Sistemas inteligentes para la gestión de la empresa

(SIGE)

Práctica 2: Deep Learning para multi-clasificación

Curso 2018-2019

En colaboración con:

Autor

Jonathan Martín Valera – jmv742@correo.ugr.es

Índice

1. Introducción

En esta segunda práctica de la asignatura Sistemas Inteligentes para la Gestión en la Empresa se estudiará cómo crear un modelo de clasificación de imágenes basado en redes neuronales profundas.

Para ello, a lo largo de esta documentación, se detallará el proceso seguido en el diseño e implementación de un sistema basado en redes neuronales para el problema de clasificación que se propone a continuación.

Descripción del problema

Se trabajará 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 el tiempo de adopción de una mascota, representado con un valor categórico {0, ..., 4}, a partir de datos y fotos del animal.

Los datos a utilizar se han descargado del siguiente enlace: https://s3.us-east-2.amazonaws.com/pets-adoption/petfinder-adoption-prediction.zip.

2. Fundamentos teóricos

El deep learning, o redes neuronales [1], es una de las áreas de la la inteligencia artificial (IA) de mayor relevancia y que más está creciendo en los últimos años. Desde 2012 a 2019, el número de papers y estudios publicados al respecto se ha multiplicado por diez y sus aplicaciones actuales están transformando todo tipo de sectores y ámbitos de la sociedad.

Desde luego, deep learning se utiliza mucho en todo lo relacionado con el procesamiento de imágenes. Entrenándolo de forma adecuada, es capaz de leer una matrícula o reconocer la cara de una persona. De hecho, estamos llegando a niveles de eficiencia que llegan al 99,4 % de exactitud. En algunos casos llegan a rivalizar con los seres humanos a la hora

de estudiar imágenes y descubrir determinadas características.

En este campo, **las redes convolucionales**(CNN) son un tipo de redes neuronales (NN) que **están muy bien preparadas para el procesamiento de imágenes**.

Las redes neuronales convolucionales trabajan modelando de forma consecutiva pequeñas piezas de información, y luego combinando esta información en las capas más profundas de la red. Una manera de entenderlas es que la primera capa intentará detectar los bordes y establecer patrones de detección de bordes. Luego, las capas posteriores trataran de combinarlos en formas más simples y, finalmente, en patrones de las diferentes posiciones de los objetos, iluminación, escalas, etc. Las capas finales intentarán hacer coincidir una imagen de entrada con todas los patrones y arribar a una predicción final como una suma ponderada de todos ellos. De esta forma las redes neuronales convolucionales son capaces de modelar complejas variaciones y comportamientos dando predicciones bastantes precisas.

Para más información acerca de su estructura y funcionamiento, se puede visitar el enlace recomendado en [2].

3. Clasificación

Dado que nuestro objetivo es el de poder realizar una clasificación de imágenes, se han construido una serie de modelos de redes neuronales convolucionales que realicen dichas clasificaciones. Para la construcción de estos modelos, se han usado diversas técnicas como ajustes en la topología de red, ajuste de hiperparámetros, transfer learning ....

Para cada modelo, se ha calculado su tasa de `loss` y `acc` en los conjuntos de entrenamiento, validación y test.

A continuación se van a detallar el proceso que se ha seguido para la construcción de dichos modelos, y los resultados obtenidos.

3.1. Partición de los datos entrenamiento/validación/test

Inicialmente tenemos un directorio clasificado por etiquetas que contiene las imágenes de entrenamiento que se van a usar en el modelo de clasificación. Dado que no tenemos un conjunto de validación, y **el conjunto de test proporcionado no está clasificado**, y por lo tanto, se desconocen las etiquetas de las imágenes, **se ha tenido que dividir las imágenes de entrenamiento en dos subconjuntos adicionales** para validación y test.

Para realizar dicha división, se ha hecho uso de una función que me han proporcionado unos compañeros de clase que automatiza todo este proceso de mover las imágenes a sus respectivos conjuntos. El código fuente es el siguiente:

```
# Función para dividir el conjunto etiquetado de train en otro conjunto
# etiquetado.
separaTrainTest <- function(carpeta_train, carpeta_test, porcentaje = 0.2) {
  clases<-list.dirs(path = carpeta_train, full.names = FALSE)

  for (clase in clases){
    if(clase != "") {
      carpeta_clase_train <- paste(carpeta_train,clase,sep = "/")
      carpeta_clase_test <- paste(carpeta_test,clase,sep = "/")

      todos <- list.files(path = carpeta_clase_train)
      a_copiar <- sample(todos, length(todos)*porcentaje)

      for (fichero in a_copiar){
        file.copy(paste(carpeta_clase_train, fichero, sep = "/"),
                        carpeta_clase_test)
        file.remove(paste(carpeta_clase_train, fichero, sep = "/"))
      }
    }
```

```
  }
}
# Se genera el conjunto de test a partir de las imágenes etiquetadas
# de train
separaTrainTest("./data/train_images", "./data/test_images")
```

Esta función se ha ejecutado dos veces, y se han repartido las imágenes de la siguiente forma:

- Conjunto de entrenamiento 65 %
- Conjunto validación: 15 %
- Conjunto de test: 20 %

3.2. Lectura de datos

Tras haber generado los conjuntos de validación y test, (se tienen 3 directorios que contienen subdirectorios que clasifican las imágenes por etiquetas) se ha procedido a cargar dichos conjuntos utilizando la biblioteca `Keras`. Para ello se han empleado los siguientes pasos:

- Se definen las rutas donde se ubican las imágenes de entrenamiento, validación y test.

```
train_dir      <- '/home/rstudio-user/data/train_images'
validation_dir <- '/home/rstudio-user/data/validation_images'
test_dir       <- '/home/rstudio-user/data/test_images'
```

- Se definen los generadores de la imagen, haciendo un reescalado 1/255.

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

- Se cargan las imágenes a través de un directorio clasificado por etiquetas.

```
# Se definen las imágenes de entrenamiento a través de un directorio
# clasificado por etiquetas
train_data <- flow_images_from_directory(
  directory = train_dir,
  generator = train_datagen,
  target_size = c(150, 150),      # (w, h) --> (150, 150)
  batch_size = 100,               # grupos de 100 imágenes
  class_mode = "categorical"      # etiquetas multiclase
)

# Se definen las imágenes de validación través de un directorio
# clasificado por etiquetas
validation_data <- flow_images_from_directory(
  directory = validation_dir,
  generator = validation_datagen,
  target_size = c(150, 150),      # (w, h) --> (150, 150)
  batch_size = 100,               # grupos de 100 imágenes
  class_mode = "categorical"      # etiquetas multiclase
)

# Se definen las imágenes de test través de un directorio
# clasificado por etiquetas
test_data <- flow_images_from_directory(
  directory = test_dir,
```

```
  generator = test_datagen,
  target_size = c(150, 150),    # (w, h) --> (150, 150)
  batch_size = 100,             # grupos de 100 imágenes
  class_mode = "categorical"    # etiquetas multiclase
)
```

- Por último, se han creado unas variables para especificar la dimensión de los datos y el número de clases del modelo. Estas variables serán necesarias para posteriormente definir los modelos de clasificación.

```
# Se define la dimensión de los datos
input_shape_images <- c(150,150,3)
# Se especifica el número de clases de este problema.
num_classes <- 5
```

3.3. Modelos de clasificación

Para este problema de clasificación, se ha construido una serie de modelos aplicando una serie de técnicas para evaluar su comportamiento y resultado. A continuación se van a especificar cómo se ha construido dichos modelos, clasificándolos según la técnica empleada.

3.3.1. Ajuste de topología e hiperparámetros

En primer lugar, se han construido una serie de modelos en los que se ha ido variando la topología de la red y e hiperparámetros. Por ejemplo, se han especificado diferentes capas, número de neuronas y algoritmo de optimización.

Modelo v1

El primer modelo que se ha construido, es un modelo base que se ha tomado de [3]. Dicho modelo combina una red convolutiva, con una red red neuronal profunda. La topología y

parámetros utilizados son los siguientes:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_1 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_2 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_3 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense (Dense)	(None, 512)	3211776
dense_1 (Dense)	(None, 5)	2565

Total params: 3,455,173
Trainable params: 3,455,173
Non-trainable params: 0

Figura 1: Modelo v1

El código fuente es el siguiente:

```
model <- keras_model_sequential() %>%
 layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation = "relu",
               input_shape = input_shape_images) %>%
 layer_max_pooling_2d(pool_size = c(2, 2)) %>%
 layer_conv_2d(filters = 64,  kernel_size = c(3, 3), activation = "relu")  %>%
 layer_max_pooling_2d(pool_size = c(2, 2)) %>%
 layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
 layer_max_pooling_2d(pool_size = c(2, 2)) %>%
 layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
 layer_max_pooling_2d(pool_size = c(2, 2)) %>%
 layer_flatten() %>%
 layer_dense(units = 512, activation = "relu") %>%
 layer_dense(units = num_classes, activation = "softmax")
```

El *algoritmo de optimización* utilizado ha sido `adam`, con una medida de *loss* de `categorical crossentropy` y utilizando la *métrica* de `acc`.

```
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adam",
  metrics = c("accuracy")
)
```

A continuación, se ha procedido a realizar el entrenamiento del modelo, especificando 5 épocas.

```
history <- model %>%
  fit_generator(
    train_data,
    steps_per_epoch = 100,
    epochs = 5,
    validation_data = validation_data,
    validation_steps = 50
  )
```

Tras haber entrenado el modelo, se ha evaluado su `acc` y `loss` con el conjunto de test.

```
test_rate <- model %>% evaluate_generator(test_data, steps = 5)
print(test_rate)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.282	1.450	5	adam	29.81 minutos

Tabla 1: Resultado modelo v1

***Nota:** *Dado que el código fuente para compilar, entrenar y validar el modelo siempre tiene la misma estructura (salvo cambiando algunos parámetros), se va a obviar dicho código fuente en el resto de modelos para evitar su replicación.*

Modelo v2

En este modelo, se ha modificado por completo la topología de la red del `modelo v1`. En este caso, se ha utilizado una topología mencionada en [4]. El modelo es el siguiente:

Layer (type)	Output Shape	Param #
separable_conv2d_6 (SeparableConv2D)	(None, 148, 148, 32)	155
separable_conv2d_7 (SeparableConv2D)	(None, 146, 146, 64)	2400
max_pooling2d_6 (MaxPooling2D)	(None, 73, 73, 64)	0
separable_conv2d_8 (SeparableConv2D)	(None, 71, 71, 64)	4736
separable_conv2d_9 (SeparableConv2D)	(None, 69, 69, 128)	8896
max_pooling2d_7 (MaxPooling2D)	(None, 34, 34, 128)	0
separable_conv2d_10 (SeparableConv2D)	(None, 32, 32, 64)	9408
separable_conv2d_11 (SeparableConv2D)	(None, 30, 30, 128)	8896
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 128)	0
dense_3 (Dense)	(None, 32)	4128
dense_4 (Dense)	(None, 5)	165

Total params: 38,784
Trainable params: 38,784
Non-trainable params: 0

Figura 2: Modelo v2

Como se puede observar en la topología de este modelo, y en comparación con el anterior, se han añadido varias capas convolutivas adicionales con un número de neuronas comprendido entre 32 y 128 para la parte convolutiva y profunda.

```
model_v2 <- keras_model_sequential() %>%
  layer_separable_conv_2d(filters = 32, kernel_size = 3,
                          activation = "relu",
                          input_shape = input_shape_images) %>%
  layer_separable_conv_2d(filters = 64, kernel_size = 3,
                          activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_separable_conv_2d(filters = 64, kernel_size = 3,
                          activation = "relu") %>%
  layer_separable_conv_2d(filters = 128, kernel_size = 3,
                          activation = "relu") %>%
```

```
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_separable_conv_2d(filters = 64, kernel_size = 3,
                          activation = "relu") %>%
  layer_separable_conv_2d(filters = 128, kernel_size = 3,
                          activation = "relu") %>%
  layer_global_average_pooling_2d() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.284	1.450	5	rmsprop	41.11 minutos

Tabla 2: Resultado modelo v2

Modelo v3

En este nuevo modelo, se ha usado el `modelo v2`, modificando el número de neuronas de la parte profunda y añadiendo nuevas capas. El modelo es el siguiente:

Layer (type)	Output Shape	Param #
separable_conv2d_12 (SeparableConv2D)	(None, 148, 148, 32)	155
separable_conv2d_13 (SeparableConv2D)	(None, 146, 146, 64)	2400
max_pooling2d_8 (MaxPooling2D)	(None, 73, 73, 64)	0
separable_conv2d_14 (SeparableConv2D)	(None, 71, 71, 64)	4736
separable_conv2d_15 (SeparableConv2D)	(None, 69, 69, 128)	8896
max_pooling2d_9 (MaxPooling2D)	(None, 34, 34, 128)	0
separable_conv2d_16 (SeparableConv2D)	(None, 32, 32, 64)	9408
separable_conv2d_17 (SeparableConv2D)	(None, 30, 30, 128)	8896
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 128)	0
dense_5 (Dense)	(None, 256)	33024
dense_6 (Dense)	(None, 128)	32896
dense_7 (Dense)	(None, 5)	645

Total params: 101,056
Trainable params: 101,056
Non-trainable params: 0

Figura 3: Modelo v3

```
model_v3 <- keras_model_sequential() %>%
  layer_separable_conv_2d(filters = 32, kernel_size = 3,
                          activation = "relu",
                          input_shape = input_shape_images) %>%
  layer_separable_conv_2d(filters = 64, kernel_size = 3,
                          activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_separable_conv_2d(filters = 64, kernel_size = 3,
                          activation = "relu") %>%
  layer_separable_conv_2d(filters = 128, kernel_size = 3,
                          activation = "relu") %>%
  layer_max_pooling_2d(pool_size = 2) %>%
  layer_separable_conv_2d(filters = 64, kernel_size = 3,
                          activation = "relu") %>%
  layer_separable_conv_2d(filters = 128, kernel_size = 3,
                          activation = "relu") %>%
  layer_global_average_pooling_2d() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.281	1.459	5	adam	41.15 minutos

Tabla 3: Resultado modelo v3

Modelo v4

En este nuevo modelo, se ha definido una serie de capas convolutivas en orden creciente de neuronas: 32,64,128 y 256 con un kernel size de 5 (a diferencia de los anteriores que su valor era 3), y una única capa densa de 512 neuronas. El modelo es el siguiente:

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 146, 146, 32)	2432
max_pooling2d_17 (MaxPooling2D)	(None, 73, 73, 32)	0
conv2d_13 (Conv2D)	(None, 69, 69, 64)	51264
max_pooling2d_18 (MaxPooling2D)	(None, 34, 34, 64)	0
conv2d_14 (Conv2D)	(None, 30, 30, 128)	204928
max_pooling2d_19 (MaxPooling2D)	(None, 15, 15, 128)	0
conv2d_15 (Conv2D)	(None, 11, 11, 256)	819456
max_pooling2d_20 (MaxPooling2D)	(None, 5, 5, 256)	0
flatten_1 (Flatten)	(None, 6400)	0
dense_8 (Dense)	(None, 512)	3277312
dense_9 (Dense)	(None, 5)	2565

Total params: 4,357,957
Trainable params: 4,357,957
Non-trainable params: 0

Figura 4: Modelo v4

```
model_v4 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation = "relu",
                input_shape = input_shape_images) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64,  kernel_size = c(5, 5), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(5, 5), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 256, kernel_size = c(5, 5), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.281	1.457	5	adam	30.21 minutos

Tabla 4: Resultado modelo v4

Modelo v5

Modelo similar al `modelo v4`, especificando un kernel size de tamaño 3 menos en la última capa que es de tamaño 5. Respecto a la parte profunda, se han añadido 3 capas adicionales. El modelo es el siguiente:

Layer (type)	Output Shape	Param #
conv2d_132 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_124 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_133 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_125 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_134 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_126 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_135 (Conv2D)	(None, 15, 15, 256)	295168
max_pooling2d_127 (MaxPooling2D)	(None, 7, 7, 256)	0
conv2d_136 (Conv2D)	(None, 3, 3, 512)	3277312
max_pooling2d_128 (MaxPooling2D)	(None, 1, 1, 512)	0
flatten_10 (Flatten)	(None, 512)	0
dense_33 (Dense)	(None, 128)	65664
dense_34 (Dense)	(None, 256)	33024
dense_35 (Dense)	(None, 512)	131584
dense_36 (Dense)	(None, 128)	65664
dense_37 (Dense)	(None, 5)	645

Total params: 3,962,309
Trainable params: 3,962,309
Non-trainable params: 0

Figura 5: Modelo v5

```
model_v5 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation = "relu",
                input_shape = input_shape_images) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64,  kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
```

```
  layer_conv_2d(filters = 512, kernel_size = c(5, 5), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.286	1.451	5	adam	29.71 minutos

Tabla 5: Resultado modelo v5

Conclusiones

Los resultados obtenidos modificando la topología de la red y sus hiperparámetros son los siguientes:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v1	0.282	1.450	5	adam	29.81 minutos
Modelo v2	0.284	1.450	5	rmsprop	41.11 minutos
Modelo v3	0.281	1.459	5	adam	41.15 minutos
Modelo v4	0.281	1.457	5	adam	30.21 minutos
Modelo v5	0.286	1.451	5	adam	29.71 minutos

Tabla 6: Comparación modelos modificando topología e hiperparámetros

Como se puede observar, no ha habido apenas cambios en el `acc`, de hecho todos se sitúan en el *0.28* `acc`. Cabe a destacar la diferencia en algunos casos del tiempo de entrenamiento empleado.

3.3.2. Fine tuning

En esta sección, se va a aplicar la técnica de `fine tuning`, por la cual se va a coger redes convolutivas ya entrenadas para problemas de clasificación de imágenes, y se va a unir a una red profunda que se definirá para nuestro problema. Dado que se está utilizando `keras`, se ha utilizado algunos modelos entrenados disponibles, que están definidos en https://keras.io/applications/.

También se mostrará la diferencia que hay entre: si se "congelan"los pesos de la red convolutiva ya entrenada, o por el contrario, modificamos dichos pesos al realizar back-propagation.

Modelo v6

En este modelo se va a utilizar la red `application_vgg16` como parte convolutiva, y se va a congelar los pesos para que no se vean modificados por el backpropagation. El modelo es el siguiente:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1048704
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 645
=================================================================
Total params: 15,764,037
Trainable params: 1,049,349
Non-trainable params: 14,714,688
_________________________________________________________________
```

Figura 6: Modelo v6

Para construir dicho modelo, se ha realizado lo siguiente:

```
# Se define la parte convolutiva
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
```

```
)


# Se congelan los pesos
freeze_weights(conv_base)


# Se define el modelo compuesto
model_v6 <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

> ***Nota:** La fase de compilación, entrenamiento y validación es igual que en el resto de modelos. Ver sección 3.3.1*

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.310	1.461	5	adam	35.65 minutos

Tabla 7: Resultado modelo v6

Modelo v7

En este modelo se va a utilizar el mismo `modelo v6`, pero en este caso no se va a congelar los pesos. A continuación veremos el resultado que esto implica. El modelo es el siguiente:

```
Layer (type)                 Output Shape              Param #
================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
________________________________________________________________
dense_2 (Dense)              (None, 128)               1048704
________________________________________________________________
dense_3 (Dense)              (None, 5)                 645
================================================================
Total params: 15,764,037
Trainable params: 15,764,037
Non-trainable params: 0
________________________________________________________________
```

Figura 7: Modelo v7

```
conv_base_v7 <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
)

model_v7 <- keras_model_sequential() %>%
  conv_base_v7 %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.238	12.27	5	adam	51.86 minutos

Tabla 8: Resultado modelo v7

Tal y como se puede observar en este resultado, el hecho de modificar los propios pesos de la red que hemos cogido entrenada ha sido desastroso. Como consecuencia ha aumentado mucho el `loss`, el `acc` es el mínimo obtenido en todos lo modelos y con un tiempo de entrenamiento bastante elevado ($\sim$ 52 minutos).

Modelo v8

En este caso, se va a utilizar el modelo `application_inception_v3`, utilizando la misma topología e hiperparámetros de la red profunda que en el `modelo v6`. El modelo es el siguiente:

```
conv_base_v8 <- application_inception_v3(
  weights = "imagenet",
  include_top = FALSE,
```

Layer (type)	Output Shape	Param #
inception_v3 (Model)	(None, 3, 3, 2048)	21802784
flatten_2 (Flatten)	(None, 18432)	0
dense_4 (Dense)	(None, 128)	2359424
dense_5 (Dense)	(None, 5)	645

Total params: 24,162,853
Trainable params: 2,360,069
Non-trainable params: 21,802,784

Figura 8: Modelo v8

```
  input_shape = c(150, 150, 3)
)


model_v8 <- keras_model_sequential() %>%
  conv_base_v8 %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

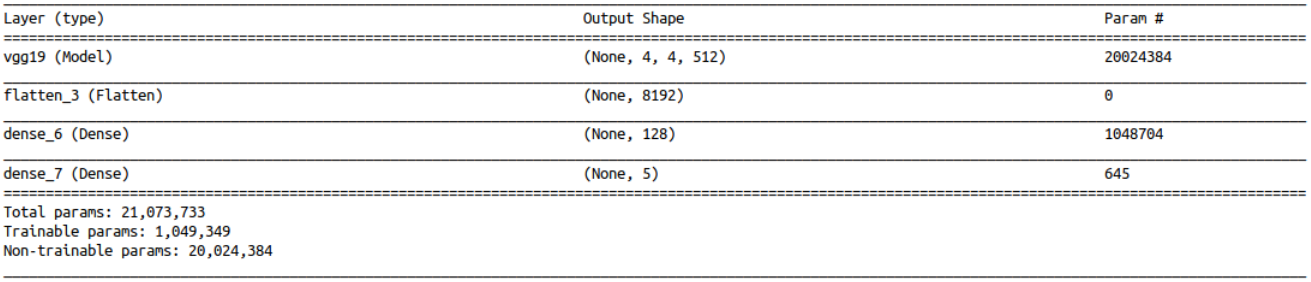
Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.307	1.462	5	adam	32.7 minutos

Tabla 9: Resultado modelo v8

Modelo v9

En este caso, se va a utilizar el modelo `application_vgg19`, utilizando la misma topología e hiperparámetros de la red profunda que en el `modelo v6`. El modelo es el siguiente:

```
conv_base_v9 <- application_vgg19(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
```



Layer (type)	Output Shape	Param #
vgg19 (Model)	(None, 4, 4, 512)	20024384
flatten_3 (Flatten)	(None, 8192)	0
dense_6 (Dense)	(None, 128)	1048704
dense_7 (Dense)	(None, 5)	645

Total params: 21,073,733
Trainable params: 1,049,349
Non-trainable params: 20,024,384

Figura 9: Modelo v9

```
model_v9 <- keras_model_sequential() %>%
conv_base_v9 %>%
layer_flatten() %>%
layer_dense(units = 128, activation = "relu") %>%
layer_dense(units = num_classes, activation = "softmax")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.304	1.447	5	adam	35.3 minutos

Tabla 10: Resultado modelo v9

Modelo v10

En este caso, se va a utilizar el modelo `application_inception_resnet_v2`, utilizando la misma topología e hiperparámetros de la red profunda que en el `modelo v6`. El modelo es el siguiente:

```
conv_base_v10 <- application_inception_resnet_v2(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(150, 150, 3)
)
```

Layer (type)	Output Shape	Param #
inception_resnet_v2 (Model)	(None, 3, 3, 1536)	54336736
flatten_4 (Flatten)	(None, 13824)	0
dense_8 (Dense)	(None, 128)	1769600
dense_9 (Dense)	(None, 5)	645

Total params: 56,106,981
Trainable params: 1,770,245
Non-trainable params: 54,336,736

Figura 10: Modelo v10

```
model_v10 <- keras_model_sequential() %>%
  conv_base_v10 %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.302	1.462	5	adam	41.5 minutos

Tabla 11: Resultado modelo v10

Conclusiones

Los resultados obtenidos al hacer `fine tuning` han sido mejores. Se ha observado que utilizar un modelo ya entrenado mejora la predicción en este problema. También se ha visto la importancia de congelar los pesos de la red entrenada, ya que por lo contrario, la red empeora con mucha diferencia.

Respecto a los modelos de la sección anterior, estos han dado mejor resultado, entre 0.30 y 0.31 `acc`.

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v6	0.310	1.461	5	adam	35.65 minutos
Modelo v7	0.238	12.27	5	adam	51.86 minutos
Modelo v8	0.307	1.462	5	adam	32.7 minutos
Modelo v9	0.304	1.447	5	adam	35.3 minutos
Modelo v10	0.302	1.462	5	adam	41.5 minutos

Tabla 12: Comparación modelos usando fine tuning

3.3.3. Data augmentation

En esta sección, se va a aplicar la técnica de `data agumentation`, por la cual se van a aplicar una serie de transformaciones a las imágenes de entrenamiento, con el objetivo de aumentar el número de imágenes que el modelo procesará.

Modelo v11

En este caso, vamos a observar el resultado que se obtiene al aplicar `data agumentation` en un modelo ya construido. Para esto, se ha seleccionado el `modelo v1`, cuyo estructura era la siguiente:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_1 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_2 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_3 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense (Dense)	(None, 512)	3211776
dense_1 (Dense)	(None, 5)	2565

Total params: 3,455,173
Trainable params: 3,455,173
Non-trainable params: 0

Figura 11: Modelo v11

Para generar las transformaciones de las imágenes se ha definido un generador de imágenes al cual se le han atribuido unas serie de atributos para realizar las transformaciones a las imágenes de entrenamiento. La mayoría de estos atributos tienen un valor de 0.2 para

que la red no se sobrecargue de imágenes parecidas.

```
data_augmentation_datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 40,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)
```

A continuación se ha definido un conjunto de entrenamiento, utilizando las imágenes de train, pero indicando que el nuevo generador es el que contiene las definiciones de las transformaciones.

```
train_augmented_data <- flow_images_from_directory(
  directory = train_dir,
  generator = data_augmentation_datagen,  # ¡usando nuevo datagen!
  target_size = c(150, 150),              # (w, h) --> (150, 150)
  batch_size = 20,                        # grupos de 20 imágenes
  class_mode = "categorical"              # etiquetas multiclase
)
```

Finalmente, se realiza el entrenamiento utilizando este nuevo conjunto de entrenamiento.

```
history <- model %>%
  fit_generator(
    train_augmented_data,
    steps_per_epoch = 100,
    epochs = 5,
```

```
    validation_data = validation_data,
    validation_steps = 50
  )
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.281	1.456	5	adam	58.75 minutos

Tabla 13: Resultado modelo v11

Modelo v12

Para construir este modelo y aplicar `data agumentation`, se va a utilizar el `modelo v6`, ya que es el mejor que se ha obtenido hasta el momento, y se observará si al aplicar esta técnica, el resultado mejorará o empeorará.

El `modelo v6` es el siguiente:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 128)               1048704
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 645
=================================================================
Total params: 15,764,037
Trainable params: 1,049,349
Non-trainable params: 14,714,688
_________________________________________________________________
```

Figura 12: Modelo v12

Al igual que en el `modelo v11`, se ha realizado el mismo procedimiento para realizar las transformaciones a las imágenes y realizar el entrenamiento.

```
history <- model_v6 %>%
  fit_generator(
    train_augmented_data,
    steps_per_epoch = 100,
```

```
    epochs = 5,
    validation_data = validation_data,
    validation_steps = 50
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.281	1.457	5	adam	64.33 minutos

Tabla 14: Resultado modelo v12

Conclusiones

Los resultados obtenidos **al aplicar** este técnica son:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v1	0.281	1.456	5	adam	58.75 minutos
Modelo v6	0.281	1.457	5	adam	64.33 minutos

Los resultados obtenidos **antes de aplicar** esta técnica son:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v1	0.282	1.450	5	adam	29.81 minutos
Modelo v6	0.310	1461	5	adam	35.65 minutos

En este caso, aplicar la técnica `data augmentation` **no ha sido efectivo para nuestro problema**, ya que ha empeorado en resultados y en tiempo de entrenamiento.

Un motivo por el cual esta técnica no ha funcionado en este caso, puede ser que esta técnica es efectiva cuando las imágenes muestran objetos o cosas que son diferentes, es decir, en este caso, al tener solo perros o gatos, y tener en cierta medida la misma forma (a diferencia de un avión y una silla por ejemplo) el hecho de tener más imágenes no es importante.

3.3.4. Algoritmos de optimización

En esta sección, se va a utilizar una serie de algoritmos de optimización en el mejor modelo que se ha obtenido hasta el momento (`modelo v6`) y se va a observar su comportamiento.

Dado que se ha usado el `modelo v6` en todos los modelos de esta sección, se va a omitir su estructura, ya que ha sido mostrada anteriormente (ver sección 3.3.2).

Modelo v13

En este caso, se ha utilizado el algoritmo de optimización `SGD` como optimizador sobre el mejor modelo obtenido hasta el momento (`modelo v6`).

```
model_v13 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "sgd",
  metrics = c("accuracy")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.285	1.458	5	sgd	34.41 minutos

Tabla 15: Resultado modelo v13

Modelo v14

En este caso, se ha utilizado el algoritmo de optimización `rmsprop` como optimizador sobre el mejor modelo obtenido hasta el momento (`modelo v6`).

```
model_v14 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "rmsprop",
```

```
  metrics = c("accuracy")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.318	1.446	5	rmsprop	34.11 minutos

Tabla 16: Resultado modelo v14

Modelo v15

En este caso, se ha utilizado el algoritmo de optimización `adagrad` como optimizador sobre el mejor modelo obtenido hasta el momento (`modelo v6`).

```
model_v15%>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adagrad",
  metrics = c("accuracy")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.281	11.58	5	adagrad	34.1 minutos

Tabla 17: Resultado modelo v15

Modelo v16

En este caso, se ha utilizado el algoritmo de optimización `adadelta` como optimizador sobre el mejor modelo obtenido hasta el momento (`modelo v6`).

```
model_v16 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adadelta",
  metrics = c("accuracy")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.314	1.437	5	adadelta	34.45 minutos

Tabla 18: Resultado modelo v16

Modelo v17

En este caso, se ha utilizado el algoritmo de optimización `adamax` como optimizador sobre el mejor modelo obtenido hasta el momento (`modelo v6`).

```
model_v17 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adamax",
  metrics = c("accuracy")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.323	1.445	5	adamax	34.11 minutos

Tabla 19: Resultado modelo v17

Modelo v18

En este caso, se ha utilizado el algoritmo de optimización `nadam` como optimizador sobre el mejor modelo obtenido hasta el momento (`modelo v6`).

```
model_v18 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "nadam",
  metrics = c("accuracy")
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.307	1.475	5	nadam	34.25 minutos

Tabla 20: Resultado modelo v18

Conclusiones

Los resultados obtenidos modificando el algoritmo de optimización son los siguientes:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v13	0.285	1.458	5	sgd	34.41 minutos
Modelo v14	0.318	1.446	5	rmsprop	34.11 minutos
Modelo v15	0.281	11.58	5	adagrad	34.1 minutos
Modelo v16	0.314	1.437	5	adadelta	34.45 minutos
Modelo v17	0.323	1.445	5	adamax	34.11 minutos
Modelo v18	0.307	1.475	5	nadam	34.25 minutos

Tabla 21: Comparación modelos modificando algoritmo de optimización

Como se puede apreciar en la tabla 21, en general se han obtenido resultados dispersos aun utilizando la misma topología e hiperparámetros de la red. En este caso, podemos observar que los tiempos de entrenamiento son casi idénticos, y que **el mejor resultado se ha obtenido utilizando el algoritmo de optimización `adamax` en el `modelo v17`.**

3.3.5. Early stopping

Otra de las técnicas que se han intentado aplicar a este problema ha sido el del `early stopping`. Esta técnica ha sido usada para identificar el número de época a partir del cual un modelo empieza a sobreaprender.

En este caso, dicha comprobación no se ha hecho programáticamente, sino que se ha realizado de forma manual a través de la visualización de la tasa de `validation_loss` conforme el número de épocas se va incrementando.

Para poder identificar cuando la red empieza a sobreaprender, nos vamos a fijar en que inicialmente la tasa de `validation_loss` irá decreciendo (esto significa que el modelo va mejorando y generalizando mejor), hasta que llegue un punto en el que `validation_loss` irá subiendo poco a poco, a pesar de que el `loss` y `acc` vayan mejorando. Este incremento de textitvalidation_loss significa que el modelo está perdiendo capacidad de generalización, y por lo tanto, estará realizando sobreaprendizaje.

A continuación se va a mostrar el proceso llevado a cabo para intentar detectar este tipo de situaciones.

Modelo v19

Para construir este modelo, se ha partido de la misma topología e hiperparámetros que el `modelo v17` (que es el mejor obtenido hasta el momento). Para intentar mejorar este modelo, se va a entrenar con 15 épocas, se observará la gráfica de evolución del `validation_loss` y se detectará cual puede ser un buen número de épocas para entrenar dicho modelo. El modelo es el siguiente:

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 128)	1048704
dense_1 (Dense)	(None, 5)	645

Total params: 15,764,037
Trainable params: 1,049,349
Non-trainable params: 14,714,688

Figura 13: Modelo v19

```
model_v19 <- keras_model_sequential() %>%
  conv_base_v19 %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

Se establece el número de épocas a 15 para entrenar la red.

```
history <- model_v19 %>%
  fit_generator(
    train_data,
    steps_per_epoch = 100,
    epochs = 15,
    validation_data = validation_data,
    validation_steps = 50
  )
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.308	1.58	15	adamax	95.55 minutos

Tabla 22: Resultado modelo v19

El resultado que se obtuvo para 5 épocas fue:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.323	1.445	5	adamax	34.11 minutos

Tabla 23: Resultado modelo v17 para 5 épocas

Tal y como se observan en las anteriores tablas, el resultado obtenido con un entrenamiento con 5 épocas es mejor respecto al modelo entrenado con 15. La causa de esto puede ser debido al sobreaprendizaje.

A continuación se visualiza la gráfica del entrenamiento del modelo para observar su comportamiento.

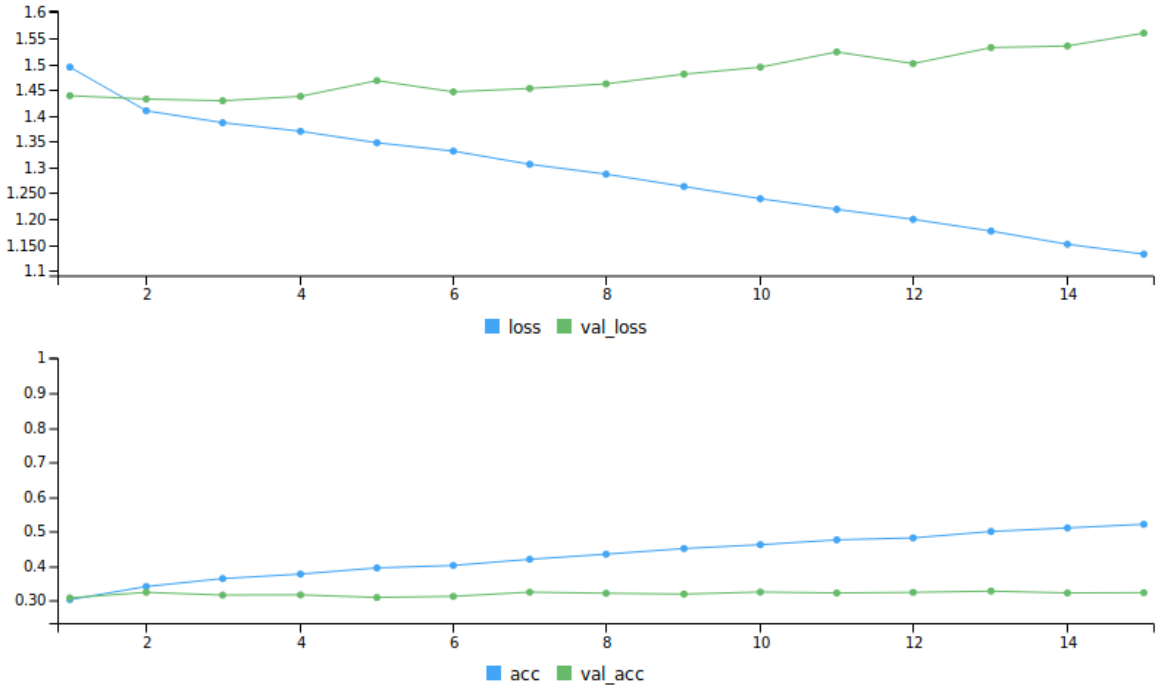


Figura 14: Gráfica del entrenamiento del modelo v19

Tal y como se observa en la figura 14, a partir de la época 2 se tiene que el `validation_loss` empieza a crecer y el `val_acc` empieza a decrecer.

Modelo v20

Para intentar resolver el sobreaprendizaje que se produce en el `modelo v19`, en este modelo se va a replicar la misma topología e hiperparámetros que el `modelo v19`, estableciendo 2 épocas de entrenamiento.

```
history <- model_v20 %>%
  fit_generator(
    train_data,
    steps_per_epoch = 100,
```

```
    epochs = 2,
    validation_data = validation_data,
    validation_steps = 50
)
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.318	1.426	2	adamax	13.65 minutos

Tabla 24: Resultado modelo v20

Conclusiones

Los resultados son los siguientes:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v19	0.308	1.58	15	adamax	95.55 minutos
Modelo v20	0.318	1.426	2	adamax	13.65 minutos

Tabla 25: Comparación modelos aplicando early stopping

Tal y como se puede observar en la tabla 25, aplicando `early_stopping` se ha conseguido un buen modelo en un tiempo de entrenamiento muy razonable. De hecho, si se compara con el modelo entrenado con 15 épocas, mejora en todos los aspectos.

Como aspecto a destacar, con este modelo se ha obtenido la tasa de `test_loss` más baja de todos los modelos, pero aun así, no ha superado (por muy poco) la tasa de `test_acc` del mejor modelo obtenido hasta ahora.

Tal y como pasaba con la técnica de `data_augmentation`, este problema al tener un gran tamaño, no se puede especificar un número de épocas alto, por lo que los resultados no son tan apreciables como lo sería en un problema más pequeño que se entrenara con un mayor número de épocas.

3.3.6. Dropout

Por último, otra técnica que se ha aplicado para intentar mejorar los modelos ha sido `dropout`.

Para ello, se ha hecho uso del `modelo v17` (mejor modelo hasta el momento) y se han añadido una o varias capas de `dropout`. El principal objetivo que se tiene al aplicar esta técnica, es el de evitar que las neuronas aprendan una característica en concreto, es decir, que se especialicen en un determinado patrón. Lo que se quiere es que el modelo sea lo más genérico posible, por lo tanto, `dropout` desactivará aleatoriamente en cada iteración una serie de neuronas con el fin de que el modelo generalice y aprenda mejor.

Observemos si para este problema en concreto, se puede mejorar el modelo aplicando dicha técnica.

Modelo v21

Partiendo del `modelo v17`, se ha construido este modelo añadiendo una capa `dropout` a la red profunda con una tasa de activación de 0.4 (40 %).

El modelo resultante es el siguiente:

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dropout (Dropout)	(None, 8192)	0
dense_2 (Dense)	(None, 128)	1048704
dense_3 (Dense)	(None, 5)	645

Total params: 15,764,037
Trainable params: 1,049,349
Non-trainable params: 14,714,688

Figura 15: Modelo v21

```
model_v21 <- keras_model_sequential() %>%
  conv_base_v21 %>%
  layer_flatten() %>%
  layer_dropout(rate=0.4) %>%
  layer_dense(units = 128, activation = "relu") %>%
```

```
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.313	1.427	5	adamax	34.4 minutos

Tabla 26: Resultado modelo v21

Modelo v22

En este caso, se ha partido del `modelo v17` y se ha modificado la parte de red profunda. Se han añadido varias capas *dense* con otras capas `dropout` enmedio.

El modelo es el siguiente:

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_2 (Flatten)	(None, 8192)	0
dropout_1 (Dropout)	(None, 8192)	0
dense_4 (Dense)	(None, 256)	2097408
dropout_2 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 128)	32896
dense_6 (Dense)	(None, 5)	645

Total params: 16,845,637
Trainable params: 2,130,949
Non-trainable params: 14,714,688

Figura 16: Modelo v22

```
model_v22 <- keras_model_sequential() %>%
  conv_base_v22 %>%
  layer_flatten() %>%
  layer_dropout(rate=0.4) %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate=0.5) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")
```

El resultado obtenido es el siguiente:

Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
0.308	1.431	5	adamax	33.21 minutos

Tabla 27: Resultado modelo v22

Conclusiones

Los resultados obtenidos son los siguientes:

- Aplicando dropout (`modelo v21` vs sin dropout `modelo v17`)

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v17	0.323	1.445	5	adamax	34.11 minutos
Modelo v21	0.313	1.427	5	adamax	34.4 minutos

- Modelo v22 (aplicando dropout)vs mejor modelo `modelo v17`)

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v17	0.323	1.445	5	adamax	34.11 minutos
Modelo v22	0.308	1.431	5	adamax	33.21 minutos

Tal y como se puede observar en las tablas anteriores, se ha conseguido reducir el `test_loss` en ambos modelos aplicando `dropout`, esto significa que se ha conseguido que el modelo generalice mejor, pero aun así, no se ha obtenido un mayor `acc` que en el `modelo v17`.

Aunque la técnica de `dropout` suele dar muy buen resultado como técnica de regularización en deep learning, en este caso no ha habido ningún cambio sustancial a la hora de mejorar nuestro modelo.

4. Discusión de resultados

Los resultados obtenidos con los modelos construidos son:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v1	0.282	1.450	5	adam	29.81 minutos
Modelo v2	0.284	1.450	5	rmsprop	41.11 minutos
Modelo v3	0.281	1.459	5	adam	41.15 minutos
Modelo v4	0.281	1.457	5	adam	30.21 minutos
Modelo v5	0.286	1.451	5	adam	29.71 minutos
Modelo v6	0.310	1.461	5	adam	35.65 minutos
Modelo v7	0.238	12.27	5	adam	51.86 minutos
Modelo v8	0.307	1.462	5	adam	32.7 minutos
Modelo v9	0.304	1.447	5	adam	35.3 minutos
Modelo v10	0.302	1.462	5	adam	41.5 minutos
Modelo v11	0.281	1.456	5	adam	58.75 minutos
Modelo v12	0.281	1.457	5	adam	64.33 minutos
Modelo v13	0.285	1.458	5	sgd	34.41 minutos
Modelo v14	0.318	1.446	5	rmsprop	34.11 minutos
Modelo v15	0.281	11.58	5	adagrad	34.1 minutos
Modelo v16	0.314	1.437	5	adadelta	34.45 minutos
Modelo v17	0.323	1.445	5	adamax	34.11 minutos
Modelo v18	0.307	1.475	5	nadam	34.25 minutos
Modelo v19	0.308	1.58	15	adamax	95.55 minutos
Modelo v20	0.318	1.426	2	adamax	13.65 minutos
Modelo v21	0.313	1.427	5	adamax	34.4 minutos
Modelo v22	0.308	1.431	5	adamax	33.21 minutos

Tabla 28: Resultados de los modelos

Haciendo un balance total de los modelos, vemos que en líneas generales se han obtenidos valores de `acc` comprendidos entre 0.28 y 0.32.

Si hacemos un resumen de los 3 mejores modelos que se han obtenido, tenemos lo siguiente:

	Test_acc	Test_loss	Nº épocas	Algoritmo	Tiempo entrenamiento
Modelo v17	0.323	1.445	5	adamax	34.11 minutos
Modelo v20	0.318	1.426	2	adamax	13.65 minutos
Modelo v14	0.318	1.446	5	rmsprop	34.11 minutos

Tabla 29: Mejores modelos ordenados por acc

- En primer lugar tenemos el `modelo v17` que fue obtenido a través de realizar *fine tuning* con la red `application_vgg16` (*importante: y congelando los pesos*), añadiendo una red profunda simple y aplicando el algoritmo de optimización `adamax`. Con este **modelo se ha obtenido el mejor porcentaje de acierto con un 32.3 %**.
- En segundo lugar tenemos el `modelo v20` que fue obtenido a través de realizar *early stopping* sobre el `modelo v17`. Este ha sido el modelo que menor tasa de `loss` ha dado con un total de 1.426, y con un valor de `acc` muy parecido al `modelo v17`. Sin duda, este puede ser el mejor modelo considerando la relación resultados/tiempoEntrenamiento.
- En tercer lugar tenemos el `modelo v14`, también generado a partir de aplicar *fine tuning* (*y congelando los pesos*), más una red profunda entrenada con el algoritmo de optimización `rmsprop`.

Aparantemente pueden resultar valores muy bajos, pero debido a la complejidad de este problema es normal obtener resultados como estos. A simple vista parece difícil poder clasificar una imagen de un animal para poder predecir cuanto tiempo pasarán para que lo adopten, y es que ni nosotros mismos somos capaces de realizar ese proceso a simple vista.

5. Conclusiones

Como conclusiones generales que se han obtenido al realizar esta práctica, se tiene que:

- Al construir la red convolutiva y profunda, modificando su topología e hiperparámetros, se han obtenido resultados muy similares entre ellos. De hecho el **añadir más capas o neuronas a la red no significa que ésta mejore**, sino que por el contrario puede empeorar en resultados y tiempo de entrenamiento.
- Utilizar modelos de clasificación de imágenes ya entrenados y aplicando **fine tuning** congelando los pesos, **ha sido efectivo en este caso**, ya que se han obtenido mejores modelos. De hecho, se ha obtenido mejores resultados que creando redes convolutivas propias.
- Se ha observado como al aplicar distintos algoritmos de optimización para entrenar un mismo modelo ha tenido cierta repercusión en los valores de `acc` y `loss`. En este caso, el mejor resultado se ha obtenido aplicando el algoritmo `adamax`.
- Se ha concluido que **para este problema de clasificación en concreto, aplicar la técnica de `data augmentation` no tiene efecto**, ya que como se ha comentado en la sección 3.3.3, en este problema de clasificación las imágenes son muy parecidas entre sí, y el hecho de tener más imágenes para entrenar no significa que vaya a mejorar el modelo, como por ejemplo lo haría en el caso de clasificar objetos diferenciados como sillas, aviones ...
- También, se ha comprobado que **al aplicar la técnica `early stopping` se ha conseguido obtener un modelo con la mínima tasa de `loss` y con un tiempo de entrenamiento inferior al resto**. Sin embargo, no ha conseguido obtener el mejor valor de `acc` (aunque por muy poco). Tal y como se comentó en la sección 3.3.5, en este problema no se ha entrenado el modelo con un número alto de épocas (debido al tamaño de los datos y su consecuente tiempo de entrenamiento), por lo que no se ha podido observar grandes cambios al aplicar dicha técnica, aunque se ha observado como un modelo entrenado con pocas épocas (p.e 2 épocas) ha sido mejor que otro modelo entrenado con un número mayor de épocas (p.e 15 épocas).

- Al aplicar la técnica de `dropout` **no se ha obtenido ninguna mejoría significativa en este modelo**. Esta técnica suele dar buen resultado en la mayoría de modelos, pero en este caso, tal y como se ha comentado en la sección 3.3.6 ha conseguido reducir el valor de `loss` respecto a la no aplicación de `dropout`, pero sin embargo, el valor de `acc` ha disminuido.

- Por último, como resultado final de esta práctica, se ha obtenido un valor de `acc`: 0.323 y un valor de `loss`:1.445. Aparentemente este resultado puede ser muy bajo, pero tal y como se comentó en la sección 4, es normal debido a que el un problema de clasificación complejo que incluso hasta a nosotros mismos nos resultaría difícil hacerlo en la vida real.

Referencias

[1] Digital Biz, Deep Learning y redes neuronales, 31 mayo, 2018, disponible en https://www.digitalbizmagazine.com/deep-learning-y-redes-neuronales/

[2] Raul E. Lopez Briega, Redes neuronales convolucionales con TensorFlow, 02 agosto, 2016, disponible en https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/

[3] Juan Gómez Romero, SIGE2019, Github, disponible en https://github.com/jgromero/sige2019/blob/master/pr%C3%A1cticas/04.%20Modelos%20avanzados/dogs_cats.R

[4] JJ-Allaire, Deep-Learning-R, Convnets, página 447.