



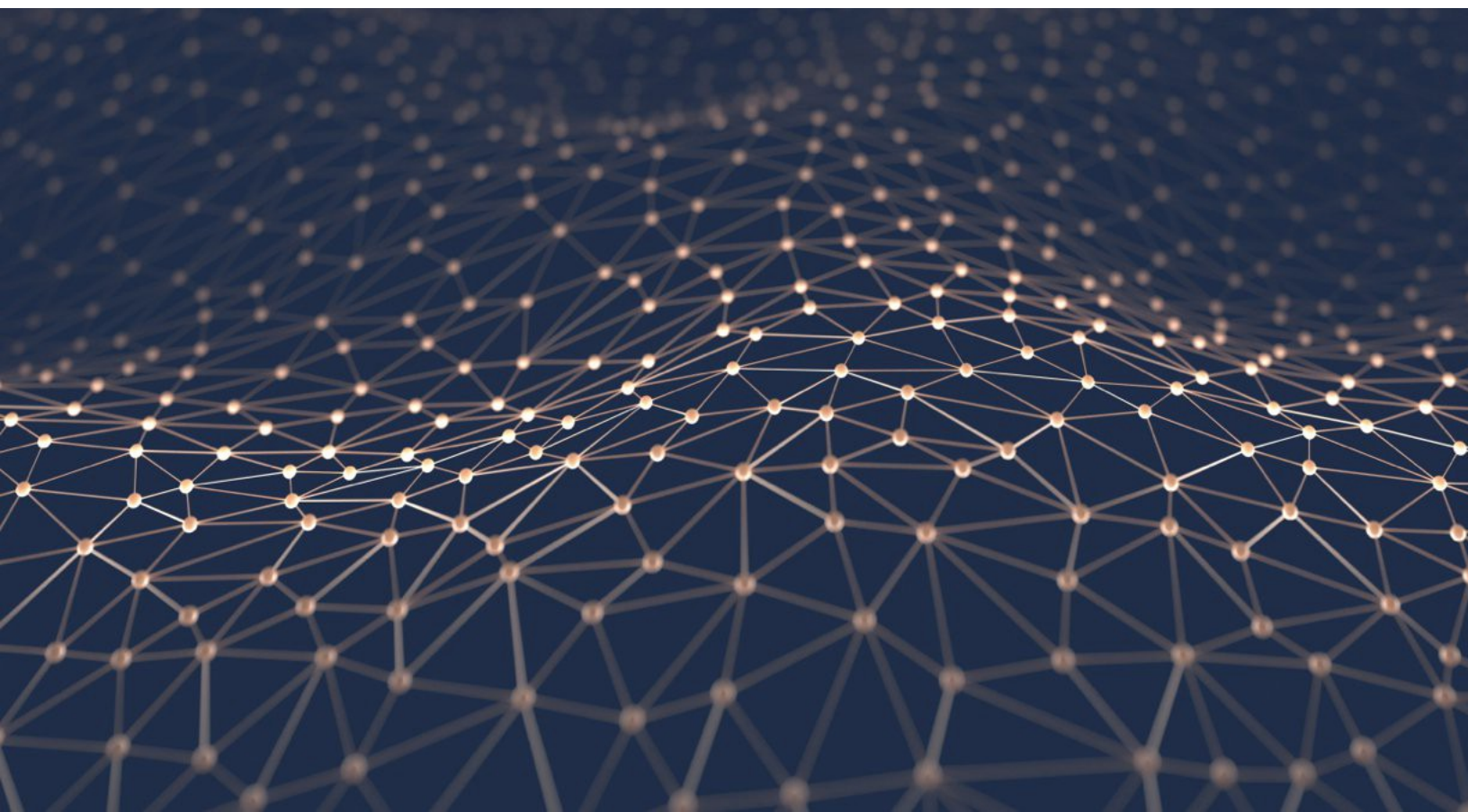
UNIVERSIDAD  
DE GRANADA



# Deep Learning

PetFinder.my Adoption Prediction

Gema Correa Fernández    Andrea Morales Garzón



SISTEMAS INTELIGENTES PARA LA GESTIÓN DE LA EMPRESA, CURSO 2018-2019

MÁSTER PROFESIONAL EN INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE GRANADA

[gecorrea@correo.ugr.es](mailto:gecorrea@correo.ugr.es) y [andreamgmg@correo.ugr.es](mailto:andreamgmg@correo.ugr.es)

*9 de junio de 2019*

## Índice de figuras

1.1	Imágenes del dataset . . . . .	7
3.1	Ejemplo de red convolucional . . . . .	16
3.2	Proceso de convolución . . . . .	20
3.3	Las capas de agrupamiento se utilizan para aplicar la reducción de muestreo no lineal en los mapas de activación . . . . .	20
3.4	Capa Dropout para técnica de regularización . . . . .	21
3.5	Capa Flatten [Staa] . . . . .	22
4.1	Gráficas para el modelo I con 3 épocas . . . . .	27
4.2	Gráficas para el modelo I con 8 épocas . . . . .	27
4.3	Funcionamiento de Data augmentation I . . . . .	29
4.4	Funcionamiento de Data augmentation III . . . . .	29
4.5	Funcionamiento de Data augmentation II . . . . .	31
4.6	Rotación de una imagen del conjunto de entrenamiento . . . . .	31
4.7	Data Augmentation en otro tipo de objetos . . . . .	32
4.8	Gráficas para el modelo III con 5 épocas . . . . .	32
4.9	Gráficas para el modelo III con 10 épocas . . . . .	33
4.10	Funcionamiento transfer para Transfer learning . . . . .	34
4.11	Arquitectura de Inception V3 . . . . .	34
4.12	Arquitectura de VGG16 . . . . .	35
4.13	Gráficas para el resultado con VGG16 . . . . .	38
4.14	Mejor modelo (pero entrenado sin dropout) . . . . .	40





# Índice general

<b>1</b>	<b>Introducción</b>	<b>7</b>
1.1	Descripción de la práctica	7
1.1.1	Estructura de la práctica	8
<b>2</b>	<b>Herramientas</b>	<b>11</b>
2.1	Descripción de las herramientas	11
2.1.1	Consideraciones a tener en cuenta	12
<b>3</b>	<b>Implementación</b>	<b>13</b>
3.1	Conjunto de imágenes de <i>PetFinder.my</i>	13
3.1.1	Cargar el conjunto de datos	14
3.2	Redes convolucionales	16
3.3	Entrenamiento de la red	17
3.3.1	Variables	17
3.3.2	Tareas	18
3.3.3	Modelo Secuencial de Keras	18
3.3.4	Configuración de los modelos	19
3.4	Compilar y probar el modelo	22
3.5	Evaluación del conjunto de prueba	23
<b>4</b>	<b>Análisis</b>	<b>25</b>
4.1	Análisis del problema	25

---

<b>4.2</b>	<b>Evaluación y métricas para los modelos</b>	<b>25</b>
<b>4.3</b>	<b>Configuración de los modelos</b>	<b>26</b>
4.3.1	Modelo simple . . . . .	26
4.3.2	Modelo simple + Data augmentation . . . . .	29
4.3.3	Transfer learning . . . . .	33
4.3.4	Transfer learning + Fine tuning . . . . .	36
<b>4.4</b>	<b>Resumen de los modelos</b>	<b>39</b>
<b>5</b>	<b>Conclusiones . . . . .</b>	<b>41</b>
<b>5.1</b>	<b>Conclusiones</b>	<b>41</b>
<b>6</b>	<b>Referencias . . . . .</b>	<b>45</b>

# 1. Introducción

## 1.1 Descripción de la práctica

En esta segunda práctica, se va a estudiar cómo crear un **modelo de clasificación de imágenes basado en redes neuronales profundas**. Para ello, la práctica consistirá en la resolución de un problema de aprendizaje utilizando técnicas de *Deep Learning*.

Se va a trabajar sobre el conjunto de imágenes de mascotas del dataset de *Kaggle PetFinder.my* (<https://www.kaggle.com/c/petfinder-adoption-prediction>). El problema consiste en predecir, a partir de datos del animal, el tiempo de adopción de una mascota, en donde el tiempo se encuentra representado por un valor categórico 0,1,2,3,4. En este caso, vamos hacer uso sólo de las imágenes proporcionadas.

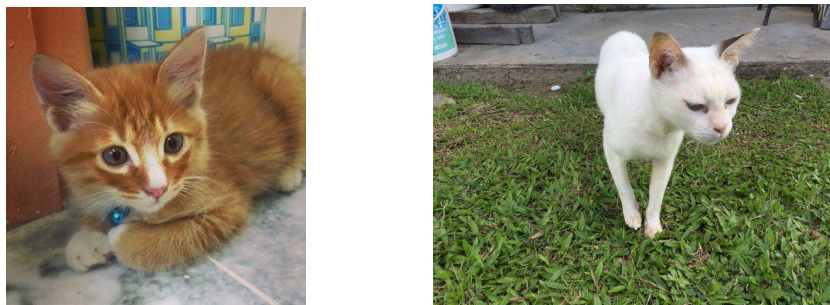


Figura 1.1: Imágenes del dataset

Las variables categóricas que podemos encontrarnos están determinadas en un rango entre 0 y 4, en donde el **0** hace referencia al **mínimo tiempo** que pasa la mascota desde que se recoge hasta que se adopta (*imagen de la izquierda*) y el **4**, hace referencia al **máximo tiempo** que pasa la mascota desde que se recoge hasta que se adopta (*imagen de la derecha*). Se debe de tener en cuenta que la asignación de estas etiquetas conlleva una variante subjetiva, lo cual dificulta la obtención de un modelo, ya que la línea que separa una etiqueta de otra es muy difusa.

Por ejemplo, desde nuestro punto de vista, si nosotros nos centramos en las imágenes anteriores observamos cómo una imagen nos transmite más afecto que otra o nos produce otro tipo de sentimientos. En concreto, estamos haciendo referencia a la imagen de la izquierda, la cual nos gusta más en caso de que fuéramos a adoptar. No obstante, todo esto depende del punto de vista de cada persona, ya que alguien puede sentir más afecto hacia la imagen de la derecha al poder pensar que ese gatito ha sufrido más. En términos generales, con este ejemplo queremos demostrar, que tanto la perspectiva humana, como la opinión o sentimientos que se transmiten en este tipo de casos, influye mucho en el etiquetado realizado en el problema.

Como se ha podido comprobar, este tipo de problemas son difíciles de clasificar. En esta memoria se explicarán las tareas llevadas a cabo para construir un modelo de multi-clasificación, cuyo objetivo sea la obtención de un modelo adecuado, así como el equilibrio entre la calidad del clasificador y el tiempo requerido para su entrenamiento. En dicho problema se pueden usar algunas de las siguientes tareas (*en negrita los que hemos usado nosotros*):

- **Ajuste de la topología de la red**
- **Ajuste de hiperparámetros**
- Ajuste de la función de coste
- **Ajuste de algoritmo de optimización**
- *Data augmentation*
- *Transfer learning*
- *Fine tuning*
- Uso de datos adicionales


A lo largo de la memoria, se explicarán las distintas aproximaciones se han utilizado, así como el impacto de éstas en los resultados de clasificación. También se detallará el proceso de generación de los conjuntos de entrenamiento, validación y test, y el razonamiento llevado a cabo para la decisión de los hiperparámetros en el transcurso de la práctica.

### 1.1.1 Estructura de la práctica

En este primer apartado, se ha explicado el problema que vamos a abarcar a lo largo de la práctica. A continuación, se detallan de forma resumida el contenidos del resto de apartados del documento:

- En el *apartado 2*, se describen las herramientas usadas para la resolución del problema de clasificación explicado previamente.
- En el *apartado 3*, se explica la implementación usada con R + Keras para la resolución del problema de predecir el tiempo de adopción de una mascota. Además, de los conceptos necesarios para comprender la creación de los modelos en el siguiente apartado.
- En el *apartado 4*, se realiza un análisis de los distintos modelos o implementaciones de redes convolucionales, hasta la obtención del mínimo error sobre el conjunto de prueba.
- En el *apartado 5*, se desarrolla una conclusión de los resultados obtenidos para la práctica.



-  Para la realización de la práctica se usarán de manera complementaria referencias de Internet, la documentación proporcionado por Keras (<https://keras.rstudio.com>), como el libro de *Deep Learning with R* (<https://www.manning.com/books/deep-learning-with-r>).



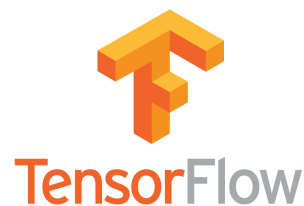
## 2. Herramientas

### 2.1 Descripción de las herramientas

Para la implementación de redes neuronales se pueden encontrar en Internet múltiples bibliotecas que implementan distintos algoritmos. Dichas implementaciones están disponibles en múltiples lenguajes de programación.

En nuestro caso y por afinidad, se va hacer uso de R, y al ser uno de los principales lenguajes de programación usados en el campo de la Inteligencia Artificial. Dentro de este lenguaje, existen varios frameworks que nos facilitan el trabajar con modelos de *Deep Learning*, entre ellos destacamos:

- TensorFlow: es una biblioteca de software libre que se utiliza para realizar cálculos numéricos mediante diagramas de flujo de datos. En su origen, *TensorFlow* fue fruto del trabajo de investigadores e ingenieros de *Google Brain Team* que formaban parte de la organización de investigación del aprendizaje automático de Google. Su objetivo era realizar investigaciones en el campo del aprendizaje automático y las redes neuronales profundas. A pesar de que este era su propósito inicial, se trata de un sistema lo bastante general como para poder aplicarse en muchos campos [Ten].



- Keras: es una API de redes neuronales de alto nivel, escrita en Python y capaz de funcionar sobre TensorFlow, además de Microsoft Cognitive Toolkit o Theano. Fue desarrollada con el objetivo de permitir una rápida experimentación con redes neuronales profundas, ya que su objetivo era poder pasar de la idea al resultado en el menor tiempo posible [Kerc]. Se enfoca

en ser fácil de usar, modular y extensible, por lo que es más fácil de usar en comparación que Tensorflow y se ejecuta sin problemas en la CPU y la GPU [Medb]. En nuestro caso, utilizaremos dicha librería desde R.



### 2.1.1 Consideraciones a tener en cuenta

Es aconsejable usar un ordenador con GPU en lugar de CPU, ya que la complejidad computacional que conlleva el entrenamiento de redes neuronales es elevado. Esta afirmación se obtiene al comparar la ejecución de un código en CPU con GPU, así como las indicaciones del profesor. Sin embargo, al no disponer de GPU en nuestros ordenadores, se ha optado por ejecutar en el *cloud*.

Por tanto, la ejecución de las redes neuronales se ha realizado en una máquina virtual con GPU y RStudio desplegada en *Google Cloud Platform*. Los pasos para la creación y puesta en marcha de dicha máquina virtual, han sido los comentados por el profesor en la asignatura.



## 3. Implementación

Una vez escogido el lenguaje de programación R y el framework Keras para la implementación del algoritmo, pasamos a la elección de los modelos concretos de redes neuronales artificiales. Dentro de los distintos tipos de redes neuronales existentes, y aprovechando que nuestro conjunto de datos está formado por imágenes, se va hacer uso de las **redes neuronales convolutivas**. Debido a experiencias previas con modelos predictivos a partir de imágenes, conocemos la potencia de la convolución a la hora de extraer características de una imagen, por lo que, como punto de origen de nuestros experimentos, partiremos directamente de este tipo de redes en nuestra configuración inicial.

### 3.1 Conjunto de imágenes de *PetFinder.my*

Para esta práctica, se va a utilizar el conjunto imágenes de las mascotas previamente mencionado. En esta base de datos se nos proporciona dos conjuntos de datos:

1. **Imágenes para el conjunto de *training*:** formado por 58311 imágenes, cada una de ellas representa una mascota. Es importante destacar que para una misma mascota aparecerán alrededor de 4 imágenes diferentes. Este conjunto será el que utilicemos para entrenar y evaluar las distintas redes neuronales que se configuren. Dentro de dicha carpeta de imágenes, se dispone de subcarpetas, cada una clasificada con las imágenes pertenecientes a cada clase.
  - Clase 0: 1363 imágenes
  - Clase 1: 11517 imágenes
  - Clase 2: 16438 imágenes
  - Clase 3: 15059 imágenes
  - Clase 4: 13934 imágenes
2. **Imágenes para el conjunto de *test*:** formado por 14465 imágenes. Como ya hemos visto, evaluar la calidad de un clasificador utilizando los mismos datos con los que entrenamos

no es representativo, y no nos dará resultados válidos que podamos extender o generalizar más allá del conjunto con el que hemos entrenado, ya que hay que tener en cuenta que hay posibles problemas con los que nos podemos encontrar, como el sobreaprendizaje. Por ello, se utiliza un segundo conjunto de aprendizaje, sobre el que probaremos la efectividad de la red que hemos entrenado previamente. Sin embargo, no se va hacer uso de este conjunto, al no disponer de la variable correspondiente a las etiquetas.

Además, debido a la complejidad del problema se va hacer uso de un tercer conjunto de datos, llamado **validación**, el cual es obtenido a partir de una parte del conjunto de training, con el fin de validar el problema. Para ello, la siguiente función nos permite realizar dicha separación a partir de las imágenes del train, ya que en la carpeta para las imágenes del test, no tenemos asociadas las imágenes a la clase que pertenecen.

```

1  # Se quita un trozo de train y se mete en test
2  separaTrainTest <- function(carpeta_train, carpeta_test, porcentaje = 0.2) {
3
4      clases<-list.dirs(path = carpeta_train, full.names = FALSE)
5
6      for (clase in clases){
7          if(clase != "") {
8              carpeta_clase_train <- paste(carpeta_train,clase,sep = "/")
9              carpeta_clase_test <- paste(carpeta_test,clase,sep = "/")
10
11              todos <- list.files(path = carpeta_clase_train)
12              a_copiar <- sample(todos, length(todos)*porcentaje)
13
14              for (fichero in a_copiar){
15                  file.copy(paste(carpeta_clase_train, fichero, sep = "/"),
16                           carpeta_clase_test)
17                  file.remove(paste(carpeta_clase_train, fichero, sep = "/"))
18              }
19          }
20      }
21  }
```

Por tanto, a partir de las imágenes del training, se obtendrá un conjunto para test y validación. Para ello, ha sido necesario crearse los directorios `my_test/` y `validation_images/`, y dentro 5 subcarpetas, cada una perteneciente a una clase.

- Número de imágenes para train: 39656
- Número de imágenes para test: 11659
- Número de imágenes para validación: 6996

### 3.1.1 Cargar el conjunto de datos

Como era de esperar, en el conjunto de imágenes proporcionado se distinguen distintas representaciones, es decir, el tamaño de las imágenes no es fijo ya que varía. Pero previamente a la carga de las imágenes es necesario establecer los directorios donde vamos a trabajar:

```

1  dataset_dir <- './petfinder-adoption-data/'
2  train_images_dir <- paste0(dataset_dir, 'train_images/')
3  train_data_file <- paste0(dataset_dir, 'train.csv/')
4  test_images_dir <- paste0(dataset_dir, 'my_test/')

```

```

5 test_data_file <- paste0(dataset_dir, 'test.csv/')
6 val_images_dir <- paste0(dataset_dir, 'validation_images/')

```

Una vez establecida la ubicación de las carpetas, sólo debemos pasar a cargar los ficheros comentados anteriormente. Para ello, primeramente, deberemos realizar un pequeño preprocesamiento a los datos, con el fin de poder trabajar en todo el problema, con los datos redimensionados a la misma escala.

```

1 train_datagen <- image_data_generator(rescale = 1/255)
2 validation_datagen <- image_data_generator(rescale = 1/255)
3 test_datagen <- image_data_generator(rescale = 1/255)

```

Seguidamente, haciendo uso de la función `flow_images_from_directory()`, generamos lotes<sup>1</sup> de datos a partir de imágenes en un directorio con las imágenes reescaladas. En dicha imagen, se llevará a cabo un escalado tanto a los datos training, test y validación, para así ayudar a acelerar la formación y reducir la posibilidad de quedarse atascado en un óptima local. Esta implementación, sirve para reducir píxeles sin perder excesiva información de nuestras imágenes. Pongamos un ejemplo, si nosotros tenemos una fotografía de una habitación con una perfecta resolución y queremos identificar un ordenador, en dicha foto nos resultará más difícil detectar un ordenador, ya que hay demasiada información. Al hacer un escalado, se transformará, manteniendo prácticamente la misma información, y, siendo capaz de detectar igualmente el objeto.

En nuestro caso, hemos decidido escalar las imágenes a un tamaño (244,244), tal y como se justificará en secciones posteriores. La decisión para elegir dicho tamaño, viene en base de las siguientes razones:

- En este área de trabajo es costumbre utilizar en el ajuste de hiperparámetros aquellos tamaños que han funcionado bien en problemas estudiados previamente. Por ello, sobretodo en nuestro caso, que no tenemos excesivos conocimientos de la materia, decididos coger de partida el tamaño de imagen utilizado por algunos modelos de Deep Learning que, al igual que en nuestro caso, hacen uso de imágenes de *Imagenet*.
- En muchos de los ejemplos, hemos trabajado con modelos y capas pre-entrenadas con dicho ajuste de tamaño para las imágenes, por lo que, para mantener una cierta coherencia, se optó por redimensionar a la misma altura y anchura.

```

1 # Conjunto de train
2 train_data <- flow_images_from_directory(
3   directory = train_images_dir,
4   generator = train_datagen,
5   target_size = c(224, 224),      # (w, h) --> (224, 224)
6   batch_size = 100,              # grupos de 100 imagenes
7   class_mode = "categorical"     # etiquetas categoricas
8 )
9
10 # Conjunto de test
11 test_data <- flow_images_from_directory(
12   directory = test_images_dir,
13   generator = test_datagen,
14   target_size = c(224, 224),      # (w, h) --> (224, 224)
15   batch_size = 20,               # grupos de 20 imagenes

```

<sup>1</sup>En secciones posteriores entraremos en detalle sobre este tema.

```

16 class_mode = "categorical"      # etiquetas categoricas
17 )
18
19 # Conjunto de validacion
20 validation_data <- flow_images_from_directory(
21   directory = val_images_dir,
22   generator = validation_datagen,
23   target_size = c(224, 224),    # (w, h) --> (150, 150)
24   batch_size = 100,             # grupos de 20 imagenes
25   class_mode = "categorical"    # etiquetas categoricas
26 )

```

## 3.2 Redes convolucionales

Las **redes convolucionales**, también conocidas como *convnets*, se caracterizan por permitir, gracias a las características proporcionadas por la operación de convolución, aprender patrones locales dentro de las imágenes, en lugar de intentar aprender de ellas como un todo.

La operación de convolución parte de la idea de que dado un píxel en una imagen, los píxeles relevantes para este son los que están a su alrededor, porque pueden estar relacionados con el valor de este mismo píxel. Por ejemplo, si se tiene una imagen con varios objetos, dado un píxel relacionado con un objeto de los representados, lo más normal sería que los píxeles que se encuentran a su alrededor también estén representando ese objeto, o su frontera, o su sombra, etc. De igual forma, dos píxeles de esquinas distintas de una misma imagen probablemente no estén relacionados porque cada uno esté representando una parte distinta de la escena. Si fuese global, estaríamos teniendo en cuenta la totalidad de píxeles, lo cuál puede no ser interesante en vistas a obtener resultados coherentes. Es por ello que, la convolución, es una técnica muy potente, que hace posible la extracción de características dentro de una imagen.

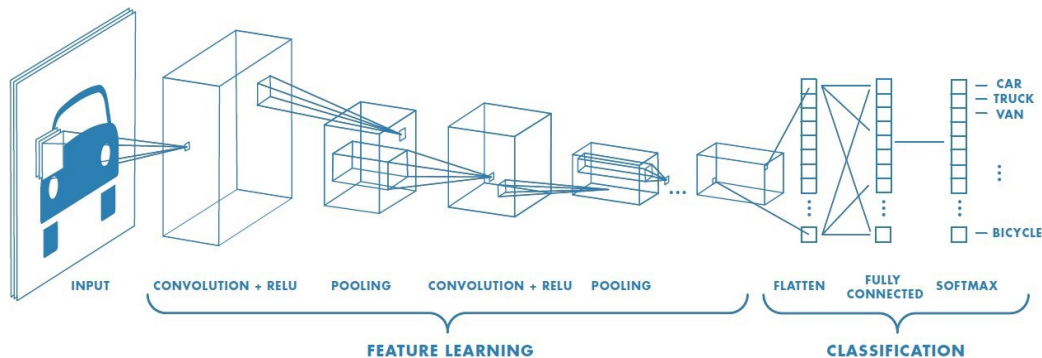


Figura 3.1: Ejemplo de red convolucional

El poder identificar características locales, tiene dos ventajas principales:

- Los patrones que aprenden son *invariantes*. Una vez identificada alguna característica local, puede identificarla también en otras zonas de una imagen, lo cual **proporciona capacidad de generalización**.



- Permite aprender *jerarquías de patrones*. A medida que vamos profundizando en capas de convolución, iremos aprendiendo patrones o información, derivada de la información de las capas anteriores (ver imagen 3.1).

Con todo lo comentado hasta ahora, ya tenemos más claro nuestra organización y cómo vamos a abordar este problema. Una vez tomadas estas decisiones, pasamos a la lectura de dicho conjunto de datos. Pero antes, resumiremos previamente los pasos necesarios para dicha implementación. De dichos pasos, cabe mencionar que los dos primeros, han sido explicados anteriormente, y que el resto siguen aún por exponer.

1. Lectura del conjunto de datos.
2. (*Opcional*) Visualización correcta de los datos.
3. Procesamiento del conjunto de datos.
4. Entrenamiento de la red.
5. Evaluación del conjunto de prueba.
6. (*Opcional*) Visualización de la variación del error durante el entrenamiento.
7. Guardar el vector de predicciones sobre el conjunto de evaluación.

### 3.3 Entrenamiento de la red

Una vez que tenemos nuestros datos cargados, ya podemos pasar a la implementación de la red neuronal convolucional. A continuación se detallan las configuraciones usadas para ello, es decir, los conceptos que vamos hacer uso en el siguiente apartado para el análisis de los distintos modelos creados. Por ello, partiremos de una configuración simple, e iremos poco a poco ajustando sus parámetros así como modificando su topología, con el objetivo de minimizar los errores que cometamos durante la predicción de nuestro modelo.

#### 3.3.1 Variables

Lo primero que vamos a comprender, son los parámetros a definir en nuestra implementación.

- `batch_size`: el tamaño del lote define el número de muestras que se propagarán a través de la red. Por ejemplo, si tenemos 60.000 muestras de entrenamiento y se desea configurar un tamaño de *batch* igual a 10000. El algoritmo toma las primeras 10.000 muestras (de la 1 a la 10.000) del conjunto de datos de entrenamiento y entrena a la red. A continuación, toma las segundas 10.000 muestras (de la 10.001 a la 20.000) y entrena de nuevo a la red. Podemos seguir haciendo este procedimiento hasta que nos hayamos propagado a través de todas las muestras de la red. En nuestro caso, este valor va a ir variando en función del resultado obtenido.
- `numero_clases`: el número de clases para predecir, en nuestro caso 5 etiquetas (de 0 a 4).
- `epochs`: el tamaño de las épocas significa cuántas veces se pasa por el conjunto de entrenamiento. Como el modelo se actualiza cada vez que se procesa un lote, esto significa que se puede actualizar varias veces durante un periodo, es por eso que si el tamaño del lote aumenta, se debe considerar el aumento de las épocas también [Stab]. En nuestro caso, se va a ir variando el número de épocas, y adaptándolas a nuestro objetivo.

- **input\_shape**: el modelo necesita saber la forma de sus datos de entrada, en nuestro caso tenemos imágenes de píxeles en escala RGB (3 canales).
- **loss**: es la función objetivo o de error, la cuál se intentará minimizar mediante el entrenamiento del modelo. Como medida del error, se ha utilizado `categorical_crossentropy`. Esta función, apropiada para problemas de clasificación, se encarga de medir el comportamiento de un modelo dando un valor de probabilidad entre 0 y 1. De esta forma, la pérdida aumenta a medida que la probabilidad pronosticada se desvía de la etiqueta que realmente corresponde a cada muestra del conjunto [Che].
- **optimizer**: es la función de optimización utilizada. Se ha utilizado *Adam* fundamentalmente, ya que se sabe previamente que suele dar buenos resultados. Se han probado otros, como *rmsprop*, pero sin mejores resultados que el anterior.
- **metrics**: es la métrica (o métricas si indicamos más de una) a evaluar, tanto para el conjunto training como el conjunto de test. Utilizaremos en nuestro caso la métrica `accuracy`, dicha medida es el número de todas las predicciones correctas dividido por el número total del conjunto de datos.

### 3.3.2 Tareas

Las tareas que vamos a realizar para la creación de nuestros modelos son:

- **Ajuste de la topología de la red**: consiste en la incorporación de capas a nuestro modelo, es decir, introducir más capas convolucionales.
- **Ajuste de hiperparámetros**: consiste en la modificación de los parámetros de nuestra red, es decir, modificar el tamaño del kernel.
- **Ajuste de algoritmo de optimización**: consiste en la modificación del algoritmo de optimización usado, después de probar varios nos hemos decantado por hacer uso de *adam*.
- **Data augmentation**: es una técnica que crea artificialmente nuevos datos de entrenamiento a partir de los datos de entrenamiento existentes.
- **Transfer learning**: es un método de aprendizaje automático en el modelo desarrollado para una tarea que se reutiliza como punto de partida para un modelo en una segunda tarea.
- **Fine tuning**: es un proceso para tomar un modelo de red que ya ha sido entrenado para una tarea determinada y hacer que realice una segunda tarea similar.

Se debe tener en cuenta que se hará uso de dichas tareas tanto de forma independiente como en combinación unas con otras. Con esto, nos referimos a que podemos mejorar nuestro entrenamiento (y por consiguiente, el modelo), mediante el ajuste de un parámetro. Véase, el ejemplo del ajuste del número de capas densas intermedias. Sin embargo, para obtener resultados competitivos, y por tanto, potentes, será necesario la combinación del ajuste de estos parámetros, normalmente muy diferentes entre ellos, para así explotar lo máximo posible las características de la red.

### 3.3.3 Modelo Secuencial de Keras

El modelo secuencial de Keras permite crear modelos capa por capa para la mayoría de los problemas. Sin embargo, está limitado en el sentido de que no permite crear modelos que compartan capas o que tengan múltiples entradas o salidas [Jov; Kerb].

### 3.3.4 Configuración de los modelos

Una vez creado el modelo, podemos configurarlo añadiendo las distintas capas que queramos. Podemos añadir capas de diferentes topologías, siempre cuidando que la capa de salida y la capa de entrada sean de la forma adecuada. Además, de la capa de entrada y de salida, podremos tener tantas capas como consideremos (y como sea adecuado y mejor para el funcionamiento de nuestro modelo).

Seguidamente, se explicará de forma detallada cada uno de los aspectos tenidos en cuenta, junto con las funciones utilizadas.

#### Capa de entrada

Debemos tener en cuenta que la capa de entrada, tiene que especificar la forma en la que se van a introducir los datos, por lo que siempre tendremos que indicar el valor correspondiente a la variable `input_shape`, a la cuál se le asociará el tamaño o forma de los datos con los que vamos a trabajar. En el caso de nuestras configuraciones, dicho argumento valdrá (224x224x3). Este valor, se debe, por una parte, al tamaño al que hemos reajustado las imágenes 224x224. Al inicio de las pruebas, se hacía uso de (150x150x3), sin embargo, después de distintos experimentos se obtuvo mejores resultados con la primera resolución. Al final, la decisión para elegir dicho tamaño, viene en base de las siguientes razones:

- En este área de trabajo es costumbre utilizar en el ajuste de hiperparámetros aquellos tamaños que han funcionado bien en problemas estudiados previamente. Por ello, sobretodo en nuestro caso, que no tenemos excesivos conocimientos de la materia, decididos coger de partida el tamaño de imagen utilizado por algunos modelos de Deep Learning que, al igual que en nuestro caso, hacen uso de imágenes de *Imagenet*.
- En muchos de los ejemplos, hemos trabajado con modelos y capas pre-entrenadas con dicho ajuste de tamaño para las imágenes, por lo que, para mantener una cierta coherencia, se optó por redimensionar a la misma altura y anchura.

Por otra parte, destacar que cuanto más alto sea dicho valor más grande será la imagen, lo que conlleva un mayor tiempo computacional, y una mejor representación de los posibles elementos que pueden definir la escena. Dentro de la literatura, se suele hacer uso de 150 y/o 224 píxeles. Recordemos que 224x224 representa los píxeles de la imagen y el tercer valor (que como se ha indicado es 3), representa que tenemos una imagen en color RGB, y que por tanto tenemos tres canales (`input_shape`).

#### Capa de neuronas convolucional

Para la capa de neuronas convolucional, se ha hecho uso de la función `layer_conv_2d(...)` en donde `filters` representa el número de nodos en la capa y `activation` es la función de activación en dicha capa:

```
1 layer_conv_2d(filters=64, kernel_size = c(5, 5),  
2             activation = "relu", input_shape = c(224, 224, 3))
```

- `filters = 64`: representa el número de nodos en la capa. Se tiene **64 características** o nodos.
- `kernel_size = c(5, 5)`: dispone de un **kernel de tamaño 5**, el cual determina los píxeles que se tendrán en cuenta cuando se procese la imagen. La decisión del número está predefinida, siendo además siempre impar, expliquemos el porqué de la máscara: la función

Gaussiana está centrada en el origen, si consideramos los valores más significativos de ambos lados, tanto los de la derecha como los de la izquierda, dichos valores se encuentran a una distancia 3 del centro (origen), por tanto, cogemos los valores dentro del intervalo  $[3\sigma, 3\sigma]$ . Por consiguiente, el tamaño de nuestra máscara sería de  $3\sigma \cdot 2 + 1$ , donde:

- $3\sigma$ : hace referencia a nuestro intervalo  $[3\sigma, 3\sigma]$
- $\cdot 2$ : hace referencia a que debemos contabilizar tanto los valores de la derecha como de la izquierda
- $+1$ : es el valor del centro (origen) y que debemos de contar, el cual hace que las máscaras sean siempre de tamaño impar

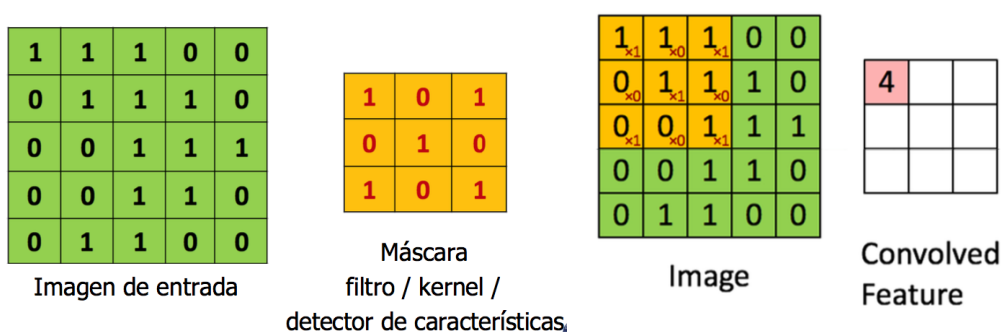


Figura 3.2: Proceso de convolución

- `activation='relu'`: se usa una función de activación de unidades lineales rectificadas [Kera], es decir, las ReLUs permiten el paso de todos los valores positivos sin cambiarlos, pero asigna todos los valores negativos a 0. Aunque existen funciones de activación aún más recientes, la mayoría de las redes neuronales actuales utilizan ReLU o una de sus variantes [Art].
- `input_shape=input_shape`: el modelo necesita saber la forma de sus datos de entrada, en nuestro caso tenemos imágenes de 150x150 píxeles en escala RGB (3 canales).

### Capa oculta - Pooling

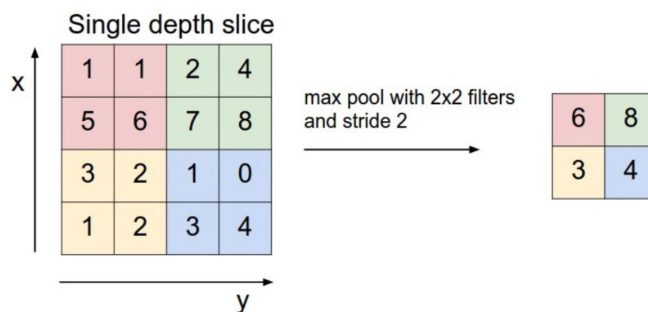


Figura 3.3: Las capas de agrupamiento se utilizan para aplicar la reducción de muestreo no lineal en los mapas de activación



Para tener un control del sobreaprendizaje y/o evitar un ajuste excesivo, las capas de agrupación se utilizan para aplicar la reducción de muestreo no lineal en los mapas de activación, es decir, para la reducción de la dimensionalidad. Es por eso, que se crea una capa de pooling que toma el valor máximo llamado MaxPooling2D, con un tamaño de de 2x2, la cual reduce progresivamente el espacio de representación, reduciendo el número de operaciones necesarias para todas las capas siguientes, pero transmitiendo la información válida de la capa anterior [Meda]. *A una capa Pooling a menudo le sigue una o dos capas de convolución.*

```
1 layer_max_pooling_2d(pool_size = c(2, 2))
```

### Capa oculta - Dropout

Esta capa es una técnica de regularización que evita el sobreaprendizaje. Se configura para excluir aleatoriamente un determinado porcentaje de las neuronas de la capa, ignorando un número determinado de neuronas en la fase de entrenamiento. Es decir, para cada caso de entrenamiento, se omite aleatoriamente cada neurona oculta con una probabilidad. Con esta técnica, lo que se consigue es evitar que las neuronas ocultas dependan/confíen demasiado en el trabajo de otras neuronas ocultas de su misma capa.

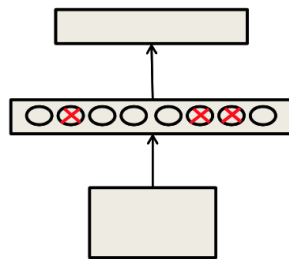


Figura 3.4: Capa Dropout para técnica de regularización

```
1 layer_dropout(0.3)
```

### Capa oculta - Flatten

Esta capa permite aplanar la red (salida unidimensional), para así poder utilizar una capa densa a continuación y, obtener un comportamiento adecuado [Staa].

```
1 layer_flatten()
```

Permite trabajar con un input unidimensional, ya que en caso contrario, obtendríamos como salida de nuestra red Dense una secuencia de vectores que nos llevaría a un comportamiento inadecuado de la red (debido a las conversiones de dimensión que se hacen de forma interna). Por tanto, es necesario que, antes de añadir una capa Dense a nuestros datos, modifiquemos la entrada que vamos a facilitarle a esta (ver figura 3.5).

### Capa oculta - Dense

Una vez que se ha aplanado la red, se hace uso de una capa densa completamente conectada con 256 neuronas y función de activación ReLU. Estas capas, a menudo, se encuentran al final de la red neuronal [Meda].

```
1 layer_dense(units = 256, activation = "relu")
```

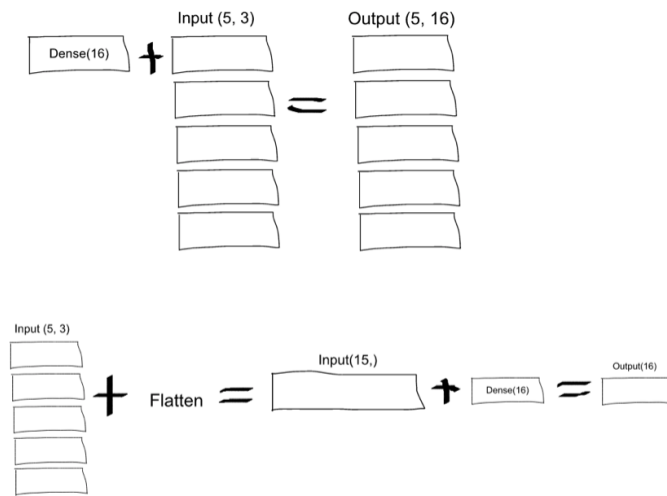


Figura 3.5: Capa Flatten [Staa]

### Capa de salida

Una vez que ya tenemos las capas ocultas definidas pasamos a la capa de salida. La capa de salida tiene 5 neuronas para las 5 clases y una función de activación *softmax* para producir predicciones probabilísticas para cada clase. A menudo la función de activación de ReLu se usa en capas ocultas, mientras que la capa final generalmente consiste en una función de activación de SoftMax. La idea es que al apilar capas de funciones lineales y no lineales, podemos detectar una amplia gama de patrones y predecir con precisión una etiqueta para una imagen determinada [Meda].

```
1 layer_dense(units = 5, activation = "softmax")
```

## 3.4 Compilar y probar el modelo

A continuación, pasamos a configurar varios modelos de Keras para el entrenamiento, con el fin de probar qué tan bien predice el tiempo de adopción de una mascota. En donde para afinar el modelo, se realizan diversas pruebas como ajustar los distintos hiperparámetros del modelo para reducir la función de pérdida (entropía cruzada categorial o `categorical_crossentropy`) que se optimiza usando el optimizador ADAM [Meda].

Primero, vamos a explicar todos los parámetros que le pasamos a la función `compile(...)` para compilar el modelo:

```
1 model %>% compile(loss = 'categorical_crossentropy', optimizer = 'adam',
2                   metrics = c('categorical_accuracy', 'accuracy'))
3 )
```

- `loss = 'categorical_crossentropy'`: es el nombre de la función objetivo, si el modelo tiene múltiples salidas, puede utilizar una pérdida diferente en cada salida pasando un diccionario o una lista de objetivos. El valor de pérdida que será minimizado por el modelo será entonces la suma de todas las pérdidas individuales. Se usa `categorical_crossentropy` en entrenamientos de clasificación multiclase.

- `optimizer = 'adam'`: existen varios métodos para minimizar el error, en este caso se ha hecho uso del optimizador Adam, el cual es un algoritmo de optimización que puede utilizarse en lugar del procedimiento clásico de descenso de gradiente estocástico. Además, según estudiamos, Adam es un algoritmo muy bueno ya que logra buenos resultados rápidamente.
- `metrics = c('categorical_accuracy', 'accuracy')`: lista de métricas a ser evaluadas por el modelo durante el training y el testing, usualmente se usa *accuracy*, para determinar la precisión del modelo. Cabe decir que ambas métricas devuelven el mismo resultado, y que simplemente se muestran dos veces en vistas a asegurarnos de que no cometemos ningún fallo.

Segundo, explicamos los parámetros que le pasamos a la función `fit_generator(...)` para entrenar el modelo en un número fijo de épocas:

```
1 history <- model %>% fit_generator(train_data,
2                                   steps_per_epoch = 1,
3                                   epochs = 5,
4                                   validation_data = validation_data,
5                                   validation_steps = 1)
```

- `train_data`: conjunto de entrenamiento.
- `steps_per_epoch = 1`: número total de pasos (lotes de muestras) para obtener el rendimiento del generador antes de declarar finalizada una época y comenzar la siguiente. Normalmente debería ser igual al número de muestras si su conjunto de datos se divide por el tamaño del lote. En este caso hemos hecho uso de un paso por época.
- `epochs = 5`: número de épocas para entrenar el modelo, en este caso hemos hecho uso de 5 épocas.
- `validation_data = validation_data`: datos sobre los cuales se va a evaluar la pérdida (conjunto de prueba). Se debe tener en cuenta que, para ello, se ha separado un pequeño porcentaje del conjunto de datos de train, para ser utilizado específicamente para este uso (el modelo no será entrenado en estos datos) [Kerd].
- `validation_steps = 1`: número total de pasos (lotes de muestras) a producir desde el generador antes de detenerse al final de cada época. En este caso hemos hecho uso de un paso por época.

### 3.5 Evaluación del conjunto de prueba

Una vez que hemos entrenado nuestra red, pasamos a la evaluación del conjunto de prueba y así, a la obtención de la tasa de error de dicho conjunto. Para ello, se carga el modelo previamente, y con el método `evaluate_generator(...)` se calcula la pérdida, así como cualquier métrica definida al compilar el modelo. En nuestro caso, la métrica usada es la precisión que se calcula de las imágenes de prueba utilizando los pesos de red indicados por el modelo guardado [Gre].

```
1 model %>% evaluate_generator(test_data, steps = 5)
```

Una vez, que hemos comprendido las funciones usadas para realizar la implementación, se pasa a explicar los distintos modelos obtenidos. Se debe tener en cuenta, que no se comentarán todas las modificaciones realizadas en los modelos, ya que se han realizado demasiadas, solo se comentarán las más significativas y las que nos han llevado a cumplir el objetivo. De otra manera, en este capítulo hemos hecho uso de unos parámetros definidos para realizar las distintas explicación, lo que no

determina que se puedan modificar para la obtención de distintos modelos. En la siguiente sección, entraremos en estos aspectos con más detalle.

***Nota:** Se debe tener en cuenta, que no siempre obtendremos resultados similares, debido a la componente aleatoria.*





## 4. Análisis

### 4.1 Análisis del problema

En este apartado se comentarán los modelos realizados más significativos, hasta la obtención del modelo que nos ha proporcionado el mínimo error alcanzado para el conjunto de prueba. No tiene sentido escribir todas las pruebas y ajustes llevados a cabo sobre los modelos, ya que muchas de esas configuraciones realizadas han sido cambiando parámetros como los nodos de las capas convolucionales o simplemente las épocas de entrenamiento. Sin embargo, haremos un recorrido por las pruebas que sí nos han funcionado, así como las conclusiones obtenidas a partir de ellas.

Como se comentó en el apartado anterior, disponer de conocimientos previos de redes neuronales nos ha llevado a la realización de una red neuronal convolucional de manera directa, sin pasar antes por otros tipos de modelos. Además, de haber hecho uso de Keras con Python en otra asignatura del Máster. A continuación, se explican de manera resumida los distintos modelos que se han probado.

### 4.2 Evaluación y métricas para los modelos

Para evaluar los distintos modelos, se ha en cuenta el valor obtenido por dos medidas distintas.

- **Función de Loss:** una medida que nos permitirá saber cuánto se alejan los resultados de la etiqueta que realmente les corresponde. Esta medida es útil para nuestro problema, puesto que, no es lo mismo que, para una imagen con etiqueta 0, le asignes un 1 (lo cuál sería, predecir que el rango de tiempo necesario es justo el siguiente al real), a asignarle un 5 (con lo que estás expresando justo lo contrario). Esta desviación del error, es útil evaluarla, y con esta función de pérdida, podemos guiarnos en esa línea.
- **Accuracy o Categorical Accuracy:** medida que nos permitirá saber, en una predicción, el porcentaje de muestras que somos capaces de predecir correctamente.

No obstante, seguiremos fundamentalmente *Loss*, por lo comentado anteriormente.

## 4.3 Configuración de los modelos

### 4.3.1 Modelo simple

Primero, nos hemos centrado en obtener modelos simples, a partir del proporcionado por el profesor. Para ello, se han hecho uso de distintos ajustes tanto para la **topología de la red**, como los **hiperparámetros**, como el **algoritmo de optimización**.

#### Modelo I

El primer modelo tiene una resolución de imagen de 150x150 (`target_size = (150,150)`), consta y `batch_size` a 100 con las siguientes capas. De igual manera, se ha hecho uso de distintas épocas (`epochs = 3` y `epochs = 8`), para comprender el peso de las épocas en el entrenamiento de la red:

1. Capa de convolución 2D con 32 filtros y kernel de tamaño 5x5
2. Capa de activación con función reLu".
3. Capa MaxPooling 2D con `pool_size=2x2`
4. Capa de convolución 2D con 64 filtros y kernel de tamaño 3x3
5. Capa de activación con función reLu".
6. Capa flatten.
7. Capa densa con 512 neuronas.
8. Capa de activación con función reLu".
9. Capa densa con 256 neuronas.
10. Capa de activación con función reLu".
11. Capa de salida con función de activación *softmax*.

```

1  # Configuración del modelo I
2  model <- keras_model_sequential() %>%
3    layer_conv_2d(filters = 32, kernel_size = c(5, 5), activation = "relu",
4                  input_shape = c(150, 150, 3)) %>%
5    layer_max_pooling_2d(pool_size = c(2, 2)) %>%
6    layer_conv_2d(filters = 64, kernel_size = c(3, 3),
7                  activation = "relu") %>%
8    layer_max_pooling_2d(pool_size = c(2, 2)) %>%
9    layer_flatten() %>%
10   layer_dense(units = 512, activation = "relu") %>%
11   layer_dense(units = 256, activation = "relu") %>%
12   layer_dense(units = 5, activation = "softmax")

```

Salida para el Modelo I con 3 épocas	Salida para el Modelo I con 8 épocas
Tiempo de Entrenamiento: 696 segundos	Tiempo de Entrenamiento: 1845 segundos
Train Loss: 1.4108	Train Loss: 0.2592
Train Accuracy: 0.3333	Train Accuracy: 0.9183
Validation Loss: 1.4744	Validation Loss: 4.0264
Validation Accuracy: 0.2808	Validation Accuracy: 0.2901
Test Loss: 1.47964	Test Loss: 3.967941
Test Accuracy: 0.2759242	Test Accuracy: 0.285447

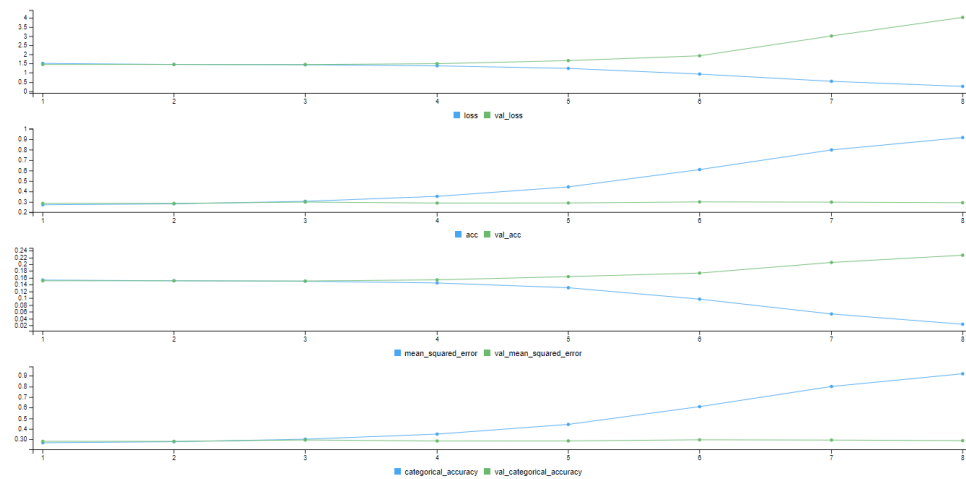


Figura 4.1: Gráficas para el modelo I con 3 épocas

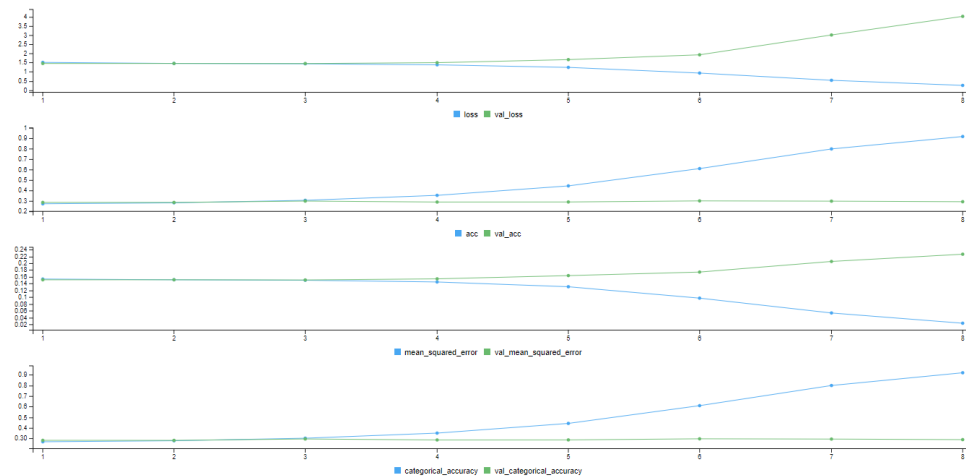


Figura 4.2: Gráficas para el modelo I con 8 épocas

Respecto las épocas, acabamos de comprobar como subir el número de épocas no supone una mejora sustancial en los resultados, pero sí un aumento en el tiempo de entrenamiento, como podemos ver para el modelo con 8 épocas. Por tanto, por más épocas que incluyamos no estamos mejorando, aunque a veces esta no mejora dependa de componentes aleatorios. En nuestro caso, queremos obtener un equilibrio entre tiempo de entrenamiento y calidad del modelo. Por lo tanto, para cada modelo que hemos obtenido se ha ejecutado con distintas épocas.

Igualmente hablando en términos de sobreajuste, destacar que para el modelo con 8 épocas se aprecia un poco de sobreajuste, pero al disponer de tan pocas imágenes para el entrenamiento, dicho sobreajuste no determina mucho la salida del conjunto de test.

**Loss del primer modelo: 1.47964**

**Modelo II**

El segundo modelo tiene una resolución de imagen de 150x150, consta de 5 épocas (epochs=5) y batch\_size a 100 con las siguientes capas:

1. Capa de convolución 2D con 32 filtros y kernel de tamaño (3x3)
2. Capa de activación con función reLu"
3. Capa MaxPooling 2D con pool\_size=2x2
4. Capa de convolución 2D con 64 filtros y kernel de tamaño (3x3)
5. Capa de activación con función reLu"
6. Capa MaxPooling 2D con pool\_size=2x2
7. Capa Dropout con 0.3
8. Capa flatten
9. Capa densa con 512 neuronas.
10. Capa de activación con función reLu"
11. Capa de salida con función de activación *softmax*.

```

1 # Configuración del modelo II
2 model <- keras_model_sequential() %>%
3   layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu",
4     input_shape = c(150, 150, 3)) %>%
5   layer_max_pooling_2d(pool_size = c(2,2)) %>%
6   layer_conv_2d(filters = 64, kernel_size = c(3,3),
7     activation = "relu") %>%
8   layer_max_pooling_2d(pool_size = c(2,2)) %>%
9   layer_dropout(0.3) %>%
10  layer_flatten() %>%
11  layer_dense(units = 512, activation = "relu") %>%
12  layer_dense(units = 5, activation = "softmax")

```

Salida para el Modelo II
Tiempo de Entrenamiento: 1624 segundos
Train Loss: 1.3465
Train Accuracy: 0.3823
Validation Loss: 1.4777
Validation Accuracy: 0.3043
Test Loss: 1.476669
Test Accuracy: 0.2974526

**Loss del segundo modelo: 1.476669**

### 4.3.2 Modelo simple + Data augmentation

Una vez, experimentado con modelos más simples, se va a aplicar **Data augmentation**. Esta técnica está orientada principalmente a intentar mejorar y obtener mejores resultados en el proceso de extracción de características.

Para ello, se lleva a cabo un incremento de los datos (en nuestro caso, imágenes) con los que vamos a entrenar nuestro modelo, aplicando distintas transformaciones, como las que se pueden observar en la imagen 4.4. Esta técnica tiene dos ventajas fundamentalmente. En primer lugar, porque nos permite aumentar el número de características e información relevante sobre la que aprender, y, por otra parte, porque con este tipo de técnicas, podemos centrarnos en aprender lo realmente importante de las imágenes, y no tanto como aprender del conjunto real que tenemos (lo cuál, puede llevar a sobreaprendizaje).

En nuestro caso, dado que no tenemos una cantidad inmensa de imágenes, añadir técnicas de este tipo puede ayudarnos a intentar generalizar mejor fuera de nuestros modelos, ya que el añadir nuevos datos derivados de los que ya tenemos, sí que nos puede ayudar a encontrar características en las imágenes del entrenamiento.

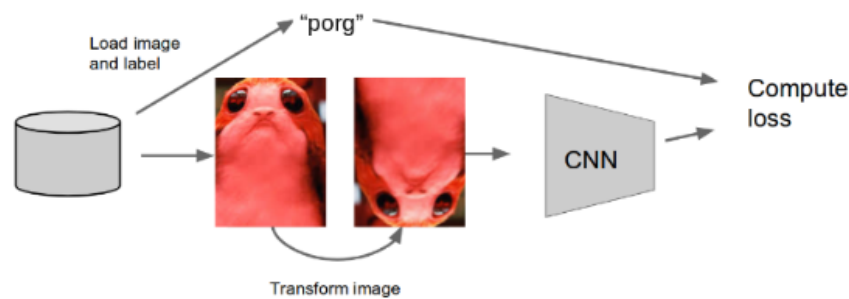


Figura 4.3: Funcionamiento de Data augmentation I

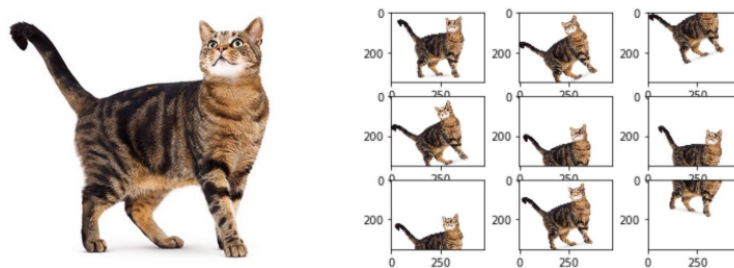


Figura 4.4: Funcionamiento de Data augmentation III

Sin embargo, cabe decir, que se trata de una técnica muy costosa computacionalmente, que provocará que nuestros modelos requieran de un mayor tiempo de entrenamiento, debido a la cantidad de datos que éstos tendrán que manejar. Sin embargo, esta técnica debe ser aplicada cuando sea necesario. Con ello, nos referimos a que aumentar el conjunto de datos, será algo adecuado a realizar siempre y



cuando las técnicas que utilicemos sean capaces de mejorar el conjunto de datos actual. Puede no interesar, para nuestro conjunto de datos, tener imágenes en posiciones o rotaciones en las que nunca nos la encontraremos, o con únicamente partes que no sean interesantes para nuestro problema.

### Modelo III

El modelo III es el mismo modelo que el II, solo que en este caso aplicamos **Data Augmentation** sobre el conjunto de datos. Se pueden ver los cambios necesarios para implementar esta técnica en el siguiente trozo de código.

```

1 data_augmentation_datagen <- image_data_generator(
2   rescale = 1/255,
3   rotation_range = 40,
4   width_shift_range = 0.2,
5   height_shift_range = 0.2,
6   shear_range = 0.2,
7   zoom_range = 0.2,
8   horizontal_flip = TRUE,
9   fill_mode = "nearest"
10 )
11
12 train_augmented_data <- flow_images_from_directory(
13   directory = train_images_dir,
14   generator = train_datagen,
15   target_size = c(150, 150),      # (w, h) --> (150, 150)
16   batch_size = 100,              # grupos de 100 imágenes
17   class_mode = "categorical"     # etiquetas categóricas
18 )

```

- `rotation_range`: rango dentro del cual se pueden rotar imágenes al azar, valor en grados entre (0-180).
- `width_shift` y `height_shift`: rangos dentro de los cuales se pueden trasladar imágenes al azar vertical u horizontalmente.
- `shear_range`: para aplicar transformaciones de cizalla al azar.
- `zoom_range`: para hacer zoom al azar dentro de imágenes.
- `horizontal_flip`: para hacer girar al azar la mitad de las imágenes horizontalmente.
- `fill_mode`: estrategia utilizada para rellenar los píxeles recién creados, que pueden aparecer después de una rotación o un desplazamiento de anchura/altura.

#### Salida para el Modelo III

Tiempo de entrenamiento: 1581 segundos
Train Loss: 1.3887
Train Accuracy: 0.3529
Validation Loss: 1.4565
Validation Accuracy: 0.3057
Test Loss: 1.458036
Test Accuracy: 0.2990822

Con las transformaciones comentadas, se pueden realizar transformaciones en las fotografías como las que se puede ver en la imagen 4.5.

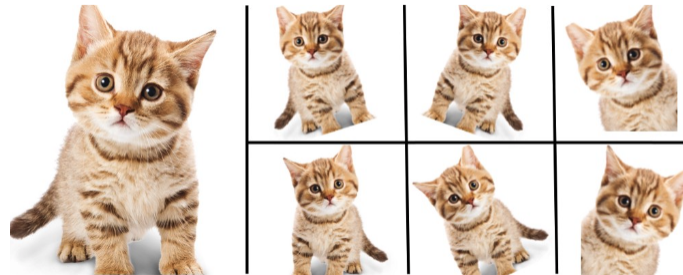


Figura 4.5: Funcionamiento de Data augmentation II

Por ello, se observa como los resultados del tercer modelo, mejoran los obtenidos respecto al segundo. Sin embargo, no debemos olvidar que el objetivo de Data augmentation es crear artificialmente nuevos datos de entrenamiento a partir de los datos de entrenamiento existentes. Y como en nuestro caso, no disponemos de una cantidad inmensa de imágenes, problema que hemos venido comentado a lo largo de la práctica, este hecho también limita la obtención de una mejora sustancial.

De igual forma, hay que tener en cuenta en estos casos la naturaleza de nuestras imágenes. Muchas de estas operaciones, pueden, a priori, no tener sentido en nuestro conjunto de datos. Esto se debe a que, rara vez, nos encontraremos una foto donde se encuentre la foto de una mascota al contrario, (véase imágenes 4.6). Es decir, no nos encontraremos en nuestro conjunto de entrenamiento, imágenes giradas 180 grados, de manera que la mascota se visualice al revés.



Figura 4.6: Rotación de una imagen del conjunto de entrenamiento

Este caso, es distinto de si tuviésemos, por ejemplo, imágenes de medicamentos (ver imagen 4.9), donde, sí puede tener sentido intentar estudiar estos distintos tipos de rotaciones, porque fácilmente se pueden dar en nuestro caso.

En resumen, con este breve ejemplo, pretendemos decir, que muchas de las imágenes generadas con Data Augmentation, sobre todo aquellas que impiden ver la mascota sin girar la cabeza, pueden no tener sentido, por lo que esta técnica no aportará todas las ventajas que sí podrían darse en conjuntos



Figura 4.7: Data Augmentation en otro tipo de objetos

de entrenamiento de otra naturaleza (como es el caso comentado con los medicamentos)[Sci]. Con todo lo comentado, pensamos que, este hecho, es una de las posibles razones por las que no hemos podido mejorar nuestros resultados de una manera notable al hacer uso de esta técnica.

Para el mismo número de épocas en ambos modelos, hemos concluido que no supone una mejora sustancia, sin embargo, al hacer uso de más datos en el entrenamiento, puede ser aconsejable subir las épocas de 5 a 10. En el nuevo tercer modelo para 10 épocas se obtiene:

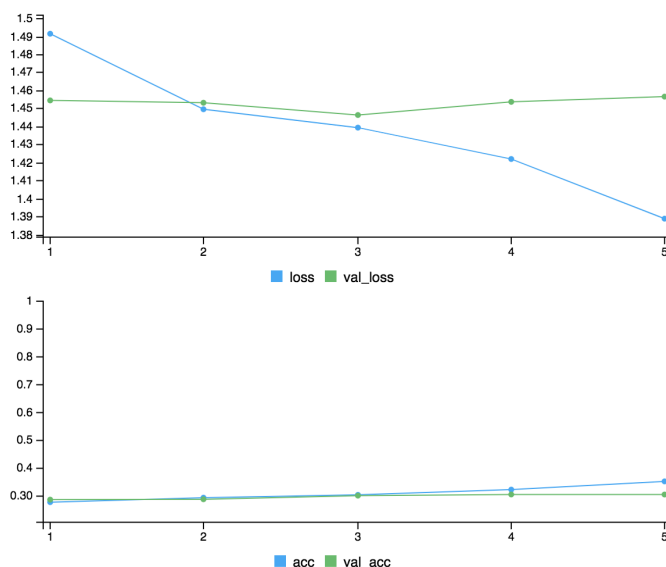


Figura 4.8: Gráficas para el modelo III con 5 épocas

- loss: 2.601714
- accuracy: 0.3141779
- Tiempo de entrenamiento: 3128 segundos

Si tenemos en cuenta solo la medida obtenida para el *accuracy* hemos aumentado el valor de 0.29 a 0.31, esto supone una mejora al haber hecho uso de más imágenes, ya que cómo hemos ido

comentando nuestro conjunto de imágenes dispone de pocas fotografías. No obstante, se aprecia como el *loss* aumenta considerablemente, lo que puede venir a causa de un pequeño sobreaprendizaje.

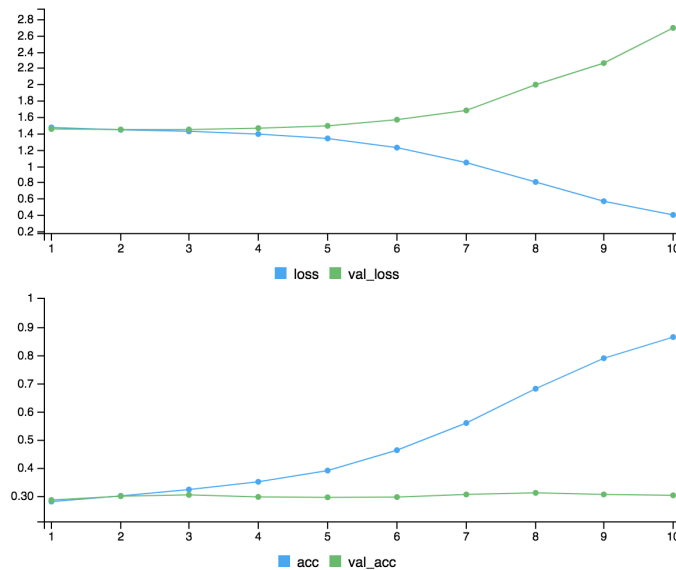


Figura 4.9: Gráficas para el modelo III con 10 épocas

**Loss del tercer modelo: 1.458036**

### 4.3.3 Transfer learning

La segunda técnica a aplicar es **Transfer learning**. Es un método de aprendizaje automático en el que un modelo desarrollado para una tarea se reutiliza como punto de partida para un modelo en una segunda tarea. Es decir, los modelos pre-entrenados se suelen utilizar como punto de partida en las tareas de procesamiento de lenguaje natural y visión computacional dados los vastos recursos de cómputo y tiempo requeridos para desarrollar modelos de redes neuronales en estos problemas.

Hemos visto, a lo largo de los modelos ya expuestos, que una de las deficiencias principales a la que nos estamos enfrentando es la poca cantidad de datos con la que estamos trabajando. Debemos tener en cuenta, dos cosas:

- El número de muestras disponibles para realizar un entrenamiento con Redes Neuronales, suele ser crucial a la hora de obtener un modelo robusto, que sea capaz de generalizar fuera del conjunto de datos con el que entrenamos. Tenemos un número de imágenes pequeño, y debemos de ser capaces de extraer, de dichas imágenes, características de calidad, y esto puede no ser viable si nos limitamos a utilizar operaciones de convolución sencillas.
- Nuestro problema, no se trata de identificar objetos, sino de ser capaces de distinguir, en las imágenes, características que determinen este tipo de decisiones. Estas características, son difíciles de determinar a nivel humano, por lo que para una máquina aún más complejo.

Bajo la necesidad de obtener características muy potentes, junto con los pocos datos que disponemos para generarlas, surge la idea de utilizar en esta práctica la técnica de Transfer Learning. Se trata de una técnica comúnmente utilizada para problemas de modelado predictivo que utilizan datos de imagen como entrada. En nuestro caso lo vamos a enfocar hacia un modelo pre-entrenado. De esta forma, vamos a utilizar una red ya entrenada para obtener características de imágenes para facilitarnos la tarea de determinar, por nuestra propia cuenta, las características relevantes de dichas imágenes. La principal ventaja reside en que nos ayudará a suplir la necesidad de mayor número de imágenes para obtener filtros de calidad, porque, ya los habrá utilizado en el momento de su entrenamiento.

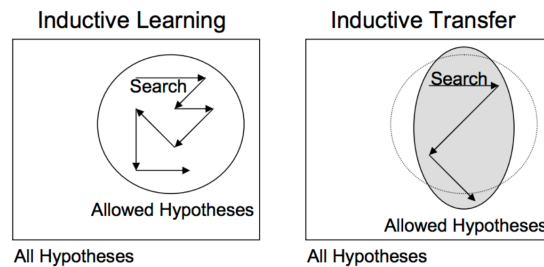


Figura 4.10: Funcionamiento transfer para Transfer learning

Mejoran los resultados, pero el problema que tenemos, no deriva tanto de las características, sino de la dificultad, de predecir de una foto, a si se adopta o no. Se va a hacer uso en esta práctica de dos modelos pre-entrenados:

- **Inception V3:** es un modelo de reconocimiento de imágenes que ha alcanzado el 78.15 % de Accuracy en el conjunto de datos de Imagenet (de donde proceden las imágenes de nuestro conjunto de datos). Tal y como se puede ver en la imagen ??, está construido a partir de bloques con capas convolucionales, de Average Pooling, de Max Pooling, de concatenación y densas. La función de LOss que utiliza es *Softmax* y para la activación de entradas hace uso de *Batchnorm*. Además, acepta tamaño de imagen superior a 75x75, por lo cuál podemos usarla para nuestro modelo [Clo].

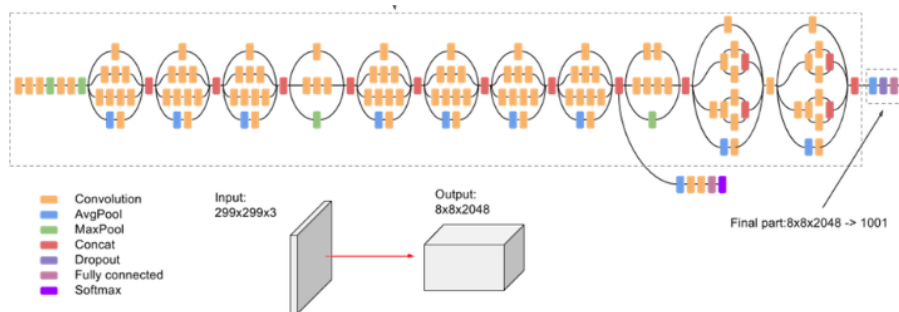


Figura 4.11: Arquitectura de Inception V3

- **VGG16:** es un modelo de red neural convolucional propuesto por K. Simonyan y A. Zisserman de la Universidad de Oxford. Es una mejora sobre otra red ya entrenada, conocida por el



nombre de *AlexNet*. Es ampliamente conocida y utilizada en el ámbito de la Visión por Computador. Su estructura se puede ver en la imagen 4.12.

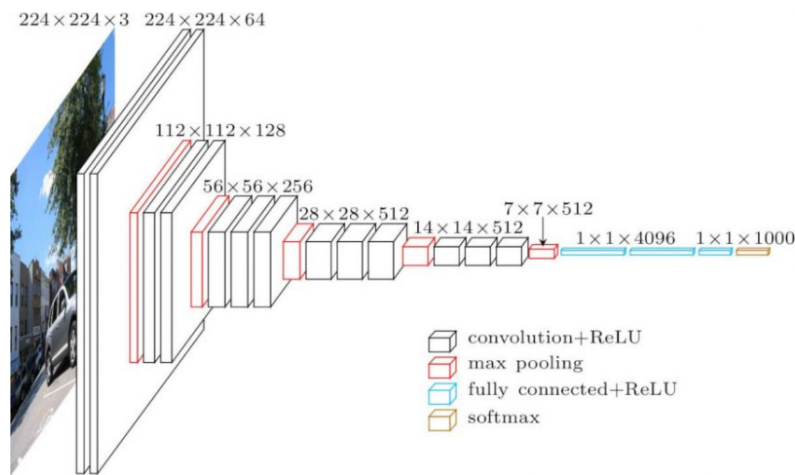


Figura 4.12: Arquitectura de VGG16

#### Modelo IV

El modelo IV es el obtenido al utilizar la técnica **Transfer learning con la red Inception V3**. Se va hacer uso de una resolución para la imagen de 224x224, como comentamos previamente, y consta de 8 épocas (epochs=8) y batch\_size a 100.

1. Output de la Red Inception V3
2. Capa de media global de pooling 2D.
3. Capa densa con 512 neuronas.
4. Capa de activación con función reLu".
5. Capa Dropout con un valor de 0.2
6. Capa densa con 256 neuronas.
7. Capa de activación con función reLu".
8. Capa de salida con función de activación *softmax*.

#### Salida para el Modelo IV

Train Loss: 1.4247
Train Accuracy: 0.3164
Test Loss: 1.458498
Test Accuracy: 0.2916202
Validation Loss: 1.4534
Validation Accuracy: 0.2962

Como se puede observar, a priori, los resultados que hemos obtenido no varían en cantidad de los que hemos ido obteniendo hasta ahora. En principio, puede parecer extraño, pero sí podemos poner

en constatar dos hechos, de entre todos los resultados que hemos visto en las pruebas:

1. Si tenemos en cuenta el valor de Loss, tiene mejor funcionamiento que nuestros modelos individuales, con los cuáles hubo situaciones en las que este valor se nos disparó. Con los modelos con Transfer Learning, el valor de error siempre ha oscilado entre 1.4 y 1.6, el cuál coincide con los mejores modelos alcanzados sin Transfer Learning.
2. Se trata de modelos, que nos permiten obtener características de imágenes, pero no están adecuados al problema concreto que estamos abordando (sino que nos permiten obtener características mucho más genéricas, y aplicables a muchos problemas). Esto da pie a pensar, que si queremos mejorar los resultados, tendremos que intentar especializar estos modelos, con técnicas como Fine Tuning, que veremos en el siguiente apartado.

**Loss del cuarto modelo:** 1.4247

#### 4.3.4 Transfer learning + Fine tuning

Viendo los buenos resultados obtenidos, y en nuestra opinión, más fiables, se ha pensado en aplicar varias técnicas combinadas. En este caso, se va a crear un modelo haciendo uso de las técnicas de **Transfer learning + Fine tuning**. En el apartado anterior, se explicó Transfer learning, en este vamos a detallar en que consiste el **Fine tuning**.

**Fine tuning** es una técnica ampliamente utilizada para la reutilización de modelos, complementaria a la extracción de características. Con ella se ajusta ligeramente las representaciones más abstractas del modelo que se está reutilizando, con el fin de hacerlas más relevantes para el problema. Consiste en descongelar algunas de las capas superiores/últimas de un modelo entrenado y utilizado para la extracción de características, y entrenar conjuntamente tanto la parte recién añadida del modelo como estas capas superiores.

De esta manera, lo que queremos es realimentar estas últimas capas con la información que se propaga desde nuestro conjunto de datos, con el objetivo de que se especialice y se ajuste más concretamente al problema que intentamos resolver, evitando así, representaciones de información abstractas, que pueden representar bien la imagen y obtener buenas características, pero quizá no las mejores para nuestro problema en sí. Buscamos conectar de una manera más natural, la red preentrenada con el entrenamiento posterior que queremos realizar sobre las características de las imágenes que nos proporciona la red preentrenada.

Los pasos para aplicar Fine tuning a una red son los siguientes:

1. Añadir nuestra red configurada encima de una red ya entrenada.
2. Congelar la red base.
3. Entrenar la parte de la red que se ha añadido.
4. Descongelar algunas capas de la red ya entrenada.
5. Entrenar conjuntamente estas dos capas y la parte que se añadió.

Como se puede observar, exceptuando los dos últimos pasos, este procedimiento es el realizado en el apartado anterior.

### Modelo V

El modelo V es el obtenido al utilizar la técnica **Transfer learning más Fine Tunning con la red Inception V3**. Se va hacer uso de una resolución para la imagen de 224x224, como comentamos previamente, y consta de 8 épocas (epochs=8) y batch\_size a 100.

En primer lugar, mencionar que se llevaron a cabo diferentes combinaciones del número de capas últimas a descongelar, sin tener realmente resultados diferenciados. Como de esta manera, no conseguimos mejorar, tanto Loss como Accuracy, probamos una estrategia distinta, y intentamos congelar un número de capas lo suficientemente grande como para poder ver si realmente obteníamos cambios en nuestros resultados. A continuación, se puede ver el modelo llevado a cabo.

1. Output de la Red Inception V3 (Se congelan las 13 primeras capas).
2. Capa de media global de pooling 2D.
3. Capa densa con 512 neuronas.
4. Capa de activación con función "reLu".
5. Capa Dropout con un valor de 0.2
6. Capa densa con 256 neuronas.
7. Capa de activación con función "reLu".
8. Capa de salida con función de activación *softmax*.

#### Salida para el Modelo V

Train Loss: 1.4598
Train Accuracy: 0.2799
Validation Loss: 1.4590
Validation Accuracy: 0.2584
Test Loss: 1.458962
Test Accuracy: 0.258427

Tal y como se puede consultar en la bibliografía, cuando se realiza Fine Tuning debemos tener en cuenta dos aspectos. El primero de ellos, es que las capas de la red pre-entrenada que nos interesan entrenar de manera conjunta con nuestra red son las últimas, ya que son aquellas más especializadas, y las que queremos orientar hacia nuestro problema concreto. El segundo, que es el que nos interesa en este caso, es que las redes pre-entrenadas ya comentadas anteriormente, trabajan con miles de parámetros. Aplicadas a un problema como el nuestro, el cuál es relativamente pequeño, corremos el riesgo de sobreaprendizaje. Con lo cuál, debemos tener cuidado con el número de capas que congelamos y el número restante que utilizamos para entrenar, puesto que, el entrenamiento de un número muy grande de parámetros para un conjunto de datos pequeño puede desencadenar problemas de sobreajuste. **Este último hecho, es el que podemos comprobar en los resultados obtenidos en este modelo, ya que, dejando tan solo las primeras capas congeladas (un número pequeño en comparación con las capas totales), hemos obtenido peor valor de Accuracy que el alcanzado con problemas sencillos sin uso de Transfer Learning.** Este es un claro ejemplo

de cómo, el uso de una técnica potente, puede llevarnos a la situación contraria, e impedirnos, debido a su gran potencia, conseguir buenos resultados.

**Loss del quinto modelo: 1.4598**

### Modelo VI

En vistas a intentar mejorar los resultados obtenidos, se probó una segunda red ya pre-entrenada, con la cuál pensábamos que también podíamos obtener buenos resultados. Esta red, es conocida como VGG16 (ya comentada anteriormente).

El modelo VI es el obtenido al utilizar la técnica **Transfer learning más Fine-Tuning con la red VGG16**. Se va hacer uso de una resolución para la imagen de 224x224 (la misma utilizada por dicha red), como comentamos previamente, y vamos a entrenarla con distinto número de épocas. A continuación podemos ver la topología de la red.

1. Output de la Red VGG16 (se congelan las 13 primeras capas).
2. Capa de media global de pooling 2D.
3. Capa densa con 512 neuronas.
4. Capa de activación con función reLu".
5. Capa Dropout con un valor de 0.2
6. Capa densa con 256 neuronas.
7. Capa de activación con función reLu".
8. Capa de salida con función de activación *softmax*.

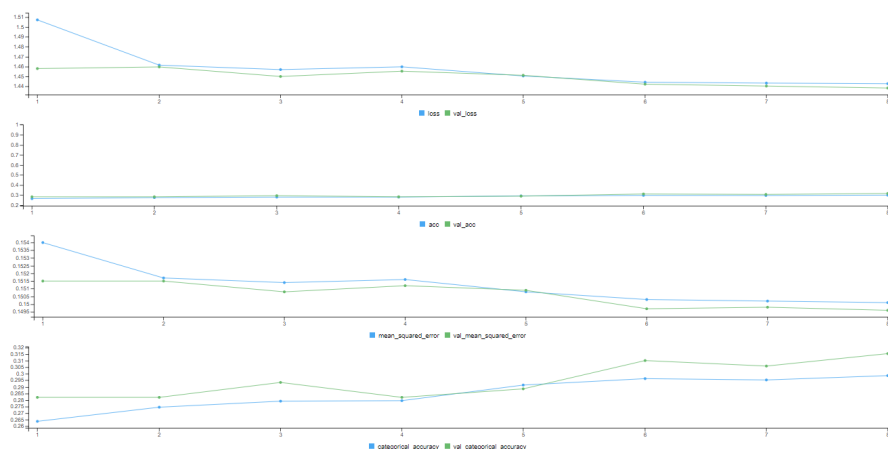


Figura 4.13: Gráficas para el resultado con VGG16

Viendo los resultados derivados de la tabla, podemos concluir:

- Cuanto mayor es el tiempo de entrenamiento (y más épocas hemos añadido), podemos mejorar ligeramente los resultados. De esta manera, hemos llegado al mejor resultado obtenido para el problema (Modelo VI con 10 épocas).

Salida para el Modelo VI con 5 épocas	Salida para el Modelo VI con 10 épocas
Tiempo de Entrenamiento: 2580 segundos	Tiempo de Entrenamiento: 4128 segundos
Train Loss: 1.4523	Train Loss: 1.4234
Train Accuracy: 0.2880	Train Accuracy: 0.3214
Validation Loss: 1.4433	Validation Loss: 1.4549
Validation Accuracy: 0.3081	Validation Accuracy: 0.3055
Test Loss: 1.446068	Test Loss: 1.439157
Test Accuracy: 0.2866807	Test Accuracy: 0.306287

- Hemos podido observar, que los resultados, sin embargo, entre un resultado y otro, difieren en un número muy pequeño. Por tanto, nuestro mejor modelo es con 10 épocas, pero, si tuviéramos restricciones de tiempo de ejecución/recursos, con un menor número de épocas alcanzaríamos resultados muy similares.

Por último, mencionar que **el uso de Dropout, nos ha permitido obtener mejores resultados**, justificando así la potencia que tienen las técnicas de regularización dentro de una red neuronal. A lo largo de la práctica, se ha ido ajustando y utilizando este valor, con el objetivo de mejorar el comportamiento del modelo. Para ver la diferencia, en la gráfica de la imagen 4.14 se puede visualizar, el comportamiento durante el entrenamiento del mejor modelo, pero eliminando en este caso cualquier tipo de regularización añadida. Podemos ver, en contraste con la imagen 4.13, que el comportamiento es mucho más inestable, y que incluso se empieza a ver lo que podría ser un principio de sobreajuste (comenzamos a obtener peores resultados fuera del modelo). En todo caso, la diferencia en los valores es mínima para concluir dicho comportamiento.

**Loss del sexto modelo: 1.439157**

#### 4.4 Resumen de los modelos

Después de realizar varios experimentos, hemos llegado a la conclusión de que bajar el error del rango (1.4-1.6) es difícil. Dichos resultados han sido obtenidos con el modelo VI con 10 épocas. A continuación, se muestra una tabla resumen de todos los experimentos y los resultados obtenidos para el conjunto de prueba:

MODELO	TEST LOSS	TEST ACCURACY
I	1.47964	0.2759242
II	1.476669	0.2974526
III	1.458036	0.2990822
IV	1.4534	0.2962
V	1.458962	0.258427
<b>VI</b>	<b>1.439157</b>	<b>0.306287</b>

Cuadro 4.1: Resumen de los modelos para el conjunto test



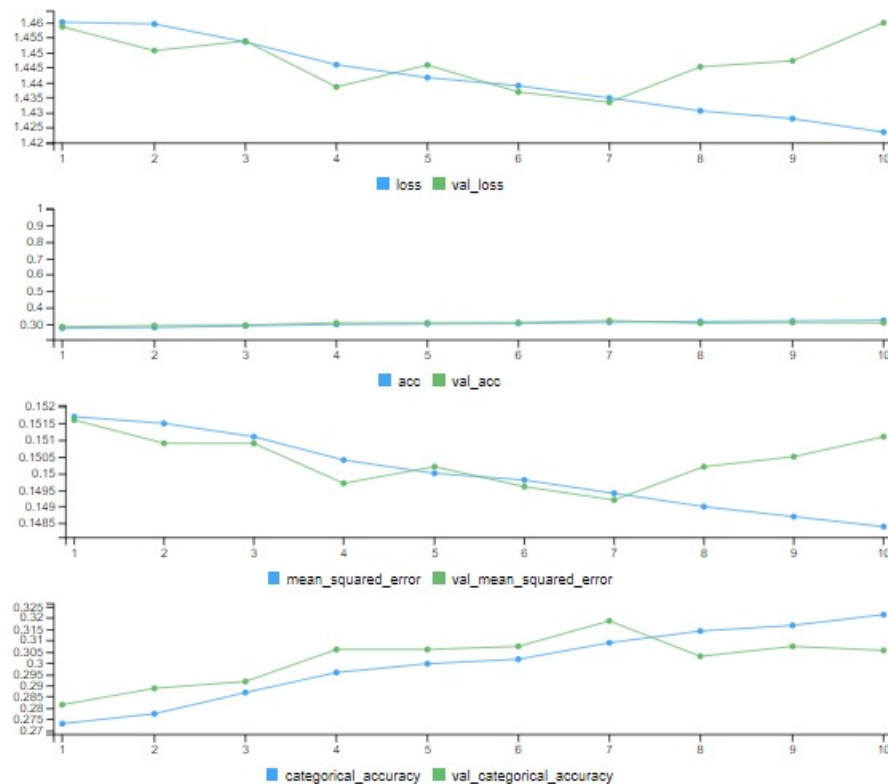


Figura 4.14: Mejor modelo (pero entrenado sin dropout)

Como se puede ver en dicha tabla, a pesar de haber añadido técnicas a nuestros modelos, con el objetivo de mejorar estos resultados, la mejora alcanzada en comparación a los modelos básicos ha sido mínima. Sin embargo, si podemos concluir, que el mejor resultado obtenido, ha sido con una Red Neuronal que se ejecuta durante 10 épocas, con Transfer Learning de VGG16, FineTuning, y técnicas de Dropout para mejorar los resultados obtenidos. Se corresponde esta configuración, con la del Modelo VI con 10 épocas.

En el siguiente apartado se van a explicar, las conclusiones obtenidas de manera resumida de lo que acabamos de comentar. Destacando, cuales son los parámetros que consideramos más importantes para el buen funcionamiento de una red neuronal convolucional.

## 5. Conclusiones

### 5.1 Conclusiones

Después de haber realizado la práctica, y partiendo de las conclusiones obtenidas en el apartado anterior, podemos destacar que:

En primer lugar mencionar, que tal y como se ha expuesto en el capítulo anterior, el mejor resultado ha sido obtenido con el Modelo VI, el cuál se basa en una Red Neuronal con Transfer Learning de la red VGG16, y Fine-Tuning para congelar únicamente las primeras capas de la red. En la tabla 5.1, se pueden ver los resultados obtenidos para este modelo.

MODELO	ÉPOCAS	BATCH	PRECISIÓN	ERROR	PÉRDIDA	TIEMPO (SEG)
<b>Modelo VI</b>	<b>10</b>	<b>100</b>	<b>0.306287</b>	<b>0.693713</b>	<b>1.439157</b>	<b>4128</b>

Cuadro 5.1: Resumen del mejor modelo para la evaluación

En primer lugar, vamos a destacar las conclusiones que están más centradas en el aspecto técnico de los entrenamientos y las técnicas utilizadas. Respecto a este aspecto, concluimos las siguientes ideas:

- En primer lugar, el tamaño de las fotos puede afectar, de alguna manera, a nuestros resultados. En nuestro caso, hemos obtenido un mejor comportamiento al utilizar tamaño de 224x224, que uno inferior. Esto se puede deber a que, con este tamaño, sea capaz de obtener mejores y más precisas características de la imagen.
- Debemos destacar la potencia de la convolución, que extrae características locales de las imágenes, las cuáles son realmente significativas en este caso. No es lo mismo tratar una imagen como un todo (lo cuál será más difícil de reconocer posteriormente con imágenes del conjunto de test), que extraer las partes más significativas, y luego usar dichas partes para identificar, posteriormente, imágenes con las mismas características. Este hecho le da una potencia a las redes neuronales que, difícilmente podrían ser igualadas por otras técnicas de Aprendizaje Automático.

- Hacer uso del algoritmo Adam como optimizador es bueno, ya que obtiene buenos resultados y de manera rápida. Dicha idea ya venía adquirida previa a la asignatura, y la hemos contrastado en nuestro caso. Tras comprobar los resultados entre los distintos optimizadores, dicho algoritmo ha sido el que mejor resultado nos ha proporcionado.
- Mencionar también que, el número de épocas que usemos será determinante en cuanto al sobreajuste de la red. No se puede concluir que al aumentar las épocas o disminuirlas estemos empeorando o mejorando la red, ya que, como se ha comentado, los valores obtenidos de Accuracy tienen tan poca diferencia que realmente no podemos distinguir si se está produciendo un verdadero sobreajuste. Sin embargo, si hemos podido ver, que con distinto número de épocas, obtenemos resultados muy similares, por lo que, para los modelos que hemos entrenado, con poco procesamiento (en comparación) podemos obtener resultados similares a los obtenidos.
- Por último mencionar, que existe una componente de aleatoriedad que no podemos controlar y debemos tener en cuenta, ya que los resultados oscilaran en torno un valor concreto, pero al haber tan poca diferencia en ellos, pueden que nos confundan. Por ello, cuando analicemos, debemos hacerlo en su totalidad, y contemplar tanto las características de la red como del problema a la hora de interpretar nuestros resultados.

Asimismo, como ya se ha comentado, realmente no se ha conseguido mejorar en profundidad, desde que comenzamos a abordar el problema hasta que llegamos a la configuración final. En nuestra opinión, derivada de todo el análisis de la práctica, se debe a las siguientes razones (principalmente):

- En primer lugar, existe un componente subjetivo muy fuerte. Ya es difícil, para una persona, determinar el rango correspondiente a una mascota, porque, no hay una solución objetiva y fundamentada que pueda justificar el adoptar antes o no a una mascota. Depende de muchas circunstancias, tanto de los gustos personales, como de la situación de esa persona, o las razones que puedan llevar a alguien a adoptar una mascota. Al haber tantas variantes no fijas, este problema se convierte en uno de muchísima mayor complejidad que el reconocimiento de imágenes, por ejemplo.
- Por otra parte, mencionar que la decisión de la adopción o no de una mascota, no se basará únicamente en una fotografía, sino que aquí entrarán muchas otras variables, como puede ser la edad o el sexo de la mascota. En el trabajo de investigación, entraremos en detalle en estos aspectos.
- Además, es importante destacar, que la relación existente entre la fotografía y la adopción no es directa, lo que dificulta aún más predecir el resultado.
- También es importante mencionar, que las técnicas de convolución son muy útiles y potentes, pero, no determinantes. Hay una gran cantidad de fotografías poco nítidas, con animales que aparecen ocultos entre manos, objetos, u otros obstáculos, lo que dificulta también el problema. Otro caso destacable es, cuando existe más de una mascota en la misma imagen, hecho que también puede confundir al clasificador y dificultar sus buenos resultados.

Por ello, para un buen resultado de este problema, habrá que combinar, tanto los resultados obtenidos a partir de las imágenes (es decir, las características extraídas gracias a las operaciones de convolución), con otra información añadida referente a la mascota, como puede ser su edad, su sexo, o incluso el anuncio con el que se intenta promover la adopción de dicho animal. Esta base, será el

punto de partida en nuestro trabajo de investigación, donde intentaremos combinar distintos tipos de información para así mejorar los resultados.

Asimismo, no debemos olvidar que el conjunto de imágenes proporcionado está muy desbalanceado, lo que dificulta la obtención de un modelo de calidad, ya que apenas existen imágenes para la clase 0, es decir, la clase correspondiente al mínimo tiempo que se tarda en adoptar una mascota,



## 6. Referencias

### Web

- [Art] Github Machine Learning for Artists. *Redes Neuronales*. URL: [https://ml4a.github.io/ml4a/es/neural\\_networks/](https://ml4a.github.io/ml4a/es/neural_networks/) (véase página 20).
- [Che] Machine Learning Cheatsheet. *Loss Functions, ML Cheatsheet Documentation*. URL: [https://ml-cheatsheet.readthedocs.io/en/latest/loss\\_functions.html](https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html) (véase página 18).
- [Clo] Google Cloud. *Advanced Guide to Inception v3 on Cloud TPU*. URL: <https://cloud.google.com/tpu/docs/inception-v3-advanced> (véase página 34).
- [Gre] Machine Learning Author for Nextjournal Gregor Koehler. *MNIST Handwritten Digit Recognition in Keras*. URL: <https://nextjournal.com/gkoehler/digit-recognition-with-keras> (véase página 23).
- [Jov] Ph.D. Jovian Lin. *Keras Models: Sequential vs. Functional*. URL: <https://jovianlin.io/keras-models-sequential-vs-functional/> (véase página 18).
- [Kera] Keras. *Keras Documentation - Activations*. URL: <https://keras.io/activations/> (véase página 20).
- [Kerb] Keras. *Keras Documentation - Model Sequential*. URL: <https://keras.io/models/sequential/> (véase página 18).
- [Kerc] Keras. *Keras: The Python Deep Learning library*. URL: <https://keras.io> (véase página 11).
- [Kerd] Keras. *Train a Keras model*. URL: <https://keras.rstudio.com/reference/fit.html> (véase página 23).
- [Meda] Medium. *Construye tu primer clasificador de Deep Learning con TensorFlow: Ejemplo de razas de perros*. URL: <https://medium.com/datos-y-ciencia/construye-tu-primer-clasificador-de-deep-learning-con-tensorflow-ejemplo-de-razas-de-perros-ed218bb4df89> (véanse páginas 21, 22).



- 
- [Medb] Medium. *Tensorflow Vs Keras? — Comparison by building a model for image classification*. URL: <https://hackernoon.com/tensorflow-vs-keras-comparison-by-building-a-model-for-image-classification-f007f336c519> (véase página 12).
- [Sci] Towards data Science. *Data Augmentation Experimentation*. URL: <https://towardsdatascience.com/data-augmentation-experimentation-3e274504f04b> (véase página 32).
- [Staa] StackOverflow. *Role of “Flatten” in Keras*. URL: <https://stackoverflow.com/questions/43237124/role-of-flatten-in-keras> (véanse páginas 21, 22).
- [Stab] StackOverflow. *What is “epoch” in keras.models.Model.fit?* URL: <https://stackoverflow.com/questions/44907377/what-is-epoch-in-keras-models-model-fit> (véase página 17).
- [Ten] TensorFlow. *An open source machine learning library for research and production*. URL: <https://www.tensorflow.org> (véase página 11).