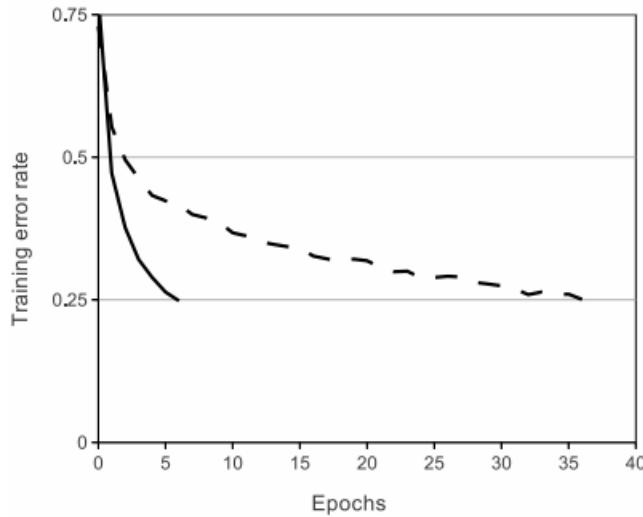
2.4.1. Estructura de una capa convolucional

El concepto de capa en una red convolucional incluye una agrupación de diferentes capas que efectúan diferentes operaciones. Como en las redes neuronales clásicas, la capa precisará de un tratamiento de las entradas con los pesos (en este caso los filtros convolucionales) y una función de activación.

Capa convolucional - CONV (*Convolutional layer*)

Las capas convolucionales son las que hemos visto hasta ahora. Tendrán como pesos n filtros convolucionales (todos del mismo tamaño) y producirán las n imágenes

FIGURA 2.7: Evolución de un entrenamiento para una red convolucional de cuatro capas usando ReLUs (línea sólida) vs usando tanh (línea discontinua) (Krizhevsky, 2012)



resultantes de aplicar estos filtros a la imagen de entrada. Por ejemplo, de una imagen de tamaño 32×32 proporcionada a una capa convolucional de 8 filtros se obtendrá una salida de tamaño $32 \times 32 \times 8$, o lo que es lo mismo, 8 imágenes 32×32 .

Capa de activación - RELU (*Rectifier Linear layer*)

Al igual que en las redes clásicas es necesario tratar la salida de la aplicación de los pesos sobre las entradas con una función de activación. En el caso de las redes convolucionales la función de activación se aplica a cada píxel resultante del filtro.

La manera habitual de modelizar la salida de una neurona en las RNAs es a través de $f(x) = \tanh(x)$ o $f(x) = (1 + e^{-x})^{-1}$ (función sigmoide). A la hora de entrenar la red con descenso del gradiente, estas funciones son mucho más lentas que otra función que también evita la linealidad: $f(x) = \max(0, x)$, llamada ReLU (*Rectifier Linear*). Las redes neuronales convolucionales entranan varias veces más rápido usando ReLU que las equivalentes usando la función *tanh* (ver figura 2.7).

Capa de muestreo - POOL (*Pooling layer*)

Al usar convoluciones en una imagen, la información del entorno de cada punto queda difuminada y redundante. Gracias a esto se puede reducir el tamaño del problema mediante muestreo. Las capas de muestreo (*pooling layers*) resumen las salidas del entorno de cada grupo de neuronas. Un ejemplo de función de muestreo sería elegir el valor máximo de cada cuadrado de 4 píxeles (figura 2.8).

Si la salida de una capa CONV + RELU es $32 \times 32 \times 8$, al aplicar la capa de muestreo la salida se convertirá en $16 \times 16 \times 8$. Al eliminar el 75 % de las activaciones se reducen la cantidad de parámetros y el tiempo de computación, y además ayuda a reducir el sobreajuste.

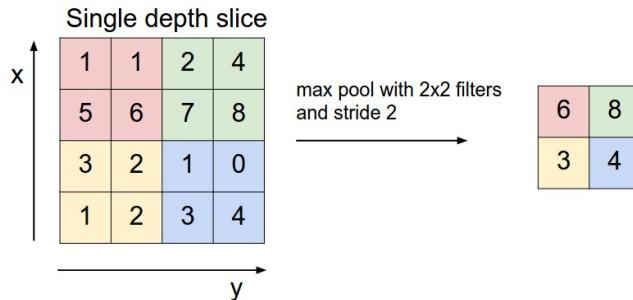
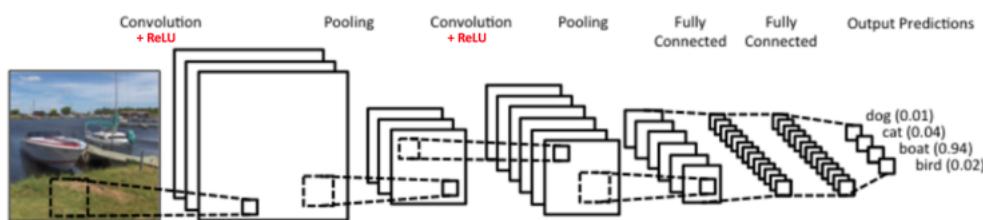
FIGURA 2.8: *Max-pooling* sobre una imagen de 4×4 

FIGURA 2.9: Ejemplo de arquitectura de una red convolucional para clasificación, (Russakovsky, 2014)



Capa densa - FC (*Fully connected layer*)

Esta capa es una capa clásica de red neuronal. Su función es calcular las probabilidades de clasificación dadas las imágenes tratadas. Cada neurona de esta capa estará conectada a cada una de las activaciones de la capa anterior, produciendo una salida por cada clase a clasificar.

Esta capa suele usarse al final de la arquitectura, cuando se han producido varios ciclos de capas convolucionales (CONV + RELU + POOL). Sin embargo, también es común ver arquitecturas con varias capas FC conectadas al final, ampliando la flexibilidad de clasificación de la red en base a características extraídas.

2.4.2. Arquitectura

Un esquema simplificado de un ejemplo de uso de las capas mencionadas se puede encontrar en la figura 2.9. La idea es, mediante capas CONV + RELU + POOL, transformar la imagen en una multitud de representaciones de conceptos cada vez más abstractos. Una vez se alcancen dichos conceptos, usar capas FC para modelizar la salida de la red.

Como se verá en la solución presentada para este proyecto, en el capítulo 4, usar esta arquitectura permite apoyarse en determinadas estrategias para mejorar la eficacia del modelo predictivo.

Capítulo 3

Definición del problema

3.1. *The Nature Conservancy Fisheries Monitoring*

En el océano Pacífico, donde se captura más del 60 % del atún del mundo, tienen lugar prácticas de pesca irregular que amenazan a los ecosistemas marinos y a la estabilidad de la pesca mundial. *The Nature Conservancy* es una asociación que trabaja con organizaciones locales y globales para preservar las especies marinas de cara al futuro.

La principal idea para controlar la pesca y explotación de recursos marinos es el uso de cámaras en barcos, que ayudan a monitorizar las actividades pesqueras de estos. Aunque funciona muy bien como sistema de control, la cantidad de datos e imágenes generadas hace que sea muy costoso de procesar manualmente.

La idea de este reto es desarrollar algoritmos que detecten y clasifiquen automáticamente especies de atunes, tiburones y otras especies que estos barcos pesqueros cazan. El que se pueda analizar rápida y automáticamente imágenes como las de la figura 3.1 permitirá asignar recursos de una manera mucho más efectiva para el control de este tipo de actividades (*The Nature Conservancy Fisheries, 2016*).

3.2. Definición del problema

El problema consiste en clasificar cada una de las imágenes de un conjunto de imágenes de barcos en una de las ocho categorías disponibles. Las imágenes suelen

FIGURA 3.1: Ejemplo de imágenes tomadas en barcos pesqueros para su posterior identificación



FIGURA 3.2: Especies de peces a clasificar en el reto



mostrar la cubierta de un barco donde puede aparecer un pez. En base al pez que aparezca hay que clasificarlo en una de las seis categorías mostradas en la figura 3.2.

En caso de que no aparezca ningún pez en la imagen, esta tendrá la categoría NOF (*No Fish*). Y si aparece un pez en la imagen no perteneciente a ninguna de las categorías mencionadas, la categoría será OTHER.

En resumen, el conjunto de posibles categorías es:

$$\text{categories} = [\text{ALB}, \text{BET}, \text{DOL}, \text{LAG}, \text{SHARK}, \text{YFT}, \text{OTHER}, \text{NOF}]$$

3.2.1. Datos

La competición proporciona tres ficheros con los que trabajar:

1. Conjunto de datos de entrenamiento: 3777 imágenes etiquetadas con una de las ocho categorías existentes.
2. Conjunto de test y evaluación: 1000 imágenes sin etiquetar.
3. Archivo de envío de prueba: Archivo CSV que muestra la estructura que debe tener el archivo con las soluciones

CUADRO 3.1: Ejemplo del archivo de envío

image	ALB	BET	DOL	LAG	NoF	OTHER	SHARK	YFT
img_00005.jpg	0.455	0.052	0.030	0.0173	0.123	0.079	0.046	0.194
img_00006.jpg	0.455	0.052	0.030	0.0173	0.123	0.079	0.046	0.194
img_00007.jpg	0.455	0.052	0.030	0.0173	0.123	0.079	0.046	0.194

3.2.2. Envío de la solución y evaluación

Para el envío de la solución hace falta clasificar las 1000 imágenes del conjunto de evaluación, indicando la probabilidad de que pertenezca a cada una de las ocho categorías diferentes. En el cuadro 3.1 se muestra un ejemplo de algunas filas del archivo CSV a enviar.

Los resultados enviados se evalúan mediante una función de pérdida logarítmica multiclas. Concretamente se usa la fórmula

$$\text{logloss} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log p_{ij}$$

siendo N el número de imágenes en el conjunto de test, M el número de categorías, y_{ij} 1 si la observación i pertenece a la categoría j y 0 si no pertenece y p_{ij} la probabilidad predicha de que el elemento i pertenezca a la categoría j .

Cuanto menor sea el valor obtenido, mejor se habrá comportado el modelo sobre el conjunto de prueba.

3.2.3. Tablas de clasificación y fases de la competición

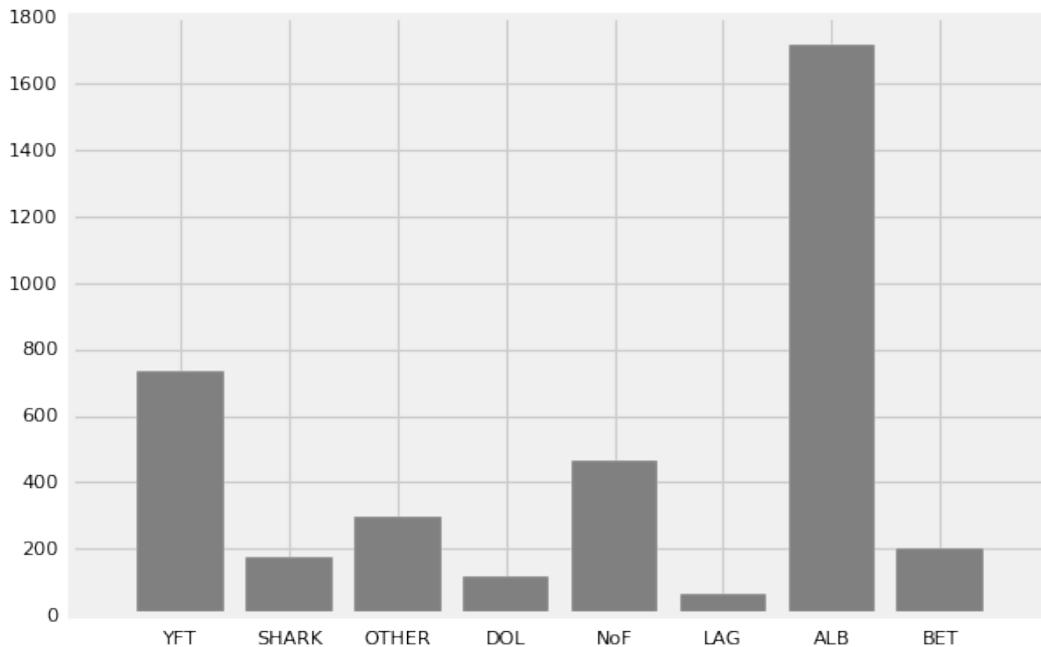
Cuando un participante envía una predicción, *Kaggle* calcula la puntuación de dicho envío sobre un subconjunto de ejemplos compuesto del 8 % del conjunto de test, mostrando la puntuación en una tabla de clasificación pública. Se usa un subconjunto para evitar que los participantes puedan aprovechar un sobreajuste a la hora de entrenar el modelo.

Antes de terminar la primera fase de la competición los participantes deberán subir a *Kaggle* el modelo utilizado y seleccionar las dos predicciones que quieren usar para la puntuación con el conjunto completo. La puntuación de estas predicciones se hará usando todo el conjunto de test y se publicará en una tabla de clasificación privada, solo visible para los participantes.

Una vez terminada la primera fase, *Kaggle* publicará un nuevo conjunto de test, con el que los participantes deberán clasificar usando el modelo entregado al final de la fase anterior.

En esta segunda fase, que dura solo cinco días, el procedimiento es el mismo. Se publicarán puntuaciones usando un subconjunto del segundo conjunto de test y al terminar se publicará la puntuación usando el resto del conjunto. Esta última puntuación será la puntuación final de la competición. Si alguno de los participantes

FIGURA 3.3: Cantidad de imágenes pertenecientes a cada categoría



ha hecho un envío en la primera fase pero no ha hecho ninguno en la segunda, quedará eliminado de la competición.

3.2.4. Conjunto de entrenamiento

El conjunto de datos de entrenamiento proporcionado en la competición consta de 3777 imágenes, divididas en ocho categorías. La figura 3.3 muestra la cantidad de imágenes pertenecientes a cada categoría.

Revisando el conjunto de datos encontramos varios problemas:

- La distribución de imágenes entre las categorías es altamente desigual. La categoría ALB tiene 1719 imágenes, que suponen más del 45 % del conjunto completo. Por otro lado, las dos categorías con menos ejemplos, DOL y LAG contienen 117 y 67 imágenes respectivamente. Un modelo entrenado con este conjunto de entrenamiento puede tener problemas clasificando estas categorías con tan pocos ejemplos.
- A la hora de dividir el conjunto en subconjuntos de entrenamiento, validación y test hay que tener cuidado de dejar ejemplos de todas las categorías en cada uno. Con tan pocos ejemplos es sencillo dejar una categoría completa fuera de un conjunto al elegir imágenes aleatoriamente.
- Si visualizamos las imágenes notamos que algunas son casi idénticas. Los ejemplos de la figura 3.4 parece que han sido sacadas de fotogramas de un mismo vídeo. Esto reduce la diversidad de ejemplos que contienen cada categoría. Teniendo una categoría con solo 67 ejemplos es preocupante que algunos de las

FIGURA 3.4: Dos imágenes del conjunto de entrenamiento casi idénticas



imágenes sean casi idénticas, ya que el modelo puede llegar a usar características del barco en el que se encuentran los peces si mostramos variaciones del mismo pez con el mismo fondo.

- Se puede observar que en algunas imágenes los peces son muy difíciles de identificar, debido a que están parcialmente ocultos, la imagen está oscura o la cámara está sucia. Recordemos que algunas de las categorías son diferentes familias de atunes, con formas bastante parecidas. Distinguir entre dos familias de atunes en una imagen oscura y con el atún parcialmente oculto puede llegar a ser imposible para un humano.
- Algunas imágenes están mal etiquetadas. Si el modelo predice la clase incorrecta de la imagen (según su etiqueta) con mucha confianza, le asignará poca probabilidad a la clase correcta. Una probabilidad cercana a cero hará que la pérdida logarítmica aumente mucho.

3.2.5. Conjuntos de test

Para el envío de predicciones es necesario clasificar cada imagen del conjunto de test que se proporciona en cada una de las fases de la competición. En la primera fase el conjunto de test consta de 1000 imágenes para clasificar, mientras que en la segunda consta de 13000.

No se ha realizado ningún tipo de inspección o exploración de ninguno de los conjuntos de test para evitar cualquier tipo de sesgo a la hora de crear el modelo. El segundo conjunto de test no se hizo público hasta seis días antes de terminar la competición, por lo que era imposible saber qué tipo de imágenes iba a contener. Una exploración del primer conjunto de test podría haber hecho que se sobreajustara el modelo a este conjunto, empeorando la puntuación para el siguiente conjunto si este era diferente.

Capítulo 4

Soluciones presentadas

4.1. Metodología

La puntuación final de la competición será calculada mediante una función de pérdida logarítmica, como vimos en el apartado 3.2.2. Al no tener las categorías del conjunto de datos de test debemos delegar el cálculo de la puntuación en *Kaggle*, que permite enviar hasta cinco predicciones al día. Esto es un obstáculo para hacer pequeñas pruebas e iterar rápido sobre los resultados. Por otra parte, el conjunto de test de la segunda fase tiene más de 13000 imágenes, que no son rápidas de cargar en memoria ni de pasar por el modelo. En determinados casos la evaluación del segundo conjunto de test ha tardado más de 6 horas.

La solución a este problema se ha resuelto usando un subconjunto del conjunto de entrenamiento dedicado solo a la evaluación de modelos. Por otra parte, para el entrenamiento es necesario usar un subconjunto de validación, por lo tanto es necesario dividir el conjunto de entrenamiento original en tres subconjuntos: entrenamiento, validación y test. La partición se ha realizado dejando un **60 %** de los datos al conjunto de entrenamiento, un **20 %** al conjunto de validación y un **20 %** al conjunto de test.

Otra posibilidad sería emplear la metodología de validación cruzada, que consiste en dividir el conjunto de entrenamiento en n partes, usando $n - 1$ de ellas para el entrenamiento y la restante para la validación. Esto se repite n veces, considerando en cada una de ellas uno de los subconjuntos como conjunto de validación. La bondad del modelo será el valor medio de todos los modelos generados. No obstante, en los algoritmos de redes convolucionales con imágenes no se suele usar la validación cruzada, ya que son algoritmos muy costosos y puede aumentar mucho el tiempo de entrenamiento.

4.1.1. Partición del conjuntos de datos

La partición de un conjunto de datos es una operación delicada. Ya vimos que uno de los problemas de nuestro conjunto de datos de entrenamiento era que la cantidad de imágenes para cada clase era muy variada, teniendo algunas de ellas un número muy bajo de ejemplos. Si seleccionamos aleatoriamente el 40 % de las imágenes, es posible que no se elijan ejemplos de una o varias de las categorías, con lo que los modelos construidos no las tendrían en cuenta.

La solución adoptada ha sido realizar un muestreo estratificado para realizar la partición del conjunto de datos. Este consiste en realizar la partición de cada clase por separado, siguiendo la proporción 60 % - 20 % - 20 % indicada anteriormente.

4.1.2. Evaluación del modelo

Para una búsqueda más eficiente de parámetros y configuraciones los modelos se evaluarán previamente usando nuestro subconjunto de test local. Aquellos modelos que resulten especialmente interesantes por tener una puntuación máxima local entre aquellos a los que se compara se subirán entonces a *Kaggle* para su evaluación usando el conjunto de test correspondiente a la fase en la que nos encontremos.

Kaggle usa la pérdida logarítmica para medir la bondad de cada modelo (sección 3.2.2). Esta métrica se verá perjudicada si puntuamos con una probabilidad muy baja la categoría correcta de la imagen. Una técnica para tratar de aliviar este problema consiste en truncar las probabilidades de cada clase, tanto en su valor máximo como en el mínimo. En nuestro caso vamos a hacer que las probabilidades superiores a 0.95 se trunquen a 0.95 y las inferiores a 0.15 se trunquen a 0.15.

4.1.3. Software

Todos los modelos de redes convolucionales construidas en este proyecto han sido entrenados usando *Keras* (véase apéndice B) sobre cuadernos de *Jupyter* (Pérez, 2017). La estructura de desarrollo y algunas utilidades están descritas en (Howard, 2017) y (Yu, 2017).

4.2. Arquitectura de los modelos

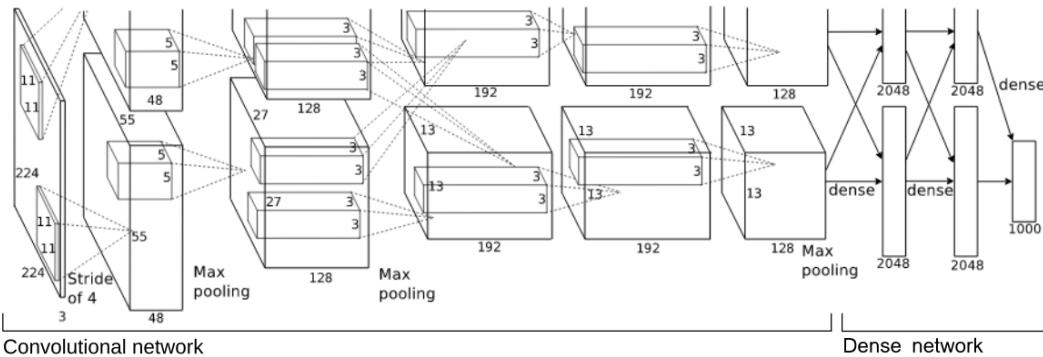
Los humanos somos expertos en la identificación visual. Esto se debe, en parte, a que nos llevamos la mayor parte de nuestra vida resolviendo problemas de identificación visual y somos muy eficientes generalizando las características de una imagen.

La competición de reconocimiento visual a gran escala de *ImageNet*, conocida por sus siglas ILSVRC (*Imagenet Large Scale Visual Recognition Challenge*) (ILSVRC, 2017) es una competición de reconocimiento de imágenes que pretende seguir el progreso alcanzado por los modelos de inteligencia artificial a la hora de identificar y generalizar elementos de una imagen. En esta competición se deben detectar elementos en 150000 fotografías y clasificarlos en una entre 1000 categorías diferentes.

En 2010 y 2011 los modelos ganadores consiguieron una tasa de error de un 28,2 % y un 25.8 %, respectivamente. En 2012, el modelo presentado en (Krizhevsky, 2012), una red convolucional profunda, ganó con una tasa de error del 16.4 %, superando en casi un 10 % al segundo clasificado. Esta victoria hizo que el enfoque de esta propuesta haya sido usado por los ganadores de los años siguientes.

En este trabajo se pretende aprovechar la capacidad de generalización de los modelos ganadores del ILSVRC. Es decir, no se construirá una red convolucional con

FIGURA 4.1: Arquitectura de una red convolucional profunda. Consta de una acumulación de capas convolucionales destinadas a extraer características de la entrada y una serie de capas densas para clasificar la entrada usando dichas características.



una estructura similar a la de (Krizhevsky, 2012), sino que se utilizará esa misma red adaptándola a nuestro problema.

4.2.1. Red convolucional

La figura 4.1 representa la red usada en ILSVRC 2012, la cual se compone de una red convolucional para la extracción de características de la imagen, seguida de una red densa para la clasificación de la imagen usando esas características extraídas.

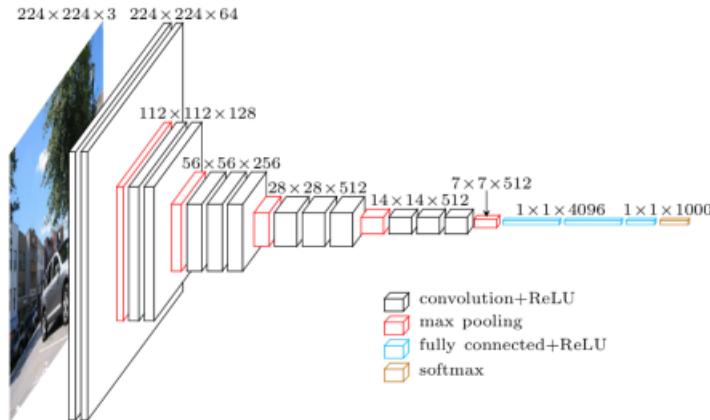
Como explicamos en la sección 2.4, cuando una red convolucional recibe una imagen devuelve N matrices bidimensionales representando el resultado de la aplicación de los N conjuntos de filtros a la imagen inicial. Intuitivamente podemos pensar que la salida de cada uno de estos filtros representa la aparición en la imagen de la característica que esté detectando el filtro. Necesitamos entonces que en las últimas capas la red aprenda filtros que se activen sólo con la aparición de una clase de pez.

Un detalle que debemos tener en cuenta que este modelo convolucional no ha sido entrenado con el conjunto de entrenamiento del problema, sino con el conjunto de entrenamiento de ImageNet, 2017. Por lo tanto, nuestro uso de esta red convolucional será únicamente para transformar cada imagen del conjunto de entrenamiento del problema a un conjunto de características.

La arquitectura del artículo original (Krizhevsky, 2012) usa capas convolucionales donde alterna filtros de 11×11 , 5×5 y 3×3 , el cual parece una buena elección para usar como modelo convolucional preentrenado. Sin embargo, la aplicación de este trabajo es una competición internacional donde se usarán soluciones *state of the art*. El modelo de la figura 4.1 representa el ganador de la edición 2012 de la competición-ILSVRC, hace ya cinco años, por lo que resulta conveniente analizar cuales fueron los ganadores de años posteriores.

El modelo principal que usaremos es VGG, desarrollado por el *Visual Geometry Group*, de la Universidad de Oxford (Simonyan y Zisserman, 2014). Es un modelo especialmente interesante por su simplicidad, aparte de obtener una de las mejores puntuaciones en ILSVRC 2014. Una de las principales características de VGG es la

FIGURA 4.2: Arquitectura de VGG



idea de que los filtros convolucionales mayores de 3×3 , como por ejemplo los de 5×5 u 11×11 , pueden ser representados por combinaciones de filtros 3×3 .

De las configuraciones descritas en (Simonyan y Zisserman, 2014) hay una que sobresale por su eficiencia, llamada VGG16. Usando un total de trece capas CONV con filtros de 3×3 , cinco capas POOL y tres capas FC (de 4096, 4096 y 1000 salidas), seguida de una función *softmax* (figura 4.2), es capaz de mejorar la eficacia del modelo de Krizhevsky. El nombre de esta configuración es VGG16, por ser esta la cantidad de capas CONV y FC que posee.

4.2.2. Ajuste fino

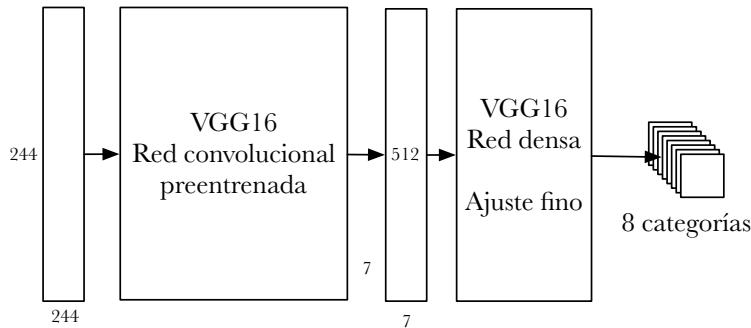
Si observamos la última capa del modelo VGG16, vemos que la salida tiene 1000 elementos. Tiene esta forma ya que ILSVRC consiste en clasificar una imagen entre mil categorías diferentes. Como en nuestro problema solo tenemos ocho categorías, es lógico modificar esta última capa para que únicamente tenga ocho salidas.

Al modificar la estructura de la última capa estamos destruyendo pesos y haciendo que muchos de los que ya existían carezcan de sentido. El hecho de que VGG tiene esta separación lógica entre la red convolucional y la red densa hace que se pueda separar el modelo en dos submodelos diferentes: uno convolucional, que no habrá cambiado con la adaptación a las ocho categorías, y otro denso, que tendrá que ser entrenado de nuevo.

Esta técnica de ajustar los parámetros de un modelo ya conocido para adaptarlo a un nuevo conjunto de datos se conoce como ajuste fino (*fine-tuning*). En la figura 4.3 se puede ver un resumen de las diferentes etapas por las que pasa la red.

Al dividir la red en dos subredes diferentes hay que tener en cuenta que la segunda, la red densa, no recibe como entrada las imágenes, sino la salida de la primera red convolucional, con todas las transformaciones que esta produce. Es necesario entonces aplicar la red convolucional a nuestro conjunto de datos de entrenamiento para crear un nuevo conjunto de datos con el que reentrenar la red densa.

FIGURA 4.3: Estructura del modelo final. Como recibe como entrada imágenes de 244 píxeles, pasa por las diferentes redes y devuelve el vector de ocho categorías.



El siguiente código (simplificado) realiza esta tarea para los subconjuntos locales de entrenamiento, validación y prueba.

```

# Carga del modelo VGG16
from vgg16 import Vgg16
model = Vgg16()

# Separacion entre red convolucional y densa
import utils
conv_model, dense_model = utils.split_at(model, MaxPooling2D)

# Carga de los diferentes conjuntos de datos
# 'path' es la carpeta donde se encuentran los
# conjuntos de datos
train, train_labels = get_data(path + 'train')
valid, valid_labels = get_data(path + 'valid')
test, test_labels = get_data(path + 'test')

# Conversion a caracteristicas mediante la red convolucional
train_feat = conv_model.predict(train)
valid_feat = conv_model.predict(valid)
test_feat = conv_model.predict(test)

# Sustitucion de la capa de salida
dense_model.pop() # Eliminar la capa final
dense_model.add(
    Dense(8, activation='softmax') # Introducir una nueva capa
)

# Compilacion del modelo
dense_model.compile(
    SGD(lr=0.01), # Factor de aprendizaje
    loss='categorical_crossentropy',
    metrics=['accuracy'], # Muestra la precision del modelo
)

```

```
# Entrenamiento de la red
dense_model.fit(
    train_feat,
    train_labels,
    batch_size=64, # Numero de imagenes a entrenar al mismo tiempo
    nb_epoch=7,     # Numero de iteraciones del entrenamiento
    validation_data=(valid_feat, valid_labels),
)

# Evaluar el modelo sobre el conjunto de test
dense_model.evaluate(test_feat, test_labels)
# Log loss, accuracy
>>> [1.90571456890184755, 0.66099843993759748]
```

La puntuación total para el conjunto de test local sería 1.906.

El segundo número que devuelve la función *evaluate* es la precisión (*accuracy*) del modelo. Al ser este un modelo de clasificación, la precisión es el porcentaje de veces que la clase con la probabilidad máxima se corresponde con la clase etiquetada en el conjunto de datos. En este caso la red ha clasificado correctamente el 66 % de las imágenes, un número muy aceptable teniendo en cuenta que solo se ha tocado la capa final de un modelo ajeno.

Al subir la predicción del conjunto de test a *kaggle* se obtiene una puntuación de **2.632** en la clasificación pública, mediante la pérdida logarítmica. Hay que recordar que esta puntuación se calcula usando un subconjunto del conjunto de test, así que la puntuación de este modelo puede cambiar en la segunda fase.

Al subir a *Kaggle* las predicciones del modelo sobre el conjunto de prueba de la primera fase de la competición se obtuvo una puntuación de pérdida logarítmica de 2.632. Aunque el error ha aumentado sensiblemente esto es perfectamente normal, ya que las imágenes proporcionadas al modelo son completamente nuevas.

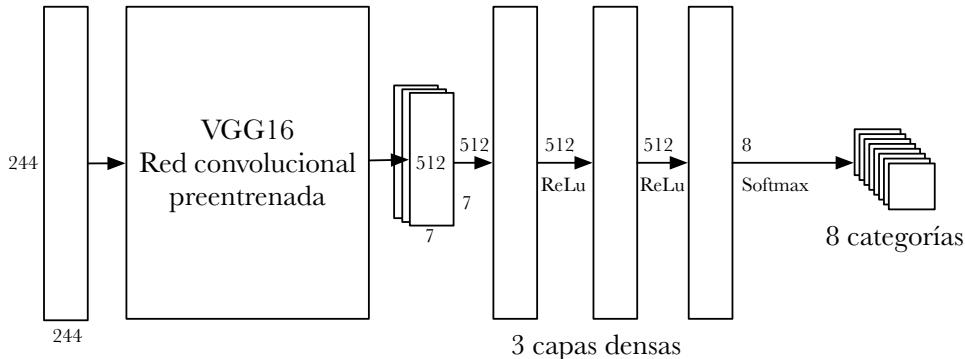
4.3. Modelo completamente conectado

El modelo original de VGG usa dos capas *densas* de 4096 neuronas para clasificar entre mil clases diferentes. Ya que nuestro problema tiene solo ocho clases diferentes probablemente no sea necesario usar capas tan grandes, por lo que se procede a probar la misma estructura original, pero con capas cuatro veces más pequeñas que las originales.

La estructura, por lo tanto, quedaría de la misma manera pero usando dos capas densas de 512 neuronas cada una y una capa densa final de 8 (figura 4.4). Para no tener que estar modificando cada vez el modelo original cada vez que haga falta es mucho más sencillo crear un modelo nuevo con *Keras* y añadir todas las capas necesarias.

Un paso importante a la hora de construir el modelo personalizado es tener en cuenta que las salidas de las redes convolucionales poseen tres dimensiones (*ancho* ×

FIGURA 4.4: Arquitectura del modelo completamente conectado



($alto \times filtros$), mientras que las redes neuronales densas poseen solo una dimensión. Es necesario convertir la salida de estas capas a una entrada permitida. *Keras* ya ofrece esta posibilidad usando una capa abstracta llamada *Flatten*.

```

def list_dense_layers():
    return [
        Flatten(),
        Dense(512, activation='relu'),
        Dense(512, activation='relu'),
        Dense(8, activation='softmax')
    ]

dense_model = keras.models.Sequential(list_dense_layers())

# Entrenar la red
dense_model.compile(...)
dense_model.fit(...)

# Evaluar el modelo sobre el conjunto de test
dense_model.evaluate(test_feat, test_labels)
>>> [1.27161316430079874, 0.73067862714508579]

```

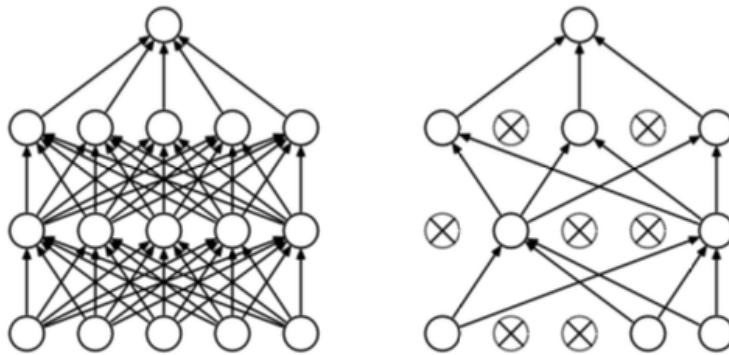
Los resultados obtenidos son ligeramente mejores que los anteriores. Sin embargo no es aquí donde está toda la mejora. El entrenamiento de este modelo ha sido un 80 % más rápido que el anterior (19.5 segundos por iteración en el primero, 3.9 segundos en este último). Esto no solo ha permitido entrenar la red durante más tiempo, sino que en el futuro hará posible entrenar sobre mayores cantidades de datos sin aumentar excesivamente el tiempo de entrenamiento.

La puntuación en la tabla pública de *Kaggle* para este modelo ha sido de **2.381**.

4.3.1. Análisis del sobreajuste

La salida de *Keras* al entrenar el modelo ofrece información de lo que está sucediendo. Este es un ejemplo de la salida del entrenamiento del modelo anterior.

FIGURA 4.5: A la izquierda una red neuronal estándar con dos capas ocultas. A la derecha la misma red aplicando un *dropout* en cada una de las capas ocultas. Las neuronas tachadas han perdido su activación.



```

Epoch 5/7
loss: 0.354 - acc: 0.992 - val_loss: 1.091 - val_acc: 0.872
Epoch 6/7
loss: 0.253 - acc: 0.996 - val_loss: 0.995 - val_acc: 0.881
Epoch 7/7
loss: 0.193 - acc: 0.997 - val_loss: 0.987 - val_acc: 0.913

```

En cada una de las iteraciones del entrenamiento, *Keras* muestra los valores de aplicar el modelo actual a los ejemplos de entrenamiento y validación. Se puede observar que el modelo funciona mucho mejor en el conjunto de entrenamiento que en el de validación.

Cuando un modelo tiene demasiados parámetros y ha sido entrenado durante demasiado tiempo aprende a clasificar los ejemplos con los que entrena usando información específica de cada uno de ellos en vez de generalizar. Esto se conoce como **sobreajuste** (*overfitting*).

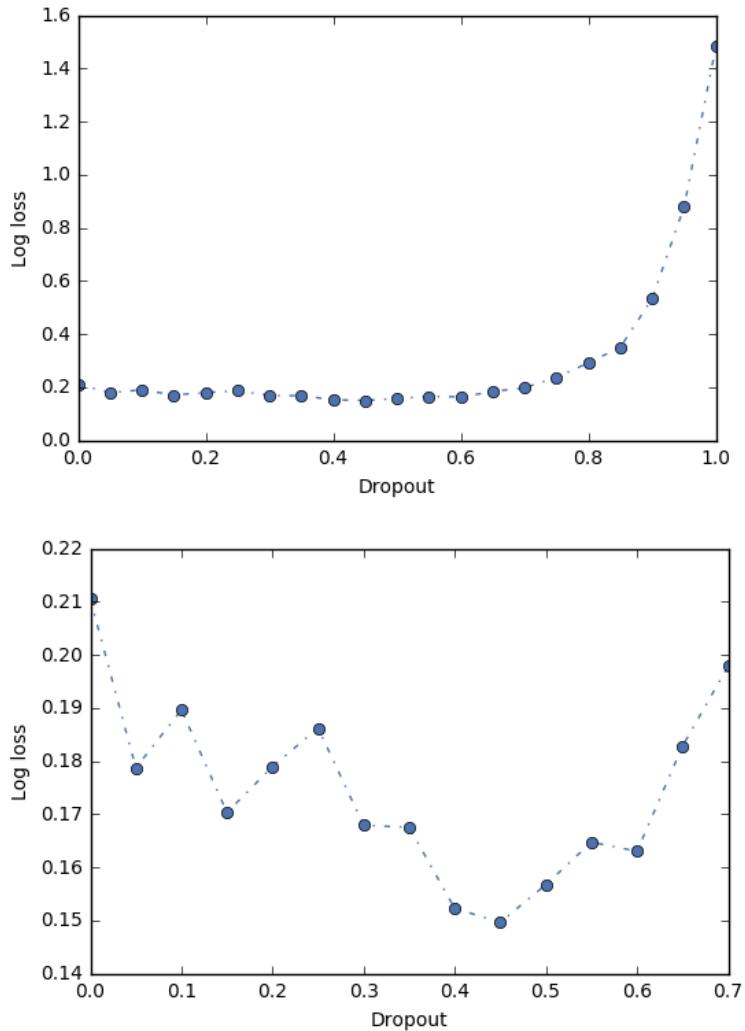
Los datos del modelo anterior indican que puede existir un sobreajuste, por lo que se va a intentar tomar medidas para arreglarlo.

4.3.2. *Dropout*

Una de las características de VGG y otras redes convolucionales es el uso de *dropout* para reducir el sobreajuste de los modelos entrenados. El *dropout* consiste en una capa que se aplica después de las capas de activación de las capas densas. Esta capa convierte activaciones aleatorias a 0, eliminando la información transportada (Figura 4.5)

En un principio parecería que esto perjudica al modelo, pero al eliminar algunos de los pesos el modelo evita centrarse en características individuales de cada ejemplo de clasificación, obligándolo a generalizar más rápido (Srivastava, 2014).

Un pequeño experimento en un modelo más avanzado (VGG con *batch normalization* y aumento de datos) permite ver cómo afecta el *dropout* a la puntuación final.

FIGURA 4.6: Evolución de la puntuación de un modelo usando *dropout*

En la figura 4.6 se puede observar que la mejor puntuación se alcanza eliminando el 45 % de las activaciones, consiguiendo una mejora de un 25 % sobre el modelo que usa todas las activaciones.

También se ve que el modelo empieza a perder eficacia a partir del 70 % de *dropout*, empeorando el modelo original. Esto significa que el modelo es capaz de generalizar con información útil incluso cuando solo posee el 30 % de las activaciones.

Por otra parte, también es importante pensar dónde se pierden las activaciones. Por un lado, perder demasiadas activaciones en la entrada sería el equivalente a trabajar sin esos ejemplos. Por otro, perderlos en la salida sería aumentar demasiado el error al clasificar, al no tener tanta capacidad de apoyo en las otras capas. La idea aplicada aquí es distribuir el *dropout* disminuyéndolo en la entrada y la salida y haciendo que alcance máximo en el centro de la red.

El modelo modificado quedaría de la siguiente manera:

```
def list_dense_layers(dropout):
    return [
```

```

        Flatten(),
        Dropout(dropout/4), # Aplicar 1/4 del dropout definido
        Dense(512, activation='relu'),
        Dropout(dropout),
        Dense(512, activation='relu'),
        Dropout(dropout/2), # Aplicar la mitad del dropout definido
        Dense(8, activation='softmax'),
    ]
dense_model = keras.models.Sequential(list_dense_layers(0.45))

```

Los resultados obtenidos con esta configuración no mejoran los resultados de la red anterior, pero como se ve en la figura 4.6 sí que lo hará a posteriori, cuando se le apliquen otro tipo de modificaciones al modelo. Modelos como VGG ya incluyen *dropout* (solo en las redes densas), por lo que al hacer fine-tuning con una red personalizada es necesario usarlo para mantener los resultados originales.

4.3.3. Normalización por lotes

En muchos campos del aprendizaje automático es habitual normalizar las entradas de los modelos. Si existe una entrada mucho mayor o menor que el resto esta puede hacer que el entrenamiento arrastre un error mayor del necesario, produciendo inestabilidad y dificultando la convergencia del modelo. Normalizar un conjunto de entradas hace que todas estén en la misma escala.

El caso de las redes neuronales no es una excepción. Una entrada con un valor demasiado grande puede llevar a tener un peso determinado demasiado grande para contrarrestarla.

Una normalización estándar en aprendizaje automático es restar de cada entrada el valor medio del conjunto de datos y luego dividirlo por la desviación estándar del mismo. Esto aún presenta problemas para casos como el de este proyecto donde se usan algoritmos como el gradiente del descenso estocástico para la propagación hacia atrás (sección 2.1.1).

Los modelos principales citados en este trabajo, (Krizhevsky, 2012) y (Simonyan y Zisserman, 2014), usan sus propias técnicas de normalización, aparte de usar activaciones *ReLU* (sección 2.4.1) que son menos sensibles a los datos de entrada sin normalizar. Sin embargo, en 2015 se presenta una técnica muy interesante de normalización que cuadra muy bien con el gradiente del descenso estocástico: la normalización por lotes (*batch normalization*) (Ioffe y Szegedy, 2015).

La diferencia con una normalización estándar es que tras normalizar las entradas de una capa se multiplican por un parámetro aleatorio y luego se suma otro, cambiando así la desviación estándar y la media de la entrada. Estos dos parámetros se hacen entonces entrenables como pesos del modelo.

Los modelos actuales que usan esta normalización por lotes consiguen la misma precisión que los modelos sin ella usando catorce veces menos pasos de entrenamiento (Ioffe y Szegedy, 2015).

Para aplicar esta normalización por lotes al modelo que se está entrenando hay que añadir las capas de normalización por lotes (también disponibles en *Keras*) al modelo denso:

```
def list_dense_layers(dropout):
    return [
        BatchNormalization(axis=1),
        Flatten(),
        Dropout(dropout/4),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(dropout),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(dropout/2),
        Dense(8, activation='softmax'),
    ]
dense_model = keras.models.Sequential(list_dense_layers(0.45))
```

Esto no es suficiente. A diferencia del *dropout*, la normalización por lotes sí que es capaz de mejorar la red convolucional del modelo preentrenado. Gracias a la popularidad de esta técnica y de los modelos preentrenados que se usan en este trabajo, ya existen entrenamientos del modelo VGG original usando normalización por lotes, ahorrando la necesidad de entrenar una red tan grande. Una descripción de estos entrenamientos se puede encontrar en (Simon, Rodner y Denzler, 2016).

En el caso de este trabajo se ha adaptado la clase *Vgg16BN*, siguiendo los principios descritos en (Howard, 2017). Al cambiar la red convolucional original hay que volver a transformar el conjunto de datos que se usaba para obtener los resultados de aplicar los filtros actualizados sobre las imágenes de entrada.

```
# Carga del modelo VGG16 con normalizacion por lotes
from vgg16bn import Vgg16BN
import utils
model = Vgg16BN()
conv_model, _ = utils.split_at(model, MaxPooling2D)

# Carga de los diferentes conjuntos de datos
train, train_labels = get_data(path + 'train')
valid, valid_labels = get_data(path + 'valid')
test, test_labels = get_data(path + 'test')

# Conversion a caracteristicas mediante la red convolucional
train_feat = conv_model.predict(train)
valid_feat = conv_model.predict(valid)
test_feat = conv_model.predict(test)

# Entrenamiento la red
dense_model = keras.models.Sequential(list_dense_layers(0.45))
```

```

dense_model.compile(...)
dense_model.fit(...)

# Evaluacion del modelo sobre el conjunto de test
dense_model.evaluate(test_feat, test_labels)
>>> [0.45127591543710, 0.96109451984491]

```

Como se puede apreciar, los dos últimos cambios han supuesto una mejora considerable sobre el modelo anterior, clasificando correctamente el 96 % de los ejemplos del conjunto de test.

La puntuación en la tabla pública de *Kaggle* para este modelo ha sido de **1.017**.

4.3.4. Aumento de datos

Uno de los principales obstáculos de este problema es el reducido tamaño del conjunto de datos de entrenamiento. El tener pocos ejemplos sobre los que entrenar hace que al modelo le cueste generalizar sobre los ejemplos disponibles, produciéndose un sobreajuste.

Uno de los métodos que se usan para tratar de reducir el sobreajuste del modelo es el llamado aumento de datos (*data augmentation*). Consiste en generar imágenes de entrenamiento adicionales a partir de las ya disponibles, rotando la imagen original, aumentándola, cambiando ligeramente el color, etc.

En el caso de nuestro problema esta técnica es especialmente interesante, ya que las cámaras de los barcos están fijas apuntando siempre a la misma zona. En las fotos sobre los peces capturados estos siempre suelen estar con la cabeza apuntando en dirección contraria al agua. Ello hace que la red reconozca más fácilmente peces dispuestos en una determinada dirección, costándole más trabajo para el resto de direcciones. Si las redes convolucionales ayudaban a detectar determinados patrones en cualquier punto de la imagen, el aumento de datos hace lo propio de otras variantes como la rotación, el tamaño o el filtro de color dado por el ambiente.

Un ejemplo de aumento de datos se muestra en la figura 4.8, que contiene cuatro imágenes obtenidas al reescalar de distintas maneras la imagen original de la figura 4.7.

Para trabajar con aumentos de datos, *Keras* proporciona una serie de opciones de transformación dentro de la utilidad para importar conjuntos de datos. Al definir dichas opciones, como por ejemplo rotación, zoom o volteo horizontal, *Keras* generará nuevas imágenes a la vez que carga el conjunto de datos en memoria.

Si es necesario conocer cómo se aplican las transformaciones a las diferentes imágenes, es posible especificar un directorio de salida del generador de imágenes, de tal manera que los resultados de las distintas transformaciones se guarden en directorios particulares.

```

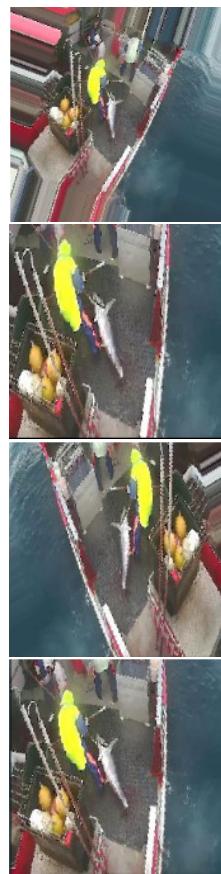
# Creacion del generador de imagenes
image_generator = image.ImageDataGenerator(
    rotation_range=30,

```

FIGURA 4.7: Imagen del conjunto de datos



FIGURA 4.8: Cuatro aumentos diferentes de la imagen, reescalada a 224×224 píxeles



```

        horizontal_flip=True,
        zoom_range=0.4,
    )

batches = img_generator.flow_from_directory(
    path+'test_aug',
    target_size=(224,224),
    class_mode='categorical',
    shuffle=False,
    batch_size=4,
    save_to_dir=path+'augmentation',
)
# Iteracion sobre lotes para extraer los datos
# En este caso se sacan 4 veces la cantidad de datos originales
data = [batch.next() for _ in range(batches.nb_samples * 4)]

```

Ahora no es necesario modificar el modelo, solo reentrenarlo con los nuevos datos. Como ha ocurrido antes, al cambiar el conjunto de datos hay que transformarlo pasándolo por la red convolucional, usando en este caso la última versión con normalización por lotes.

Lo interesante en este caso es la reducción del sobreajuste que se ha obtenido.

```

Epoch 5/7
loss: 0.0768 - acc: 0.9817 - val_loss: 0.1142 - val_acc: 0.9538
Epoch 6/7
loss: 0.0639 - acc: 0.9895 - val_loss: 0.1212 - val_acc: 0.9334
Epoch 7/7
loss: 0.0708 - acc: 0.9860 - val_loss: 0.1139 - val_acc: 0.9520

dense_model.evaluate(test_feat, test_labels)
>>> [0.31255581648, 0.96723868954]

```

El modelo, que ahora ha predicho correctamente casi un 97% de las imágenes del conjunto de test, ha disminuido ligeramente la puntuación de pérdida logarítmica. Hay que tener en cuenta que a medida que esta puntuación tiende a cero, es más difícil disminuirla, ya que corresponde cada vez más a un modelo casi perfecto.

Cuando hablamos de un modelo casi perfecto, nos referimos respecto al conjunto de test elegido. En este caso, debido al pequeño tamaño del conjunto de test, la perfección del modelo estará lejos de la perfección del modelo real que va a ser probado en el reto.

La puntuación en la tabla pública de *Kaggle* para este modelo ha sido de **0.9364**, el mejor hasta ahora.

4.4. Modelo convolucional

Hasta ahora se ha seguido siempre el mismo esquema: una vez transformado el conjunto de datos entrenamos una red densa (completamente conectada) de tres capas. Además, una parte del preprocesamiento de las imágenes consiste en redimensionarlas a un tamaño más manejable, de 224×224 píxeles.

Una de las ventajas de las redes convolucionales es que permite aumentar el tamaño de las imágenes elegidas sin que por ello se incremente la complejidad de la red, ya que el número de pesos no cambiaría, solo el tamaño de la salida. Sin embargo, la complejidad de la red neuronal estándar que hemos colocado tras la red convolucional sí que vería incrementarse muchísimo su complejidad con el cambio de tamaño de las imágenes.

Para intentar aprovechar esta ventaja que ofrecen las redes convolucionales frente a las redes densas, se puede intentar trabajar usando solo capas convolucionales en todo el proceso, clasificando al final con una única capa de activación clásica.

Aquí se estaría desarrollando una idea que se comentaba al principio de este trabajo, el conseguir ocho filtros cuya salida sea la probabilidad de que en la zona de la imagen esté apareciendo el pez de la clase elegida.

Se usarán inicialmente tres capas de convoluciones, con 128 filtros cada una. La combinación de estos 128 filtros permitirá capturar la combinatoria de características que representa la red convolucional preentrenada. Al final se añade una última capa de convolución con 8 filtros, el mismo número de categorías en las que clasificar.

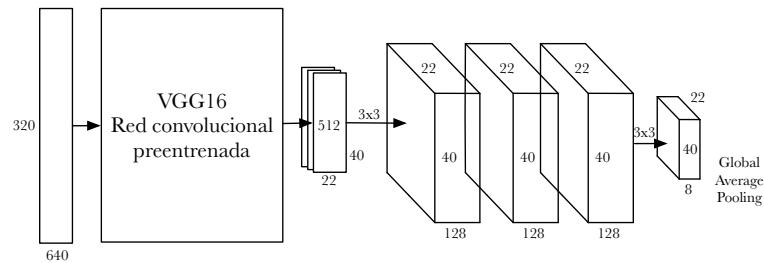
Las imágenes resultantes de estos filtros se transforman a una sola salida mediante una capa *GlobalAveragePooling2D*. Esta capa funciona como las capas POOL de la red convolucional, pero usando la media del total de los píxeles de la imagen, generando una salida de un valor por filtro que pueda ser entendida por la capa de activación.

La arquitectura final de la red se muestra en la figura 4.9.

```
def build_conv_layers():
    return [
        BatchNormalization(axis=1,
                           input_shape=conv_layers[-1].output_shape[1:])
    ],
    Convolution2D(128, 3, 3, activation='relu', border_mode='same'),
    BatchNormalization(axis=1),
    Convolution2D(128, 3, 3, activation='relu', border_mode='same'),
    BatchNormalization(axis=1),
    Convolution2D(128, 3, 3, activation='relu', border_mode='same'),
    BatchNormalization(axis=1),
    Convolution2D(8, 3, 3, border_mode='same'),

    # Output layer
    GlobalAveragePooling2D(),
    Activation('softmax'),
]
```

FIGURA 4.9: Arquitectura de la red completamente convolucional



De nuevo habrá que transformar el conjunto de datos por medio de la red convolucional preentrenada, ya que se va a usar un tamaño de imagen de 360×640 . Aunque, como se ha comentado, la red convolucional admite un aumento del tamaño de la imagen sin que crezca exponencialmente el tiempo de entrenamiento, el entrenamiento de una red convolucional es mucho más costoso que el de una red clásica. Esto se debe a que hay que actualizar n pesos por cada píxel de cada imagen. En este caso el entrenamiento de la red completamente convolucional ha tardado 24 veces más que el entrenamiento de la red densa del capítulo anterior.

```
conv_model.evaluate(test_feat, test_labels)
>>> [0.243715549991, 0.969645081488]
```

Al evaluar el modelo se ve que consigue una pequeña mejora respecto al modelo denso. La puntuación en la tabla pública de *Kaggle* para este modelo ha sido de **0.8109**.

4.4.1. Visualización del modelo

El trabajar en todo momento (salvo en la última capa) con redes convolucionales, implica que los pesos son siempre filtros que se aplican a imágenes. Se pretende entonces conseguir filtros que resalten la existencia en cada zona de la imagen de un pez de una determinada clase.

Se pueden usar los diferentes valores de los filtros para construir un mapa de calor para la categoría que le corresponde. Un mapa de calor es una representación gráfica de los valores de una matriz bidimensional. Generalmente valores pequeños se asocian a colores más claros, como el azul o el verde, y valores mayores a colores más cálidos, como el rojo o el morado.

Si se redimensiona el filtro (ya que ha sido reducido por las capas de *MaxPooling2D*) al tamaño de la imagen original, se puede ver cómo encuentra el pez a clasificar en cada imagen. Como ejemplo, al aplicar la red a la imagen de la figura 4.10 podemos visualizar el filtro perteneciente a su categoría (ALB: *Thunnus alalunga*) en la figura 4.11.

FIGURA 4.10: Una imagen correspondiente a la clase ALB: *Thunnus alalunga*

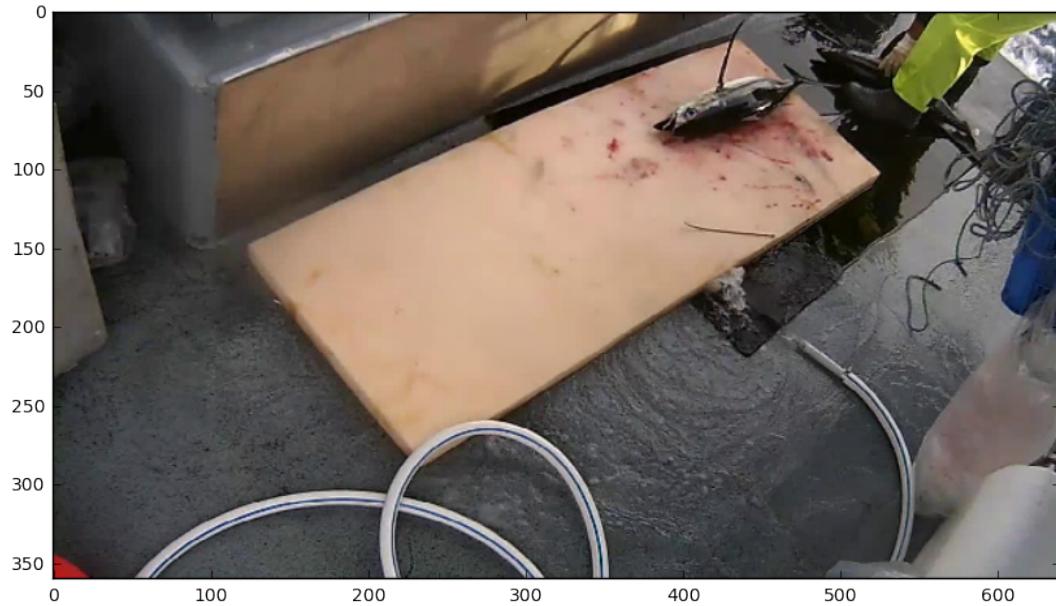


FIGURA 4.11: Aplicación del mapa de calor reescalado a la imagen de la figura 4.10

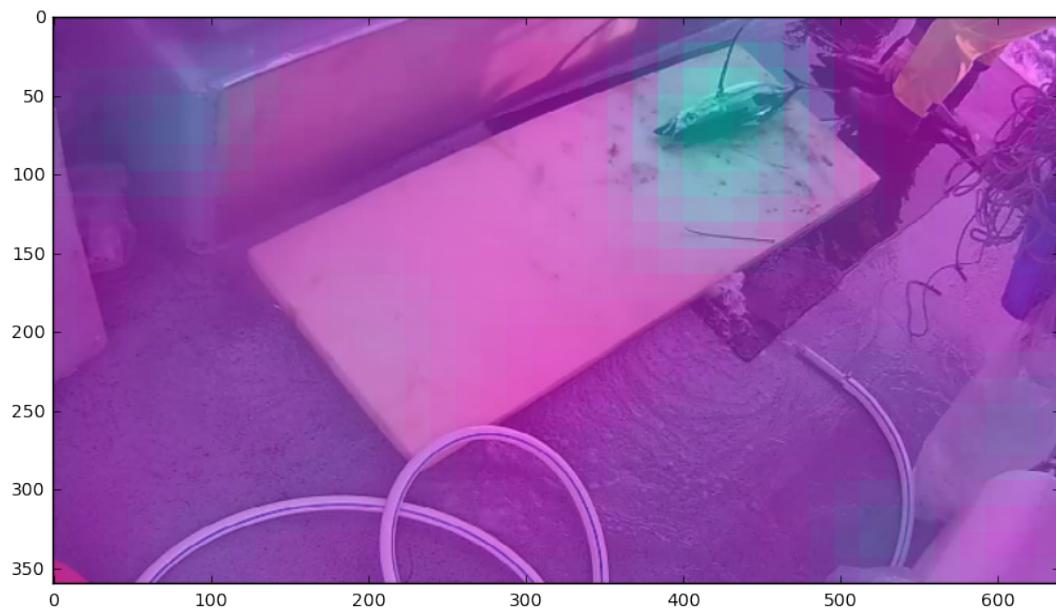


FIGURA 4.12: Una imagen correspondiente a la clase ALB: *Thunnus alalunga*



Esto no solo permite comprobar que el modelo funciona correctamente, sino también buscar los ejemplos que peor clasifica y analizar qué está señalando el mapa de calor correspondiente.

Un ejemplo claro de cómo se puede usar esta técnica de visualización para detectar los errores en el modelo es el intento de clasificación un atún de cola amarilla (YFT: *Thunnus albacares*), en la imagen de la figura 4.12. Una característica de los atunes de cola amarilla es su banda de tonos amarillos en el lomo, característica que el modelo podría haber aprendido gracias a los ejemplos. Sin embargo, la imagen muestra el atún visto desde abajo, con las aletas abiertas. Esto esconde una de las principales características del atún, siendo difícil clasificarlo incluso para un humano no familiarizado con atunes.

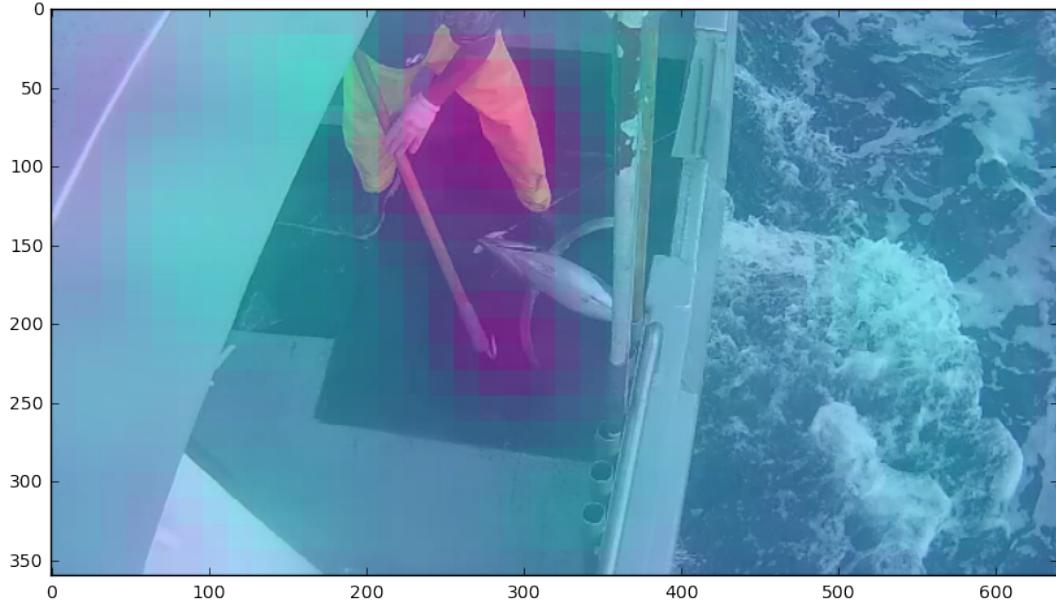
Al mostrar el mapa de calor correspondiente (figura 4.13) se puede apreciar que la hipótesis de que el modelo ha aprendido a detectar los atunes de cola amarilla a través del color podría ser cierta, ya que una de las partes más importantes de la foto para el filtro son los pantalones del pescador, de un color amarillo fuerte.

Esto permite tomar ciertas decisiones sobre el entrenamiento del modelo. Por ejemplo, si se ve que la fijación del modelo con el color es un error y es preferible que use otras cosas como formas o texturas, se puede eliminar el color de las imágenes o aumentar los datos usando transformaciones de color que hagan especial hincapié en la varianza del color amarillo.

4.5. Modelo con entrada múltiple

Como se vió en el análisis del conjunto de datos (sección 3.2.4), hay otras propiedades de las imágenes cuyo uso podría permitir una mejora de los modelos. Por

FIGURA 4.13: Aplicación del mapa de calor reescalado a la imagen
4.10



CUADRO 4.1: Ocurrencia de los diferentes tamaños de imagen

Tamaño de la imagen	Cantidad de imágenes
1192×670	169
1276×718	192
1244×700	23
1280×720	1880
1280×750	520
1280×924	51
1280×974	344
1334×750	28
1518×854	37
1732×974	33

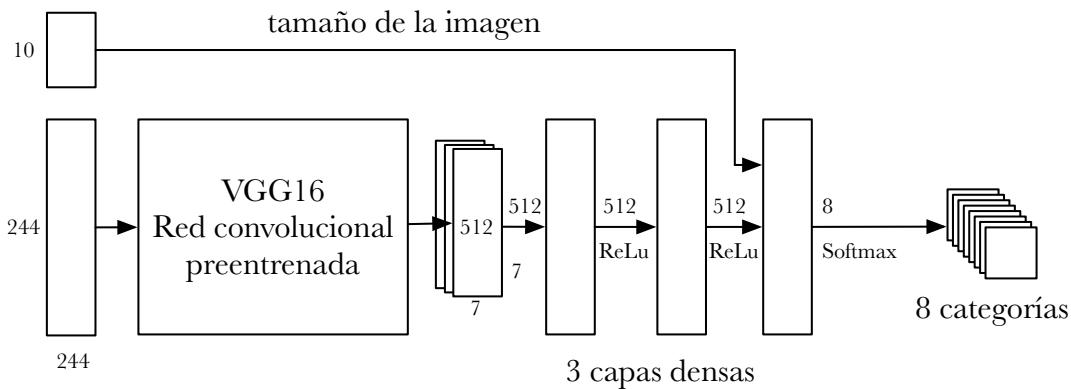
ejemplo, las imágenes están tomadas en diferentes barcos, con diferentes modelos de cámaras. No todas las cámaras generan imágenes del mismo tamaño, por lo que se podría averiguar el barco del que se ha tomado la fotografía por el tamaño de la imagen.

Ya que cada barco suele pescar diferentes tipos de peces con mayor o menor probabilidad, el conocer el tipo de barco puede ser aprovechado por el modelo para balancear las posibilidades de cada categoría dependiendo del tipo de pesca que suela llevar.

La idea no es hacer todo esto a mano, sino entrenar un modelo que tenga en cuenta tanto las imágenes como otra información válida; en este caso, cual es el tamaño de la imagen. En el conjunto de datos hay diez tamaños diferentes de imágenes, las cuales se codificarán en *onehot* para facilitar el entrenamiento. Los diferentes tamaños de las imágenes son los indicados en el cuadro 4.1.

Para la arquitectura del modelo vamos a usar la red densa, con la diferencia de que la última capa tendrá dos entradas: la salida de las capas ocultas de la red neuronal

FIGURA 4.14: Arquitectura del modelo con entrada múltiple



y la clasificación del tamaño de la imagen, codificado en *onehot*, como se muestra en la figura 4.14.

La estructura del código es ligeramente diferente, ya que ahora vamos a generar todas las capas que teníamos anteriormente menos la última en una misma función:

```
def build_dense_layers(dropout=0.5):
    # Usamos las entradas de la red convolucional, como antes
    inp = Input(conv_layers[-1].output_shape[1:])
    x = MaxPooling2D()(inp)
    x = BatchNormalization(axis=1)
    x = Flatten()
    x = Dropout(dropout/4)
    x = Dense(512, activation='relu')
    x = BatchNormalization()
    x = Dropout(dropout)
    x = Dense(512, activation='relu')
    x = BatchNormalization()
    x = Dropout(dropout/2)
    return x
```

Usando esta función, añadimos una capa final para la clasificación, cuya entrada será la unión entre la red densa y la información sobre los tamaños de las imágenes. Esta capa deberá clasificar cada imagen usando tanto la información disponible de la red densa como el barco al que pertenece.

```
# Generar las capas de la red
net = build_dense_layers()

# Usar las entradas de las dimensiones de cada elemento
size_inputs = Input(len(sizes))
bn_inputs = BatchNormalization()(size_inputs)

# Unir las dos entradas en una sola capa
net = merge([net, bn_inputs], 'concat')
```

```
net = Dense(8, activation='softmax')(net)

# Crear la red
dense_model = keras.models.Model(
    [inp, size_inputs], # El modelo ahora tiene multientrada
    net
)
```

Tras construir el modelo con las entradas múltiples y entrenarlo sobre los dos conjuntos de entrenamiento (las imágenes y los tamaños de las imágenes) vemos la evaluación del modelo.

```
dense_model.evaluate(test_feat, test_labels)
>>> [0.373333961543, 0.941351055711]
```

El modelo ha sacado una evaluación peor que modelos anteriores, incluso usando un modelo que ya se conocía con una evaluación superior.

Aunque la idea es buena, lo que está haciendo por debajo es clasificar los diferentes barcos en base a los tamaños de sus imágenes. Debido a la amplia diferencia de las imágenes de los diferentes tipos de barcos, ya sea por los colores, iluminación, objetos, etc, el anterior modelo denso probablemente ya esté teniendo en cuenta de qué barco es cada imagen. Por lo tanto lo único que hace el introducir estos datos es añadir complejidad del modelo, haciendo más difícil que converja a una buena solución.

El envío de este modelo ha obtenido una puntuación de **1.0466** en la tabla pública de *Kaggle*, empeorando el anterior envío.

4.5.1. Fuga de datos

En las competiciones de modelos predictivos como *Kaggle* a veces se usan datos no diferentes al conjunto de datos para clasificar. En este caso hemos usado los tamaños de las imágenes, pero podría usarse también información oculta en los metadatos de la imagen, como coordenadas GPS, hora de toma de la fotografía o marca de la cámara. Esto se conoce como **fuga de datos**.

No todos estos datos son publicados de una manera voluntaria por comunidades de campeonatos, pero son útiles para conseguir una pequeña mejora en el modelo, siempre que el modelo puede aprovecharse de estos datos.

4.6. Salida múltiple

Como vimos anteriormente en la figura 4.13, la red completamente convolucional es capaz de clasificar correctamente la imagen pero por las razones incorrectas: los pantalones amarillos del pescador. Un modelo que sea capaz de clasificar la imagen identificando la localización del pez podría evitar este tipo de errores. Intuitivamente

esta idea tiene sentido, ya que es lógico buscar primero si hay un pez en la imagen y luego distinguir a qué categoría pertenece.

Al usar técnicas de aprendizaje supervisado como las que estamos usando es necesario disponer de ejemplos con respuesta conocida para poder entrenar los modelos. Si queremos un modelo que localice peces en las imágenes necesitaremos ejemplos con localizaciones conocidas para el conjunto de entrenamiento, cosa que el conjunto de datos no posee.

Afortunadamente, un usuario ha localizado todos los peces del conjunto de entrenamiento en rectángulos y ha publicado un conjunto de datos con las coordenadas de los rectángulos que los contienen. Las reglas de la competición en *Kaggle* indican que es posible usar conjuntos de datos no oficiales generados por otras fuentes siempre que estos se hagan públicos y estén disponibles para todos los participantes.

Usando esta información se va a hacer un modelo que para cada imagen genere una salida múltiple: la clasificación en una de las ocho categorías y las coordenadas del rectángulo que contiene al pez.

El conjunto de datos consiste en un archivo *json* con la siguiente estructura.

```
{
  "annotations": [
    {
      "class": "rect",
      "height": 79.0,
      "width": 256.0,
      "x": 815.0,
      "y": 124.0
    }
  ],
  "class": "image",
  "filename": "img_07795.jpg"
},
```

Cada una de las imágenes posee una lista de rectángulos, definidos por su altura, anchura y las coordenadas del centro. Algunas imágenes poseen varios peces, por lo que la lista de rectángulos puede contener más de un elemento.

Para ilustrar estos rectángulos y comprobar que están señalando la localización correctamente se ha pintado un rectángulo con las mismas coordenadas en la figura 4.15. Se puede observar que captura con precisión la localización del pez.

4.6.1. Arquitectura del modelo

Generar la caja que contiene al pez no es un problema de clasificación, porque no tiene que distinguir entre diferentes tipos de peces, solo tiene que identificar dónde hay uno. Esto favorecerá que la red busque características generales de peces, en

FIGURA 4.15: Representación de un rectángulo del conjunto de datos auxiliar que ubica cada pez en su imagen

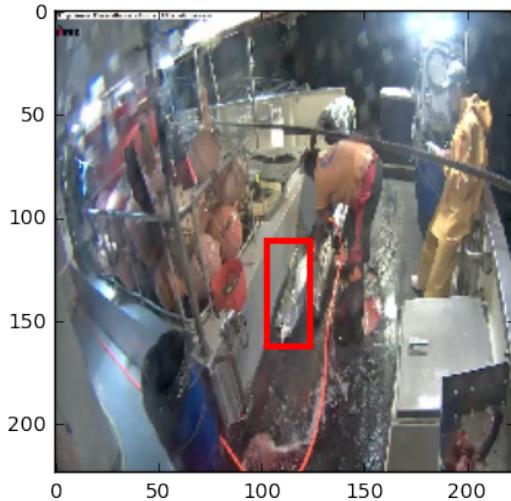
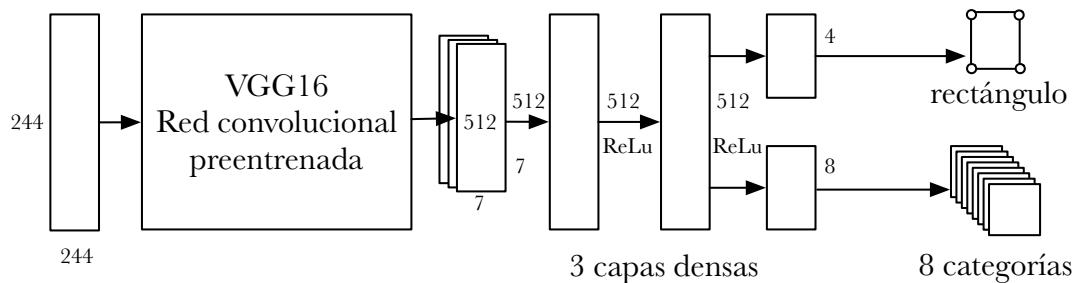


FIGURA 4.16: Arquitectura del modelo denso con salida múltiple



vez de características generales de cada una de las categorías. Por otro lado, encontrar la clasificación correcta sí que requiere buscar características específicas de cada categoría.

Al usar la misma red para ambas tareas, esta deberá encontrar un balance entre encontrar características específicas de cada categoría y características generales de los peces.

Para la creación de la red vamos a seguir el mismo procedimiento que en la entrada múltiple (sección 4.5), solo que en vez de añadir la entrada extra en la última capa, dividiremos esta última capa en dos redes diferentes, como se muestra en la figura 4.16.

La primera capa de salida, de localización, no posee una función de activación ya que las cuatro salidas representan las cuatro propiedades con las que representamos el rectángulo (x , y , altura y anchura) y es necesario que la red indique el valor de cada una de esas cuatro propiedades.

La segunda capa es la que hemos usado hasta ahora: ocho salidas representando la codificación *onehot* de las ocho categorías.

Generar la red sin la capa de salida

CUADRO 4.2: Resultados de la evaluación del modelo de salida múltiple sobre el conjunto de test

	Pérdida	Precisión
Localización	239.54	0.8454
Clasificación	0.2403	0.9698

```
net = build_dense_layers(dropout)
# Agregar la capa de salida para la localizacion
net_bb = Dense(4, name='bb')(net)
# Agregar la capa de salida para la clasificacion
net_class = Dense(8, activation='softmax', name='class')(net)
# Generar el modelo usando la entrada + ambas capas de salida
model = Model([inp], [net_bb, net_class])
```

Al entrenar el modelo es necesario indicar de dónde sale el valor a optimizar en cada una de las salidas. Para la nueva salida vamos a usar el error cuadrático medio (*MSE*, en inglés) entre los valores del conjunto de datos de rectángulos y los valores generados. El *MSE* consiste en la media del cuadrado de la diferencia entre los valores esperados y los valores de la salida. Esto penalizará mucho más los valores lejanos a la salida esperada que los pequeños errores.

Ya que el error cuadrático será de una magnitud mayor que la pérdida logarítmica, se pondrá cada una con diferentes pesos.

```
model.compile(
    Adam(lr=0.001),
    loss=['mse', 'categorical_crossentropy'],
    metrics=['accuracy'],
    loss_weights=[.001, 1.], # Ponderar los errores
)
```

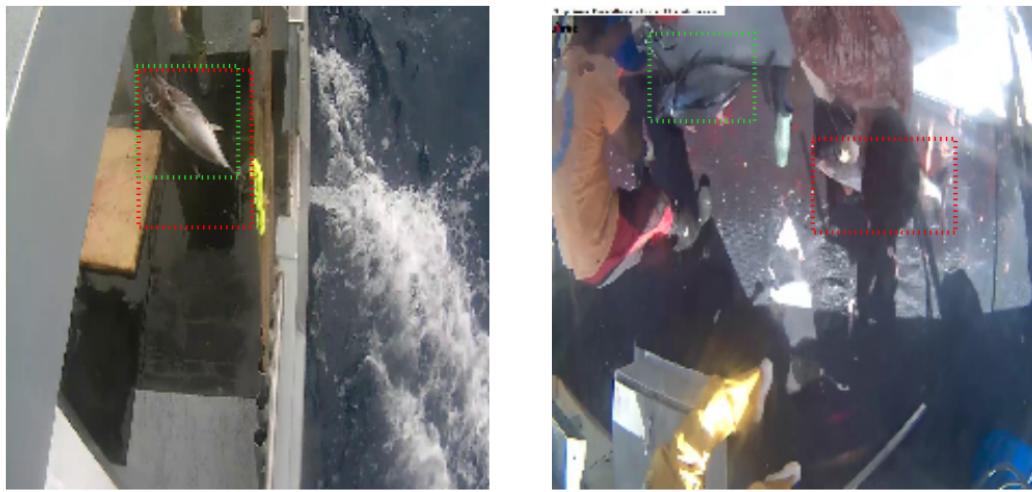
Ahora hay que proporcionar los ejemplos de las salidas de cada uno de los conjuntos de datos para cada capa de salida, así como para el conjunto de validación.

```
model.fit(
    conv_feat,
    [trn_bbox, trn_labels],
    batch_size=batch_size,
    nb_epoch=7,
    validation_data=(conv_val_feat, [val_bbox, val_labels]),
)
```

La evaluación del modelo sobre el conjunto de test generará una evaluación para cada capa de salida. En el cuadro 4.2 podemos ver los valores de cada una de las capas de salida.

La precisión de la capa de clasificación, que es la que vamos a usar para la competición, es la más alta hasta ahora. Esto puede deberse a que, como decíamos al

FIGURA 4.17: En rojo, el rectángulo correspondiente a la localización del pez en el conjunto de datos alternativo. En verde, el rectángulo generado por el modelo con salida múltiple.



principio de esta sección, tiene que conseguir características para distinguir peces así como características de cada categoría.

La puntuación en la tabla pública de *Kaggle* para este modelo ha sido de **0.8675**, siendo el segundo mejor modelo de los entrenados hasta ahora.

4.6.2. Visualización de la localización de peces

Como se veía en la imagen 4.15 se puede representar la localización anotada en el conjunto de datos de cada imagen. De la misma manera, podemos visualizar los valores de salida de la capa de localización para ver si el modelo es capaz de encontrar dónde se encuentra cada pez en la imagen.

En la figura 4.17 podemos ver en rojo la localización original y en verde la generada por el modelo. Aunque no es perfecta, captura aproximadamente dónde se encuentra el pez. Es curioso notar que en la primera de las dos imágenes solo está encontrando las características más visibles del pez. La cola, que casi no es distingible, no se incluye dentro del rectángulo generado por el modelo.

En la segunda imagen el modelo ha sido capaz de identificar un pez, pero no el que se le había indicado en el conjunto de entrenamiento. El pez que ha localizado es más visible y está mejor iluminado, por lo que es más fácil de localizar. El pez original está parcialmente tapado por la cabeza de un pescador.

4.7. Modelos *ensemble*

En competiciones de modelos predictivos es habitual que la mejor puntuación no la obtenga un único modelo, sino un modelo construido combinando varios. Este tipo de modelos se conocen como *ensemble models* (modelos de conjunto).

CUADRO 4.3: Predicciones de diferentes modelos para una imagen, unidas en un modelo final mediante una media aritmética

Modelo	ALB	BET	DOL	LAG	NoF	OTHER	SHARK	YFT
Salida múltiple	0.010	0.052	0.030	0.001	0.123	0.079	0.046	0.875
Red base	0.020	0.001	0.030	0.002	0.123	0.079	0.046	0.844
Red convolucional	0.030	0.301	0.029	0.001	0.123	0.079	0.046	0.502
Ensemble	0.020	0.118	0.030	0.001	0.123	0.079	0.046	0.740

Para generar estos modelos no es necesario reentrenar los modelos disponibles, sino trabajar con las salidas de estos. Primero calculamos las predicciones de cada modelo que queremos combinar. Luego generamos una predicción del modelo *ensemble* agregando los valores de las predicciones. Hay otros métodos para construir modelos *ensemble* más sofisticados, pero aquí se van a considerar únicamente los más simples:

- **Votación:** La salida de cada modelo se entiende como una votación de a qué categoría debería pertenecer dicha entrada. La salida del modelo final es la elección ganadora de esa votación. Por ejemplo, si el modelo 1 ha elegido ALB y los modelos 2 y 3 han elegido YFT, la salida del modelo *ensemble* será YFT.
- **Media ponderada:** En el caso de que la salida sean probabilidades de cada clase se puede usar una media ponderada entre los valores de los modelos. La ponderación de la media permite dar preferencia al modelo con más confianza. En el cuadro 4.3 podemos ver un ejemplo de cómo sería la salida de un modelo *ensemble* ejemplo usando una media aritmética entre tres modelos diferentes.

Debido a que los modelos presentados generan un vector de probabilidades para cada categoría, los modelos *ensemble* generados usarán la media ponderada.

Usar la pérdida logarítmica como métrica hace que se castigue mucho las bajas probabilidades en las categorías correctas. Usar la media para estos modelos puede ayudar a evitar este comportamiento, ya que solo tendrán probabilidades cercanas a cero aquellas categorías que sean casi cero en todos los modelos.

Un ejemplo puede verse en la categoría LAG del cuadro 4.3. Todas los modelos predicen una baja puntuación para esa categoría, por lo que la puntuación final también es muy baja. Sin embargo en la categoría BET existe un modelo que le da una puntuación media, haciendo que esta se aleje de cero. En cuanto uno de los modelos no está de acuerdo en que la puntuación sea cero, esta se aleja.

4.7.1. Resultados

Se han presentado dos modelos diferentes usando esta técnica. Ambos modelos están compuestos de tres modelos anteriores:

- Modelo con clasificador denso, dropout, normalización por lotes y aumento de datos.
- Modelo completo convolucional

- Modelo con salida múltiple

Estos tres modelos son los que mejores resultados obtuvieron previamente, así que se usarán para generar dos modelos. El primero usará la media aritmética de la salida mientras que el segundo será ponderado de la siguiente manera: 25 % para el clasificador denso, 25 % para el modelo con salida múltiple y 50 % para el modelo completo convolucional.

Al evaluar estos modelos en *Kaggle*, los resultados de la tabla pública son los siguientes

- Media aritmética sobre los tres modelos: **0.9175**
- Media ponderada favoreciendo el modelo completo convolucional: **0.8975**

Desafortunadamente las puntuaciones de *ensemble*, que tradicionalmente suelen conseguir las mejores posiciones en las competiciones de *Kaggle*, no han conseguido ser las mejores.

4.8. Modelos finales presentados

En los cuadros 4.4 y 4.5 se presenta un resumen de los modelos presentados, junto a sus puntuaciones.

Las reglas de la competición establecen que solo es posible puntuar dos modelos en la tabla de puntuaciones privada y solo es posible usar un modelo para la segunda fase.

El mejor modelo según las puntuaciones públicas de la primera fase es el que usa un clasificador convolucional en las últimas capas, seguido del que usa una salida múltiple para intentar clasificar los peces. Hay que recordar que estas puntuaciones se calculan usando un subconjunto del conjunto de test, por lo que puede cambiar cuando se publique la tabla privada.

Tradicionalmente los modelos *ensemble* consiguen mejores puntuaciones en las competiciones de *Kaggle*, por lo que se ha decidido enviar un modelo *ensemble* (el tercero en mejor puntuación) y el convolucional completo.

Al publicarse las puntuaciones privadas, vemos en el cuadro 4.5 que el modelo convolucional consigue mejor puntuación, así que es seleccionado como el modelo final a enviar.

La puntuación final de este modelo es de **0.907** en la tabla pública y **1.998** en la privada, consiguiendo una posición 210 de 2293 participantes y una medalla de bronce (por clasificar dentro del mejor 10 % de los participantes).

CUADRO 4.4: Identificadores de los modelos

Id	Modelo
vgg	VGG16 con ajuste fino
v-512	VGG16 con capas de 512 salidas
v-norm	VGG16 con normalización por lotes (+ 512 s, <i>dropout</i>)
v-aug	VGG16 con aumento de datos (+ 512 s, <i>dropout</i> , norm.)
v-fcn	VGG16 con red convolucional (+ norm.)
m-in	VGG16 con entrada múltiple (+ 512 s, <i>dropout</i> , norm, datos)
m-out	VGG16 con salida múltiple (+ 512 s, <i>dropout</i> , norm, datos)
e-arit	<i>Ensemble</i> con media aritmética
e-pond	<i>Ensemble</i> con media aritmética ponderada

CUADRO 4.5: Identificadores de los modelos

Id	Local	1 ^a fase		2 ^a fase	
		Pública	Privada	Pública	Privada
vgg	2.383	2.632	-	-	-
v-512	1.272	2.381	-	-	-
v-norm	0.451	1.017	-	-	-
da-aug	0.312	0.936	-	-	-
fcn	0.243	0.811	0.770	0.907	1.998
m-in	0.373	1.047	-	-	-
m-out	0.240	0.867	-	-	-
e-arit	-	0.917	-	-	-
e-pond	-	0.897	0.841	-	-

Capítulo 5

Conclusiones

El objetivo principal de este trabajo era afrontar un problema real en una de las competiciones ***featured*** de *Kaggle*, incluso aspirar a conseguir un premio.

Ha sido un trabajo muy interesante, ya que he aprendido las dificultades de la implementación de redes de aprendizaje profundo en problemas del mundo real. Esto requiere enfrentarse a problemas que no aparecen en la teoría.

Los tres principales problemas han sido los siguientes:

- **Cómo afrontar el problema:** Hasta el momento había leído bastante sobre técnicas de reconocimiento visual usando herramientas fijas que permitían poca soltura y por supuesto sin usar ningún tipo de aprendizaje automático.

Esto se solucionó gracias a la gran comunidad de participantes de *Kaggle*. Es de esperar que en una competición con buenos premios existan reticencias a ayudar al resto de participantes, pero resultó ser al revés. Hay una comunicación permanente en los foros de la competición de las técnicas que se están usando, referencias, lecturas, etc. Quiero destacar aquí las aportaciones de Felix Yu (Yu, 2017) y de Jeremy Howard (Howard, 2017). Incluso publicando después de haber terminado la competición he entendido otros pasos a seguir que estaba perdiendo.

- **Trabajar con grandes cantidades de datos:** Los 13000 ejemplos de la segunda fase de la competición hicieron que muchos de mis modelos no funcionasen, ya que la máquina se quedaba sin suficiente memoria RAM. Esto me hizo tener que **cuidar la gestión de memoria**, saber descargar datos cuando no se están usando y guardar cálculos en disco para no tener que repetirlos.
- **Automatización de procesos:** La segunda fase de la competición tenía un conjunto de datos de prueba desconocido que iba a tener que ser clasificado por un modelo fijo. Esto hizo tener que automatizar toda la parte del tratamiento de datos para el problema.

La conclusión principal que saco de este trabajo es la potencia de las redes convolucionales. Han mejorado cualquier expectativa que tenía sobre el reconocimiento de imágenes y es posible que vayan a convertirse en la base de la visión artificial en el futuro.

Creo que se pueden abordar muchos problemas diferentes de clasificación de imágenes usando una arquitectura común que use las modificaciones realizadas en este problema (*dropout*, normalización por lotes, aumento de datos, etc).

5.1. Kaggle

Participar en esta competición de Kaggle ha sido divertido y didáctico ya que, aunque hubiera una competición por un premio, los participantes siempre han estado dispuestos a ayudarse entre ellos. Creo que dedicarle tiempo a una competición tiene sus recompensas, independientemente si se gana o no el premio.

Por otro lado, he notado que a la organización promotora (*The Nature Conservancy*) le falta un poco de experiencia en *Kaggle*. Esto se ha notado en diferentes detalles. La comunicación en el foro de la plataforma no ha sido del todo fluida, lo cual ha provocado algunos errores que no tenían que haberse cometido, como por ejemplo descalificación del mejor clasificado.

Otro ejemplo es la falta de diversidad del conjunto de datos de entrenamiento. Era muy difícil extraer información de determinados peces con muy poca información, haciendo complicado hacer modelos que generalizaran correctamente.

Un ejemplo de esto es que el archivo de ejemplo de envío (sección 3.1), que contiene en cada elemento la misma predicción (la proporción de ejemplos de cada categoría, normalizado) haya conseguido medalla de plata, acabando en la posición 33.

5.1.1. Puntuación final

Aunque se le haya dado varias vueltas a usar las capas densas como clasificadores finales del modelo, el ganador ha sido el que no las usa: el modelo completamente convolucional. Creo que esto puede ser debido a que es capaz de, en esas últimas capas convolucionales, detectar elementos más generales de la imagen, descartando así el ruido que pueden producir los fondos de la imagen.

5.2. Siguientes pasos

Las siguientes mejoras de este problema creo que vienen reduciendo el ruido generado por todos los elementos de las imágenes que no son peces. Para esto habría primero que recortar automáticamente los peces de las imágenes y entrenar un nuevo modelo con dichas imágenes, pero para esto habría que tener un buen modelo de localización, ya que errar en la localización hará que se clasifiquen ejemplos incorrectamente.

Por otro lado creo que también existe una posible mejora usando diferentes conjuntos de datos, como imágenes de peces sacadas de otras fuentes. De esta manera el modelo sería capaz de aprender características de los peces que no aparecen en los ejemplos.

Apéndice A

Infraestructura

A.1. Requisitos

Al principio de este trabajo se comentó que era necesario un enfoque de desarrollo rápido para probar diferentes ideas y encontrar rápidamente los errores y mejoras. Los algoritmos de aprendizaje sobre redes convolucionales son tradicionalmente costosos en cómputo, por lo que es una dificultad a superar si queremos una evolución rápida de la solución.

A.2. Hardware

Keras permite aumentar la velocidad de entrenamiento usando el procesador gráfico del sistema, siempre que este sea compatible con CUDA (NVIDIA, 2017). Para este problema se han usado dos equipos diferentes, ambos compatibles con CUDA.

Para la mayoría de entrenamientos y experimentos se ha usado un ordenador con las siguientes características:

- Procesador: Intel i3, 2.7 GHz
- Procesador gráfico: Nvidia GeForce GTX 950, 2GB de memoria
- 16 GB de RAM

Este equipo era suficiente para la mayoría de entrenamientos. Sin embargo cuando era necesario entrenar un conjunto de datos más grande, como la segunda fase de la clasificación, con 13000 ejemplos, el equipo tardaba demasiado en entrenar algunos modelos.

Por eso se optó por buscar una solución en la nube que permitiese alquilar una potencia gráfica superior durante tiempo limitado. Al final se ha optado por las máquinas virtuales de *Amazon AWS*, que permite contratar máquinas por horas. Una de ellas, identificada *p2.xlarge*, ofrece las siguientes características:

- Procesador: Intel i7, 3.5 GHz, 4 núcleos

- Procesador gráfico: Nvidia Tesla K80
- 61 GB de RAM

Este equipo, con un precio de 0.80 euros la hora, ofrece potencia más que suficiente para entrenar grandes cantidades de datos rápidamente.

Apéndice B

Keras

Keras es una API escrita en *Python* para el desarrollo de redes neuronales capaz de ejecutarse en sistemas como *TensorFlow* o *Theano*. Fue desarrollada con la idea de permitir una experimentación sencilla y un desarrollo rápido con redes neuronales. Posee soporte para redes neuronales recurrentes y convolucionales, así como combinaciones de ambas. También es capaz de aprovechar la GPU para acelerar el entrenamiento de las redes.

B.1. Instalación

Para la instalación de *Keras* primero es necesario instalar uno de los diferentes *frameworks* con los que *Keras* trabaja: *Theano* o *TensorFlow*. En este caso se ha trabajado con *Theano*, ya que es un sistema más antiguo y que dispone de más documentación. Para la instalación se usará *Conda*, un gestor de paquetes para *Python*. Otros métodos de instalación y posibles soluciones a problemas se pueden encontrar en la documentación de *Theano* (*Theano*, 2017).

```
>>> conda install theano pygpu
```

Una vez instalado *Theano* es necesario instalar *Keras*, también usando *Conda*.

```
>>> conda install keras
```

B.2. Uso

El uso de la herramienta se puede dividir en tres partes: generación de la red neuronal, carga del conjunto de datos y entrenamiento y evaluación del modelo. Aquí se explicarán las partes más usadas en este trabajo, pero es posible obtener una descripción más precisa en la documentación de *Keras* (Chollet, 2017).

B.2.1. Arquitectura

Lo más importante del uso de *Keras* es ser capaz de definir la arquitectura de la red. Para ello existe la clase `Sequential`. Esta clase representa un conjunto lineal de capas dentro de la red. Como ya hicimos en la sección 4.3, vamos a crear un objeto `Sequential` pasando las capas de las que se compondrá la red mediante objetos.

```
from keras.models import Sequential
red_neuronal = Sequential([
    Dense(512, activation='relu'),
    Dense(512, activation='relu'),
    Dense(8, activation='softmax')
])
```

Existen muchos tipos de objetos capa, cada uno representando una variación de un concepto diferente en la red neuronal. Algunos de los usados en este trabajo son los siguientes:

- `Dense`: Capa densa, completamente conectada. Las capas usadas en las redes neuronales clásicas.
- `Convolution2D`: Capa convolucional. Existen con 1, 2 ó 3 dimensiones.
- `Dropout`: Aplicación de dropout (sección 4.3.2) a la salida de la capa anterior.
- `MaxPooling2D`: Capa de muestreo. Elige la mayor de cada cuatro entradas que entran.
- `BatchNormalization`: Capa de normalización por lotes.

Cada capa incluye el tamaño de salida de la capa y una función de activación, que debe indicarse en el parámetro `activation`.

Por último, para terminar de crear la red, hay que configurar el aprendizaje de esta. Esto se hace usando la utilidad `compile`, que necesita tres parámetros:

- Optimizador: Mecanismo de optimización, como por ejemplo SGD (descenso de gradiente estocástico).
- Función de pérdida: Este es el objetivo que el modelo tratará de minimizar. En nuestro caso será '`logloss`', la pérdida logarítmica.
- Métricas: Las diferentes métricas que queremos que muestre en la evaluación, además de la función de pérdida

```
red_neuronal.compile(
    optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'],
)
```

B.2.2. Carga de datos

Para este proyecto hemos tenido que cargar grandes cantidades de imágenes. Keras necesita los datos de entrenamiento como matrices de la biblioteca numérica *numpy*, sin embargo las imágenes están en formato *jpg*.

La utilidad para convertir imágenes a matrices se llama `ImageDataGenerator`. Lo interesante de esta utilidad es que no solo permite leer archivos y convertirlos, sino que lo hace como un generador. Un generador en *Python* va devolviendo más elementos a medida que estos son consumidos. Esto evita tener que generar todos las matrices al mismo tiempo y cargarlas en memoria. También permite aplicar otras transformaciones a la imagen para crear un aumento de datos.

```
from keras.preprocessing.image import ImageDataGenerator
base_generator = ImageDataGenerator(featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    zca_epsilon=1e-6,
)
```

Una vez configurada la herramienta hay que llamarla en el directorio sobre el que se quiere actuar.

```
image_generator = base_generator.flow_from_directory(
    'data/train',
    target_size=(150, 150),
    batch_size=32,
```

B.2.3. Entrenamiento y evaluación

El entrenamiento del modelo se realiza con la función `fit`. Es necesario proveer a la función de los ejemplos de entrada con sus salidas esperadas (`data` y `labels` en nuestro ejemplo). Se le indica también el número de iteraciones a realizar.

```
data, labels = image_generator.flow_from_directory(...)
red_neuronal.fit(data, labels, epochs=10, batch_size=32)
```

Esto actualizará los pesos de la red para minimizar la función de pérdida indicada. Una vez finalizado podemos hacer una evaluación del modelo usando otro conjunto de datos (generalmente el de test) con la función `evaluate`. Esta función devolverá la pérdida del modelo sobre el conjunto más el resto de métricas que se hayan definido.

```
red_neuronal.evaluate(test_data, test_labels)
>>> [1.90571456890184755 , 0.66099843993759748]
```

Para una mayor compresión de como usar esta librería es mejor dirigirse a la documentación (Chollet, 2017).

Bibliografía

- Chollet, François (2017). *Documentación de Keras*. <https://keras.io/>.
- Howard, Jeremy (2017). *Practical Deep Learning For Coders*. <http://course.fast.ai/>.
- ILSVRC (2017). *Large Scale Visual Recognition Challenge*. <http://www.image-net.org/challenges/LSVRC/>.
- ImageNet (2017). *Image database*. <http://www.image-net.org/>.
- Ioffe, Sergey y Christian Szegedy (2015). «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift». En: *CoRR* abs/1502.03167. URL: <http://arxiv.org/abs/1502.03167>.
- Krizhevsky, Alex y col. (2012). «Imagenet classification with deep convolutional neural networks». En: *Advances in neural information processing systems*, págs. 1097-1105.
- LeCun, Yann y Corinna Cortes (2010). «MNIST handwritten digit database». En: URL: <http://yann.lecun.com/exdb/mnist/>.
- Mitchell, Thomas M. (1997). *Machine Learning*. 1.^a ed. New York, NY, USA: McGraw-Hill, Inc. ISBN: 0070428077, 9780070428072.
- NVIDIA (2017). *Documentación de CUDA*. http://www.nvidia.com/object/cuda_home_new.html.
- Powell, Victor (2015). *Image kernels*. <http://setosa.io/ev/image-kernels/>.
- Pérez, Fernando (2017). *Cuadernos de Jupyter*. <http://jupyter.org/>.
- Russakovsky, Olga y col. (2014). «ImageNet Large Scale Visual Recognition Challenge». En: *CoRR* abs/1409.0575. URL: <http://arxiv.org/abs/1409.0575>.
- Russell, Stuart y Peter Norving (2004). *Inteligencia Artificial. Un Enfoque Moderno. Segunda edición*. Pearson Educación, S. A. Madrid.
- Simon, Marcel, Erik Rodner y Joachim Denzler (2016). «ImageNet pre-trained models with batch normalization». En: *CoRR* abs/1612.01452. URL: <http://arxiv.org/abs/1612.01452>.
- Simonyan, Karen y Andrew Zisserman (2014). «Very Deep Convolutional Networks for Large-Scale Image Recognition». En: *CoRR* abs/1409.1556. URL: <http://arxiv.org/abs/1409.1556>.
- Snow, John (1855). *On the mode of communication of cholera*. John Churchill.
- Srivastava, Nitish y col. (2014). «Dropout: A simple way to prevent neural networks from overfitting». En: *The Journal of Machine Learning Research* 15.1, págs. 1929-1958.
- Taigman, Yaniv y col. (2014). «DeepFace: Closing the Gap to Human-Level Performance in Face Verification». En: *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '14. Washington, DC, USA: IEEE Computer Society, págs. 1701-1708. ISBN: 978-1-4799-5118-5. DOI: [10.1109/CVPR.2014.220](https://doi.org/10.1109/CVPR.2014.220). URL: <http://dx.doi.org/10.1109/CVPR.2014.220>.
- The Nature Conservancy Fisheries* (2016). *Kaggle challenge*. <https://www.kaggle.com/c/the-nature-conservancy-fisheries-monitoring>.

Theano (2017). Documentación de Theano. <http://deeplearning.net/software/theano/index.html>.

Yu, Felix (2017). Fine-tune Convolutional Neural Network in Keras with ImageNet Pretrained Models. https://github.com/flyyufelix/cnn_finetune.