Universidad de Sevilla

Trabajo fin de máster

Fish recognition using deep learning

Autor:
Marco Herrero

Tutor:
Álvaro Romero Jiménez

Máster Universitario de Lógica, Computación e Inteligencia Artificial

Departamento de ciencias de la compucación e inteligencia artificial

June 12, 2017

"*Esta página venía en la plantilla y probablemente la borre*"

Marco Herrero

Universidad de Sevilla

Abstract

Faculty Name

Departamento de ciencias de la compucación e inteligencia artificial

Máster Universitario de Lógica, Computación e Inteligencia Artificial

Fish recognition using deep learning

by Marco Herrero

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or

Physical Constants

Speed of Light $\quad c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

For/Dedicated to/To my...

Chapter 1

Introducción

1.1 La plataforma

Dentro de los muchos avances que internet ha hecho posible se encuentra la capacidad de que varias personas colaboren solucionando el mismo problema. Esto se ha notado especialmente en el mundo del software, donde grandes proyectos de software libre han salido adelante gracias al aporte de muchas personas de diferente nacionalidad.

Chapter 2

Conceptos básicos

2.1 Redes neuronales

Introducción a las redes neuronales, back-propagation, capas, input-outpus, etc

2.2 Convoluciones

Una imagen en el ámbito digital se entiende como una matriz de puntos. Cada uno de estos puntos puede ser interpretado como un número, que representa la localización de este punto en la escala de grises. Para representar la posición del punto en la escala de grises usaremos los valores menores para los puntos más oscuros y los mayores para los más claros.

Por lo tanto, una imagen como la siguiente:

Sería representada por esta matriz.

```
[ 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.    ]
```

Para simplificar, la mayoría de los ejemplos van a usar una escala de grises, pero son aplicables a imágenes RGB aplicando las operaciones a las tres diferentes capas al mismo tiempo.

2.2.1 Filtros

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.

Imaginemos una matriz filtro de 3×3 (también llamada matriz de convolución):

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

Se puede usar la matriz como un filtro para una imagende 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. 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 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 la operación descrita con la matriz F sobre una imagen, los píxeles más brillantes (aquellos con mayor valor) serán los que su fila superior es cero, eliminando los valores negativos y la fila intermedia es 1. Esto ocurrirá con más frecuencia en los bordes superiores de objetos claros con fondo oscuro.

Para ver la utilidad vamos a verlo aplicado a la imagen de un dígito escrito a mano, sacado del dataset MNSIT (cita req).

Si aplicamos el filtro a la imagen podemos observar como resalta en blanco los bordes superiores y en negro los inferioes. Filtros similares, rotando los valores del filtro F, son capaces de resaltar bordes lateras u oblicuos.

Lo interesante de este método es que hemos conseguido resaltar características del objeto representado en la imagen solo multiplicando matrices.

Las matrices de convolución pueden ser de mayor tamaño, permitiendo capturar características más complejas. La matrix de 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 usar un modelo de color. Uno de los más usados, RGB, se compone de tres capas, una dedicada a la intensidad roja, otra a la verde y la última a la azul, de ahí su nombre. Cada filtro se aplicaría a cada capa por separado, permitiendo de esta manera detectar diferentes características de la imagen que ocurran solo en uno de los colores.

Para ver como afectan diferentes filtros a una imagen, existe una página (http://setosa.io/ev/image-kernels/ mover a biblio) donde se pueden probar ejemplos con filtros personalizados, haciendo el concepto mucho más sencillo de comprender.

2.3 Redes neuronales convolucionales

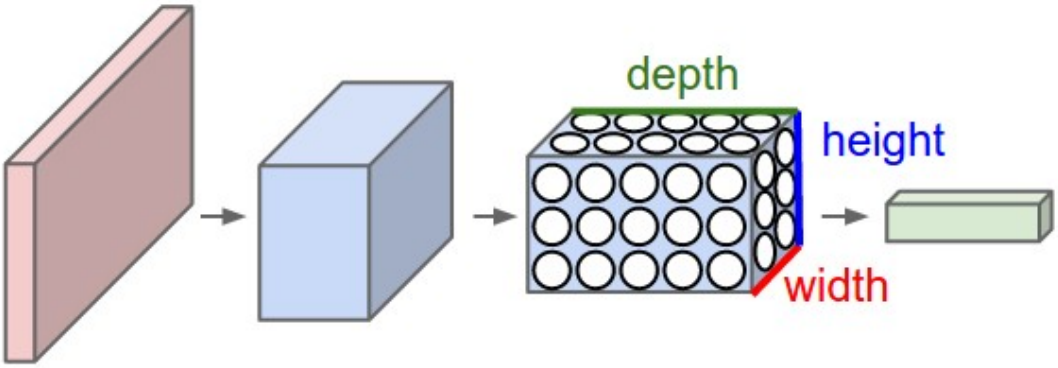
Hemos explorado la idea de que determinados filtros son capaces de extraer información localizada sobre características de la imagen. En el ejemplo del apartado anterior, dada una imagen podíamos saber si había bordes superiores y dónde se podían encontrar. Esto puede ser de gran utilidad en el campo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

El problema está en cómo encontrar los mejores filtros para sacar las características más relevantes de una imagen.

Analizando como funcionan los filtros convolucionales vemos su parecido con las redes neuronales. Al igual que las redes neuronales, los filtros son matrices que estamos aplicando a los datos de entrada que producirán unos datos de salida relevantes con la función buscada. El entrenamiento de una red neuronal va modificando los pesos de las diferentes capas hasta que produce una salida relevante con los ejemplos del conjunto de entrenamiento.

Si entendemos los pesos como la matriz convolucional, podemos hacer que sea la misma 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 imitar funciones no lineales, podemos aplicar la misma idea a las redes con filtros: componer diferentes filtros para poder extraer características más complejas.

En esta idea se basan las redes convolucionales que trabajan con imágenes. Cada capa de la red va a aplicar varios filtros a la imagen y a devolver las imágenes tratadas por dicho filtro para pasarlo 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.



La manera de representar esto es entendiendo los pesos de cada capa como una matriz de tres dimensiones. Estas tres dimensiones se pueden entender como d (profundidad) filtros de dimensión $w \times h$ (anchura y altura). Cada capa tendrá su conjunto de filtros.

2.3.1 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 aumentar la relevancia de las activaciones.

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 salidas de aplicar la imagen de entrada a estos filtros. Una imagen de tamaño 32×32 aplicada sobre una capa convolucional de 8 filtros producirá una salida de tamaño $32 \times 32 \times 8$, o lo que es lo mismo, 8 imágenes de 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.

La manera estándar de modelar 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 mucho más lentas que otra función que también evita la linealidad: $f(x) = max(0, x)$, llamada ReLU (*Rectifier Linear*). Las redes neuronales convolucionales entrenan varias veces más rápido usando ReLU (ver figura 2.1) que las equivalentes usando la función *tanh* (Krizhevsky et al., 2012)

FIGURE 2.1: Evolución de un entrenamiento para una red convolucional de cuatro capas usando ReLUs (línea sólida) vs usando tanh (línea discontínua) (Krizhevsky et al., 2012)

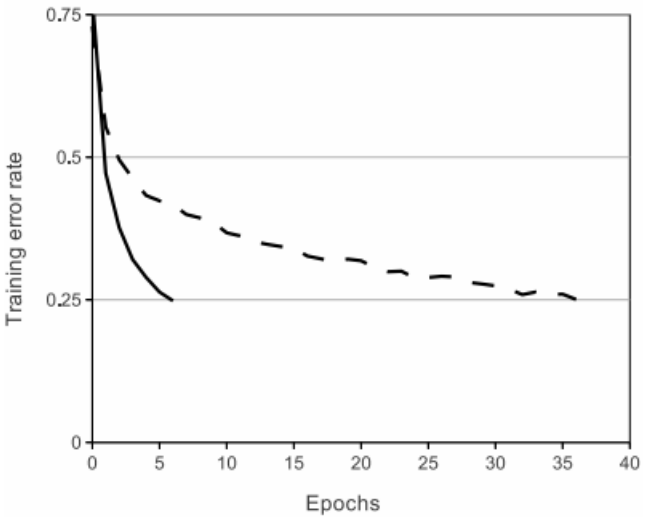
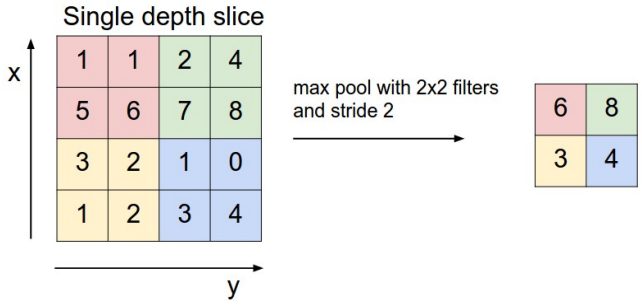


FIGURE 2.2: *Max-pooling sobre una imagen de* 4×4



Capa de muestreo - POOL *(Pooling layer)*

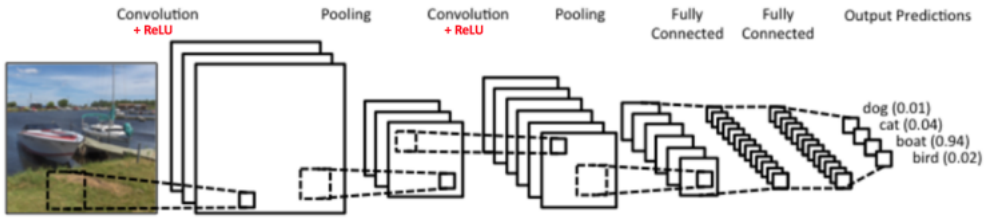
Al usar convoluciones en la imagen sobre el entorno de cada punto la información relevante de este queda difuminada y redundante. Gracias a esto se puede reducir el tamaño del problema mediante muestreo. Las capas de muestreo (*pooling layers*) resumen las salidas del entorno de cada grupo de neuronas. Un ejemplo de función de muestreo sería elegir el valor máximo de cada cuadrado de 4 píxeles (figura 2.2)

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.

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.

FIGURE 2.3: Ejemplo de arquitectura de una red convolucional para clasificación, (Russakovsky et al., 2014)



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 extraidas.

2.3.2 Arquitectura

Un esquema simplificado de un ejemplo de uso de las capas mencionadas se puede encontrar en la figura 2.3. 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 modelar 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.

Chapter 3

Definición del problema

3.1 Kaggle

Kaggle es una plataforma donde se organizan competiciones de modelos predictivos o de aprendizaje automático sobre un problema real. Diferentes organizaciones plantean un problema y ofrecen un premio para el mejor modelo que lo resuelva, haciendo que miles de participantes prueben ideas diferentes y consigan puntos de vista que, en organizaciones tradicionales, un solo equipo no tendría.

En este trabajo se ha optado por la participación en uno de esos problemas, The Nature Conservancy Fisheries Monitoring.

3.2 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 hecho de que se pueda analizar este tipo de imágenes de una manera rápida y automática permitirá asignar recursos de una manera mucho más efectiva para el control de este tipo de actividades. (Cita: fisheries monitoring)

3.3 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 mostrar la cubierta de un barco donde puede aparecer un pez. En base al pez que aparezca hay que clasificarlo en una de las siguientes seis categorías:

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 pero no perteneciente a ninguna de las categorías mencionadas, la categoría será OTHER.

$$categories = [ALB, BET, DOL, LAG, SHARK, YFT, OTHER, NOF]$$

3.3.1 Datos

El descargable que se provee en el reto consta de tres elementos:

1. Conjunto de datos de entrenamiento: 3777 imágenes etiquetadas con una de las ocho categorías existentes.

Table 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_00007.jpg	0.455	0.052	0.030	0.0173	0.123	0.079	0.046	0.194
img_00009.jpg	0.455	0.052	0.030	0.0173	0.123	0.079	0.046	0.194

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

3.3.2 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 caiga en cada una de las ocho categorías diferentes. En la tabla 3.1 se muestra un ejemplo de las primeras filas del archivo CSV a enviar.

El envío se evalúa usando multi-class logarithmic loss. Cada imagen ha sido etiquetada con una clase, pero es necesario enviar las probabilidades de cada clase para cada imagen. La formula final es:

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

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

Leaderboard y fases de la competición

Existe una tabla donde se publican los resultados de los envíos de cada participante. Estos resultados están calculados solo con un subconjunto del conjunto de test, de esta manera se evita que los participantes puedan aprovechar un sobreajuste para subir en la clasificación.

La puntuación final vendrá determinada por la puntuación de todo el conjunto de test.

Esta manera de evaluar es algo bastante común en kaggle, sin embargo los organizadores de este reto en particular han decidido hacer un pequeño cambio a la hora de evaluar la puntuación.

3.3.3 Doble fase de evaluación y envío

La evaluación en este reto contará de dos partes. La primera, donde se entrenará el modelo, requiere (como ya se ha dicho) evaluar el conjunto de test de mil imágenes. La fecha límite para la entrega de modelos de esta primera fase es siete días antes de

la finalización del reto. Será necesario enviar el CSV con las predicciones y el modelo usado.

Una vez finalice la primera fase, un segundo conjunto de test de muchas más imágenes será hecho público. Este conjunto de test deberá ser evaluado con el modelo generado en la fase anterior, siendo razón de descalficación modificar el modelo. Todos los participantes que no envíen la predicción de la segunda fase serán eliminados del reto.

3.4 El conjunto de datos

Aqui hago un pequeño análisis del conjunto de datos, la distribución de imágenes por clases, por estructura (tamaño de imaggen), color, etc

Chapter 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 3.3.2. 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 segundo conjunto de test tiene más de 13000 imágenes, que no son rápidas de cargar en memoria ni de pasar por el modelo que se genere. 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 realidado dejando un **60%** de los datos al conjunto de entrenamiento, un **20%** al conjunto de validación y un **20%** al conjunto de test.

4.1.1 Partición de conjuntos de datos

La partición del conjunto de datos es una operación delicada. Ya vimos que uno de los problemas del dataset era que la cantidad de imágenes para cada clase era muy variada, teniendo algunas clases un número muy bajo de ejemplos. Si sacamos el 40% de las imágenes del dataset, podemos estar vaciando una o varias categorías de ejemplos de entrenamiento, haciendo inútil cualquier clasificación de esas clases.

La solución por la que se ha optado es generar un subconjunto del 20% usando el 20% de los ejemplos de cada clase, evitando así vaciar alguna de las clases.

4.1.2 Evaluación del modelo

Para una búsqueda más óptima de parámetros y configuraciones el modelo se evaluará contra el grupo de test generado. Una vez se encuentren determinados modelos

especialmente interesantes por tener una puntuación máxima local entre aquellos a los que se compara habrá que enviarlo a Kaggle para su evaluación completa.

Enviar el modelo a Kaggle se puede hacer sobre una partición más grande del conjunto de entrenamiento, ya que no es necesario usar un conjunto de test para evaluar, solo uno de validación. Esto permite entrenar el modelo sobre el 80% de los datos consiguiendo, por lo general, resultados más precisos.

4.1.3 Software

Todos los modelos de redes convolucionales que se van a entrenar en este proyecto han sido entrenados usando Keras

4.2 Idea

A la hora de afrontar este problema de clasificación de peces es lógico seguir una estrategia separada en dos pasos: primero buscar si existe un pez en la foto y luego intentar clasificarlo en una de las categorías existentes.

Para encontrar un pez en la foto es necesario encontrar una serie de características que puedan ser identificadas con algún pez. La idea de la solución parte de esta base. A la hora de clasificar una imagen primero es necesario encontrar el contenido relevante para ser usado en la clasificación.

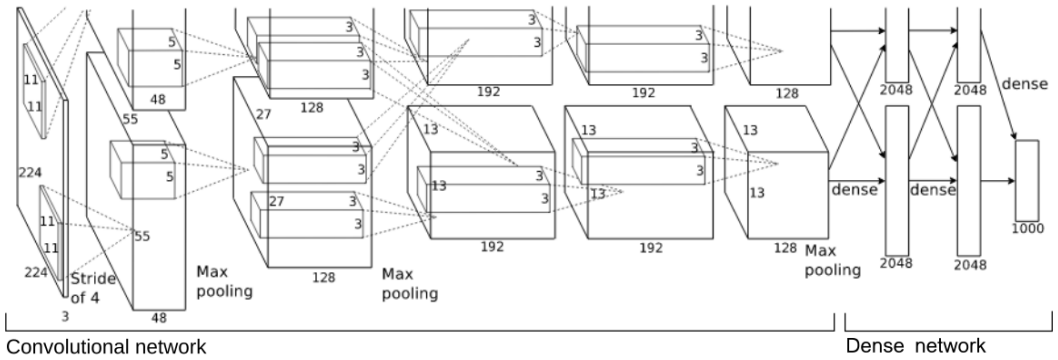
Como ya se comentó al hablar de las redes convolucionales (capítulo 2.3.2), las arquitecturas encontradas en problemas similares (Krizhevsky et al., 2012) permiten separar con claridad estas dos etapas mediante el uso de redes convolucionales (CONV) y capas densas (FC).

4.3 Arquitectura

La arquitectura general usada, que luego sufrirá pequeños cambios, es la descrita en la figura 4.1 (Krizhevsky et al.)

Esta arquitectura usa en su primera parte una red convolucional preentrenada sobre un conjunto de imágenes mucho más generalista, en este caso Imagenet. Al usar una red convolucional preentrenada sobre fotografías tendrá muchas más capacidad para distinguir características de diferentes objetos, además que existen varias categorías de peces en Imagenet, por lo que sabrá diferenciar este tipo de fotografías.

FIGURE 4.1: Arquitectura de la red en dos partes



4.3.1 Red convolucional

Como explicamos en el capítulo 2.3, cuando un modelo convolucional recibe una imagen va a devolver N matrices bidimensionales representando el resultado de las aplicación de los N conjuntos de filtros a la imagen inicial. Al ser N matrices bidimensionales también puede considerarse una matriz tridimensional.

Intuitivamente, y haciendo una simplificación, podemos pensar que cada uno de estos filtros representa un mapa de calor de la aparición en la imagen de diferentes características. Por ejemplo, ¿cúanto se parece cada parte de esta imagen a la piel de un pez?.

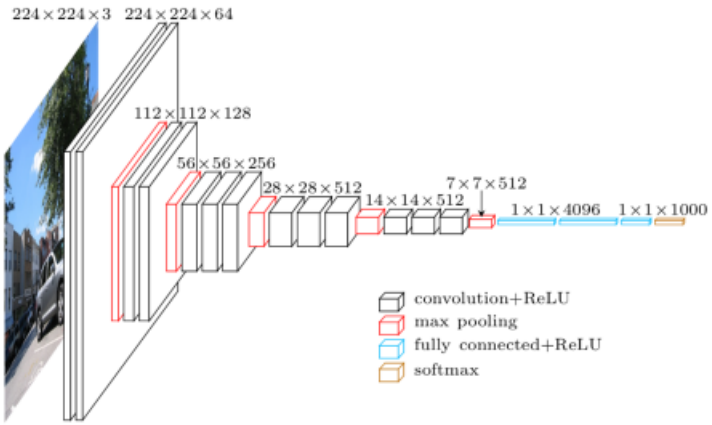
Hay que tener en cuenta que este modelo convolucional no ha sido entrenado con el conjunto de entrenamiento, si no con el conjunto de entrenamiento de Imagenet. Para lo único que vamos a usar esta red convolucional es para transformar estas imágenes a

4.3.2 Modelo preentrenado

La arquitectura del artículo original (Krizhevsky et al., 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. Un buen punto de partida puede ser mirar los modelos ganadores de años posteriores.

El modelo principal a usar es VGG, desarrollado por el *Visual Geometry Group*, de la Universidad de Oxford. Es un modelo especialmente interesante por su simplicidad, aparte de obtener una de las mejores puntuaciones en ILSVRC 2014.

FIGURE 4.2: Arquitectura de VGG



4.4 VGG16

Una de las principales características de VGG es la 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 and 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 ya que es la cantidad de capas CONV y FC que posee.

Estamos usando la configuración VGG16 de todas las descritas en su definición original ya que, junto con VGG19, consigue los mejores resultados.

4.4.1 *Fine-tuning*

Si observamos la última capa del modelo VGG16, vemos que la salida tiene 1000 elementos. Tiene esta forma ya que ILSVRC consistía en clasificar una imagen entre mil categorías diferentes. Si el problema a resolver consiste en clasificar entre ocho categorías, es lógico modificar esta última capa para que tenga solo 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 tenga esta separación lógica entre la red convolucional y la red densa (las tres capas FC) hace que se pueda separar el modelo en dos modelos diferentes: uno convolucional, que no habrá cambiado con la adaptación a las ocho categorías, y otro denso, que tendrá que ser reentrenado de nuevo.

Al partir una red en dos partes diferentes hay que tener en cuenta que la segunda parte, la red densa, no recibe como entrada las imágenes, si no la salida de la red convolucional, con todas las transformaciones que esta produce. Es necesario entonces

aplicar la red convolucional a todo el conjunto de datos para crear un nuevo conjunto de datos con el que reentrenar la red densa.

Esta técnica de ajustar los pafametros de un modelo ya conocido para adaptarlo con nuevo conjunto de datos se conoce como *fine-tuning*.

Usando el conjunto de datos definido en el capítulo 4.1, con la separación 60%, 20% y 20% para los conjuntos de entrenamiento, validación y test se ha entrenado la red densa.

Primero es necesaria la transformación del conjunto en las features producidas por la red convolucional.

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

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

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

# Convertir dataset a features mediante la red convolucional
train_feat = conv_model.predict(train)
valid_feat = conv_model.predict(valid)
test_feat = conv_model.predict(test)
```

Este código no es completamente válido, ya que se han tomado algunas libertades para mejorar la comprensión y lectura del mismo. Define una estructura general a seguir para este problema (carga de datos, transformación, *fine-tuning* y evaluación). Para una mayor comprensión el la mayor parte de las librerías usadas así como las utilidades están descritas en el Apéndice 1 (cuando lo pase a LaTex)

Una vez cargadas las librerías necesarias (Vgg16 es una clase abstracta que importa las bibliotecas necesarias de Keras), ambos modelos y transformado el dataset mediante la red convolucional se puede pasar a reentrenar la última parte de la red. Primero hay que cambiar el final de la red para conseguir que tenga ocho salidas y luego entrenar.

```
# Cambiar el final de la red densa de 1000 outputs a 8
dense_model.pop()  # Elimina la  ltima  capa
dense_model.add(
    Dense(8, activation='softmax')  # Introducir una nueva
)

# Compilar el modelo
dense_model.compile(
```

```
    SGD(lr=0.01),
    loss='categorical_crossentropy',
    metrics=['accuracy'],  # Muestra la precision del modelo
)

# Entrenar 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, accurtcy
>>> [2.38985158622892523, 0.66099843993759748]
```

La puntuación total de kaggle para este conjunto de test sería 2.389. Esto es sobre un conjunto de test muy pequeño, solo el 20% del conjunto de entrenamiento. Como se ya ha dicho anteriormente habría que mandar las predicciones del conjunto de test final a kaggle para verificar la puntuación del modelo, pero por problemas de restricciones esta es la forma de medir los modelos generados.

El segundo número que devuelve la función *evaluate* es la precisón (*accuracy*) del modelo. Al ser el modelo un modelo de clasificación, el mandar "*accuracy*" como parámetro a la hora de compilar el modelo hace que eliga como métrica la precisión por categorías. Esto va a devolver el porcentaje de veces que la clase con la probabilidad máxima se corresponde con la clase etiquetada en el dataset. En este caso el modelo ha adivinado 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.

4.5 Modelo personalizado

La idea original que se comentaba al principio consistía en separar el problema en el modelo convolucional y el modelo de clasificación. Es en el segundo donde se puede conseguir toda la flexibilidad.

El modelo original de VGG usa dos capas *densas* de 4096 neuronas para clasificar entre mil clases diferentes. Ya que este 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. Para no tener que estar modificando el modelo original cada vez que haga falta es mucho más sencillo crear un modelo nuevo con *Keras* y añadir todas las capas necesarias.

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

dense_model = keras.models.Sequential(build_class_layers())
```

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 ($ancho \times alto \times filtros$), mientras que la 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.

```
# Entrenar la red
dense_model.compile(...)
dense_model.fit(...)
# Evaluar el modelo sobre el conjunto de test
dense_model.evaluate(test_feat, test_labels)
>>> [1.2772131643007987, 0.6956786271450857]
```

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 el primero, 3.9 segundos el último). Esto no solo ha permitido entrenar la red durante más tiempo, si no que en el futuro hará posible entrenar sobre mayores cantidades de datos sin sacrificar velocidad.

4.5.1 Mejorar el modelo

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.

```
Epoch 5/7
loss: 1.406 - acc: 0.621 - val_loss: 2.0197 - val_acc: 1.861
Epoch 6/7
loss: 0.992 - acc: 0.674 - val_loss: 1.4356 - val_acc: 1.001
Epoch 7/7
loss: 0.872 - acc: 0.761 - val_loss: 1.2772 - val_acc: 0.695
```

Los valores mostrados indican los resultados de la evaluación del modelo sobre el conjunto de entrenamiento y el de validación, respectivamente. Esto es mostrado para cada uno de los pasos. Se puede observar que el modelo funciona mucho mejor en el conjunto de entrenamiento que en el de evaluación.

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.

4.5.2 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. Esta capa convierte activaciones aleatorias a 0, eliminando la información transportada.

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 (Krizhevsky et al., 2012).

Un pequeño experimento en un modelo más avanzado (VGG con *batch normalization* y aumento de datos) permite ver como afecta el *dropout* a la puntuación final. En la figura 4.3 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.

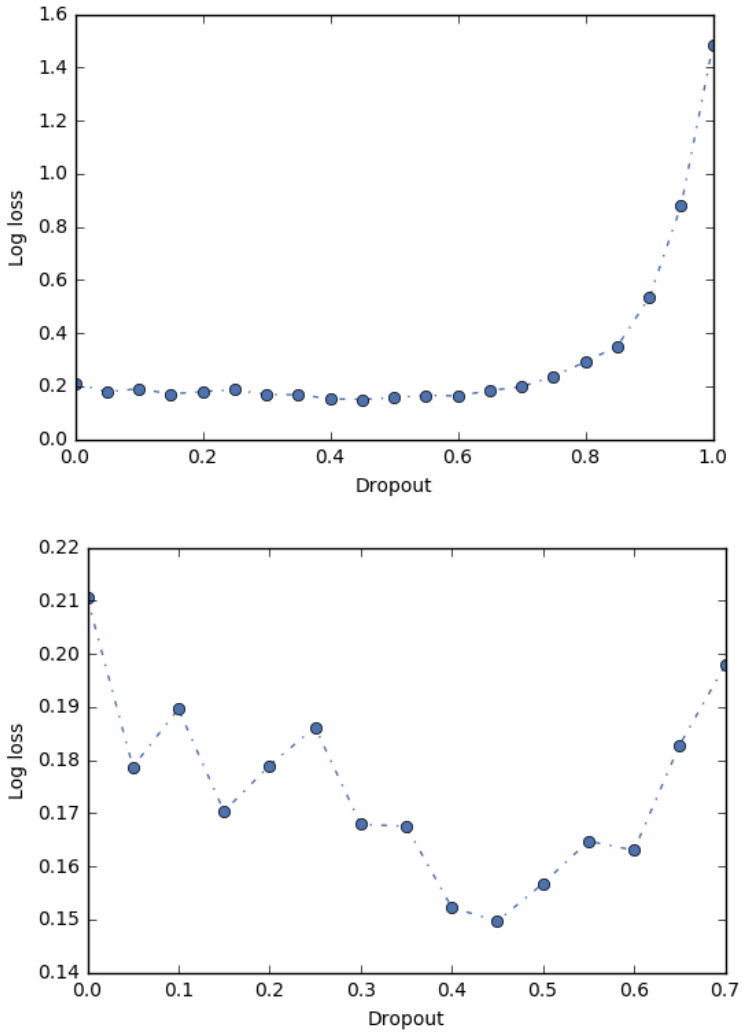
También se ve que el modelo empieza a perder eficacia a partir del 70% de *dropout*, empeorando el modelo original. Esto significa que el modelo es capaz de generalizar con información útil incluso cuando solo posee el 30% de las activaciones.

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. La idea aplicada aquí es distribuir el *dropout* disminuyéndolo en la entrada y la salida y haciendo que tenga su punto máximo en el centro de la red.

La definición de la red anterior quedaría de la siguiente manera:

```
def build_dense_layers(p):
    return [
        Flatten(),
        Dropout(p/4)
        Dense(512, activation='relu'),
        Dropout(p)
        Dense(512, activation='relu'),
        Dropout(p/2)
        Dense(8, activation='softmax')
    ]
dense_model = keras.models.Sequential(build_class_layers(0.45))
```

FIGURE 4.3: Evolución de la puntuación de un modelo usando *dropout*



Los resultados obtenidos con esta configuracion no mejoran mejoran los resultados de la red anterior, pero como se ve en la figura 4.3 sí que lo hará a posteriori, cuando se le apliquen otro tipo de mejoras al modelo.

4.5.3 Batch normalization

4.5.4 Data augmentation

En muchos algoritmos de

4.5.5 Fully convolutional network

Bibliography

Krizhevsky, Alex et al. (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Russakovsky, Olga et al. (2014). "ImageNet Large Scale Visual Recognition Challenge". In: *CoRR* abs/1409.0575. URL: http://arxiv.org/abs/1409.0575.

Simonyan, Karen and Andrew Zisserman (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556. URL: http://arxiv.org/abs/1409.1556.