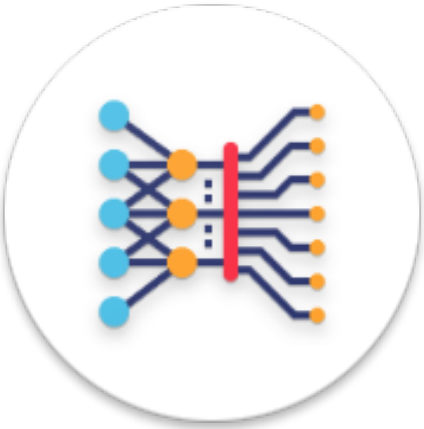

Aprendizaje Profundo para clasificación

Sistemas Inteligentes para la Gestión de la Empresa

José Antonio Córdoba Gómez
Fernando Izquierdo Romera

Granada, España - 30 de mayo de 2021
Máster en Ingeniería Informática

Índice general

Capítulo 1

Introducción

1.1 Definición de la tarea

El conjunto de datos a analizar trata sobre un conjunto de hilos de la red social **Reddit**. El funcionamiento de esta plataforma consiste en "foros.[en] los que los usuarios pueden hacer comentarios sobre un tema que se haya propuesto (por ejemplo, */music*). Estos comentarios llevan asociados un conjunto de metadatos como la fecha y la hora. También pueden ir acompañados de imágenes y emoticonos.

El objetivo de esta práctica es analizar esos comentarios e imágenes y clasificar este conjunto de forma binaria, intentando detectar lo que hoy en día se conocen como *fake news*. Para conseguir este objetivo usaremos **R** como lenguaje de programación y **Keras** para crear modelos de aprendizaje automático y profundo.

Capítulo 2

Análisis exploratorio de datos

2.1 Carga de datos

Antes de comenzar a realizar una análisis exploratorio inicial del conjunto de datos necesitamos realizar una lectura de los mismos.

Los datos originales se pueden encontrar en la dirección aportada por el profesor de la asiganatura en una carpeta de Google Drive: `https://drive.google.com/drive/folders/1qYWWdfdp-OAxKNXbKgAMh2x3p04X55TO`

Mientras que los datos ya limpios y procesados los dejamos disponible en una carpeta de Mega: `shorturl.at/bhwH4`

Comprobamos una imagen al azar, en nuestro caso **3zaywx.jpg** que se encuentra en el directorio `./data/images/medium10000_sixClasses/test/1/`. Una vez cargado convertimos la imagen a una array unidimensional y la graficamos para comprobar el resultado:

```
img_sample <- image_load(path =
 ↪  './data/images/medium10000_sixClasses/test/1/3zaywx.jpg',
                          target_size = c(150, 150))

```

```
img_sample_array <- array_reshape(image_to_array(img_sample),
                                  c(1, 150, 150, 3))

plot(as.raster(img_sample_array[1,,,] / 255))
```

Y el resultado es:

Figura 2.1: Resultado de la conversión a vector unidimensional de una imagen aleatoria del conjunto de test

Para la carga total de las imágenes definimos variables de posición de directorios para poder ubicar los datos fácilmente y de forma unívoca:

```
dataset_dir           <- './data/images/medium10000_twoClasses'
train_images_dir      <- paste0(dataset_dir, '/train')
val_images_dir        <- paste0(dataset_dir, '/val')
test_images_dir       <- paste0(dataset_dir, '/test')
comments_file         <- './data/comments/all_comments.tsv'
```

Vamos a cargar el conjunto de los metadatos desde el fichero `/multimodal_train.tsv`, convertimos el *timestamp* a tiempo *POSIX* y mutamos la columna **2_way_label** en función de su contenido: si es cero, la mapeamos a la clase **Disinformation**, y si es uno, a **Other**.

Columna	Tipo
author	*String*
clean_title	*String*
created_utc	*Double*
domain	*String*
hasImage	*Boolean*
id	*String*
image_url	*String*
linked_submission_id	*String*
num_comments	*Double*
score	*Double*
subreddit	*String*
title	*String*
upvote_ratio	*Double*
2_way_label	*Double*
3_way_label	*Double*
6_way_label	*Double*

Cuadro 2.1: Especificación de las columnas del conjunto de metadatos

```
metadata_train <- read_tsv(paste0(train_images_dir,
 ↪  "./data/images/medium10000_twoClasses/train/multimodal_train.tsv"))
metadata_train <- metadata_train %>%
  mutate(created_at = as.POSIXct(created_utc, origin="1970-01-01"))
   ↪  %>%
  select(-one_of('created_utc')) %>%
  mutate(class = ifelse(`2_way_label` == 0, 'Disinformation',
   ↪  'Other'))
```

2.2 Valores perdidos y procesamiento

Continuamos leyendo los comentarios. Podemos ver en el conjunto de comentarios que existen valores perdidos y podemos comprobar como la mayor parte de los valores perdidos se concentran en las dimensiones *isTopLevel* y *ups*. Omitimos estos valores perdidos.

```
comments <- read_tsv(comments_file)
summary(comments)
```

```
comments <- comments %>%
  drop_na()
```

```
       X1                id              author             body          isTopLevel
 Min.   :       0   Length:10002177   Length:10002177   Length:10002177   Mode :logical
 1st Qu.: 2625314   Class :character   Class :character   Class :character   FALSE:6134918
 Median : 5256822   Mode  :character   Mode  :character   Mode  :character   TRUE :3866201
 Mean   : 5290277                                                           NA's :1058
 3rd Qu.: 7973879
 Max.   :10601592
 NA's   :639
  parent_id         submission_id          ups
 Length:10002177   Length:10002177   Min.   :-9432.00
 Class :character   Class :character   1st Qu.:    1.00
 Mode  :character   Mode  :character   Median :    2.00
                                      Mean   :   23.81
                                      3rd Qu.:    7.00
                                      Max.   :42573.00
                                      NA's   :1058
```

Figura 2.2: Estado de resumen de los comentarios

2.3 Balanceamiento

A continuación combinamos los metadatos iniciales con estos comentarios y vamos a seleccionar posteriormente la distribución de las clases para analizar su estado y detectar posibles desbalanceos.

```
metadata_train_comments <- left_join
        (   x = metadata_train,
            y = comments,
            by = c("id" = "submission_id"),
            keep = FALSE, suffix = c('.publication', '.comment')
        )

metadata_train_comments

data_binary <- metadata_train %>%
  select(-one_of('3_way_label', '6_way_label', '2_way_label'))
```

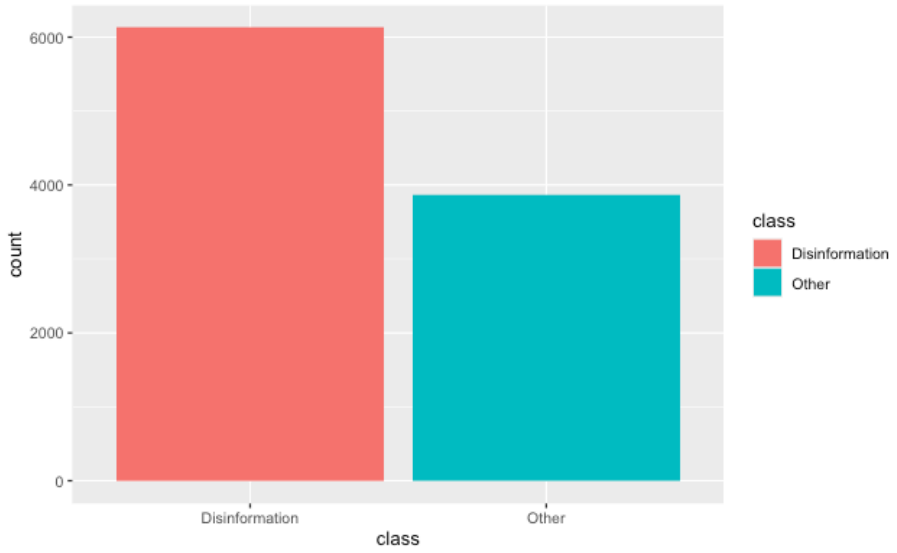

Figura 2.3: Las clases están desbalanceadas claramente del lado de la desinformación

Como hemos visto anteriormente, este problema de desbalanceo de clases podría incurrir en un proceso de aprendizaje erroneo y claramente sesgado, por lo que tomamos la decisión de corregir este suceso aplicando la técnica de **downsampling**.

```
data_factor <- data_binary
data_factor$class <- as.factor(data_factor$class)
predictors <- select(data_factor, -class)
data_balanced <- downSample(x = predictors,
                            y=data_factor$class, yname='class')
```

Comprobamos que este desbalanceamiento se ha corregido:

```
table(data_balanced$class)

ggplot(data_balanced) +
  geom_histogram(aes(x = class, fill = class), stat = 'count')
```

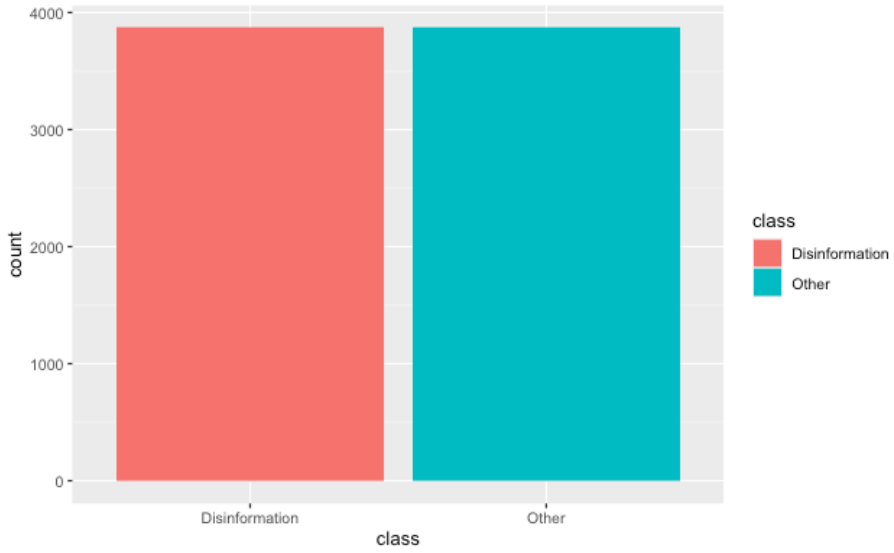

Figura 2.4: Las clases están balanceadas

2.4 Estudio gráfico

A continuación vamos a estudiar la evolución temporal de la densidad de los hilos con desinformación. Podemos notar como ha ido creciendo de forma notoria la desinformación con la evolución del tiempo.

```
ggplot(metadata_train, aes(x = created_at)) +
  geom_histogram(aes(fill = class))
```

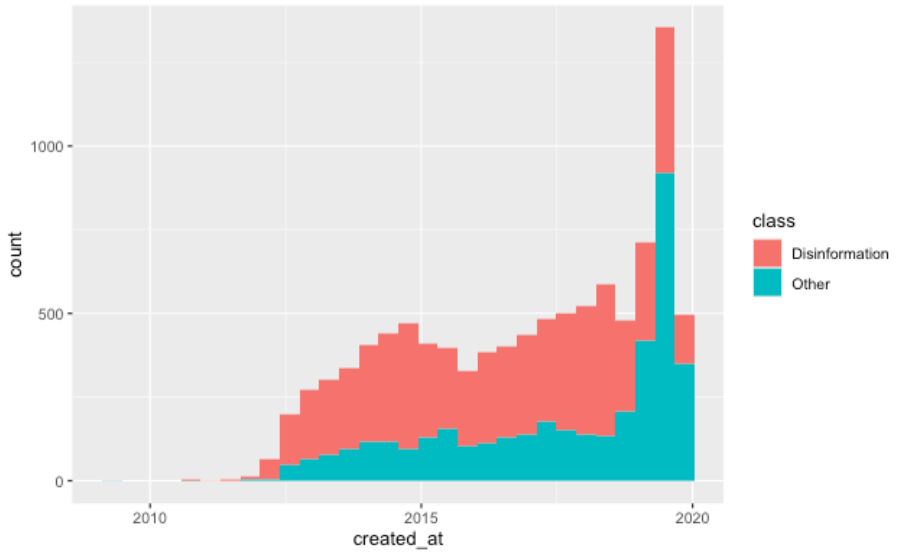

Figura 2.5: Evolución de la densidad de desinformación a lo largo del tiempo

Llegados a este punto, vamos a tratar de averiguar los 25 autores que más desinformación han propagado durante todo el tiempo.

```
plotdata <- data_binary %>%
  filter(class == "Disinformation") %>%
  count(author) %>%
  slice_max(n = 25, order_by = n, with_ties = FALSE)

ggplot(plotdata) +
  geom_bar(aes(x = author, y = n), stat = 'identity') +
  coord_flip()
```

Podemos observar como usuarios como all-top-today_SS o ApiContraption son los usuarios que más desinformación ha propagado en en toda la serie temporal.

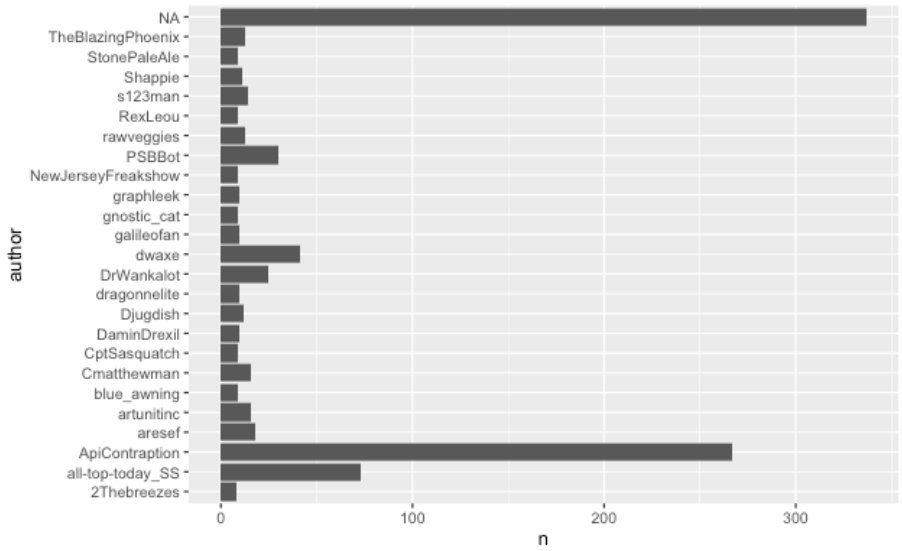

Figura 2.6: Los 25 autores que más desinformación han generado

Si analizamos los dominios que más desinformación han sufrido:

```
plotdata <- data_binary %>%
  filter(class == "Disinformation") %>%
  count(domain) %>%
  slice_max(n = 25, order_by = n, with_ties = FALSE)

ggplot(plotdata) +
  geom_bar(aes(x = domain, y = n), stat = 'identity') +
  coord_flip()
```

Podemos observar como gran parte de la desinformación la ha sufrido **reddit** y **imgur**.

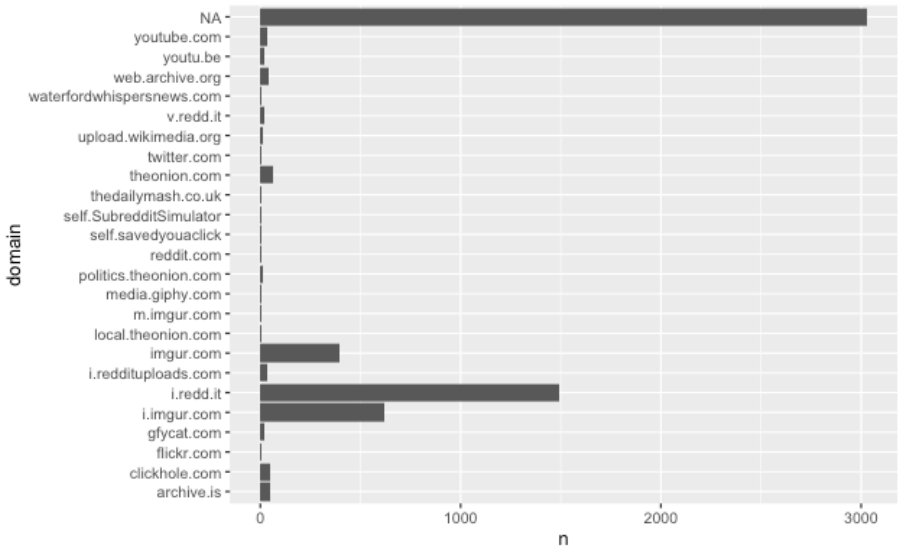

Figura 2.7: Los 25 dominios que más desinformación han sufrido

Podemos extraer las características de los comentarios como la longitud del título, la aparición de dígitos, o la aparición de *emojis* y analizar la distribución de la desinformación en función de alguno de estos parámetros.

```
data_binary_extended <- data_binary %>%
  mutate(title_text_exclamations = str_count(title, "!")) %>%
  mutate(title_text_caps = str_count(title, "[A-Z]")) %>%
  mutate(title_text_digits = str_count(title, "[0-9]")) %>%
  mutate(title_text_emojis = str_count(title,
   ↪  '[\U{1F300}-\U{1F6FF}]')) %>%
  mutate(title_text_emoji_flag = str_count(title,
   ↪  '\U{1F1FA}|\U{1F1F8}]'))
```

Vamos a analizar la distribución de densidad con respecto a la longitud del título.

```
ggplot(data_binary_extended) +
  geom_density(aes(x=title_text_caps, color=class, fill=class), alpha
   ↪  = 0.5)  +
  scale_x_continuous(trans="log10")
```

Podemos observar como los comentarios clasificados como desinformación se agrupan en la zona de títulos con una longitud baja.

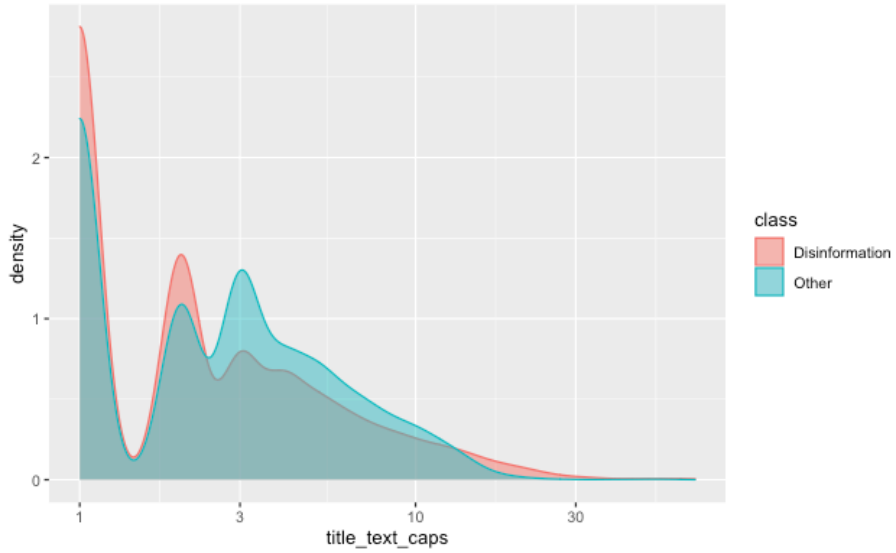

Figura 2.8: Hay una densidad de desinformación mayor en la zona de títulos más cortos

Si analizamos la distribución de densidad de la longitud del cuerpo del comentario, podemos ver que no se revela ninguna información útil ya que la distribución es equitativa en toda la serie temporal.

```
ggplot(data_binary_comments_extended) +
  geom_density(aes(x=body_text_caps, color=class, fill=class), alpha =
  ↪ 0.5)  +
  scale_x_continuous(trans="log10")
```

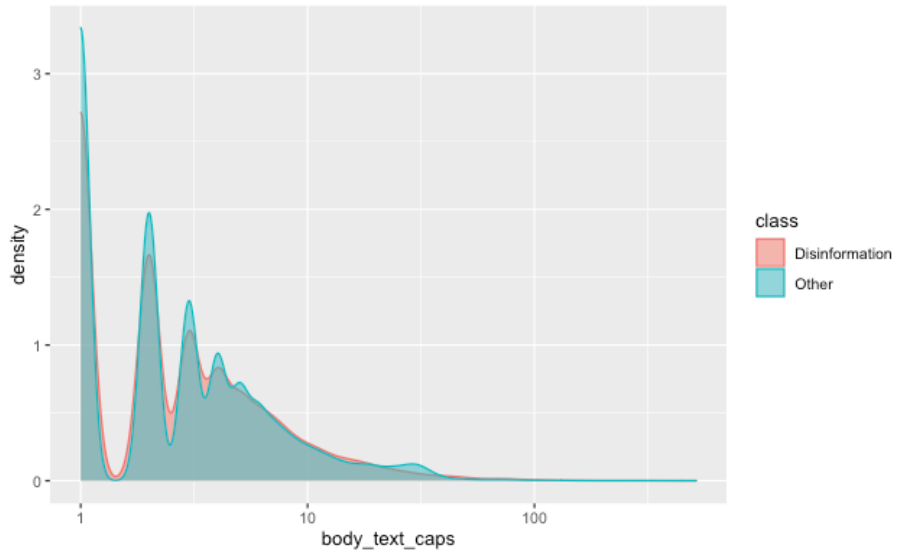

Figura 2.9: La distribución no refleja ninguna información evidente o característica

Por último vamos a analizar la distribución de la densidad de desinformación (y su clase negativa) con respecto a la cantidad de *emojis* utilizados dentro del cuerpo del comentario.

```
ggplot(data_binary_comments_extended) +
  geom_density(aes(x=body_text_emojis, color=class, fill=class), alpha
  ↪  = 0.5)  +
  scale_x_continuous(trans="log10")
```

Y podemos observar que aunque la distribución de desinformación y su clase contraria se encuentra prácticamente a la par durante toda la serie temporal, podemos notar que un uso medio alto de *emojis* se encuentra la tasa de desinformación más alta.

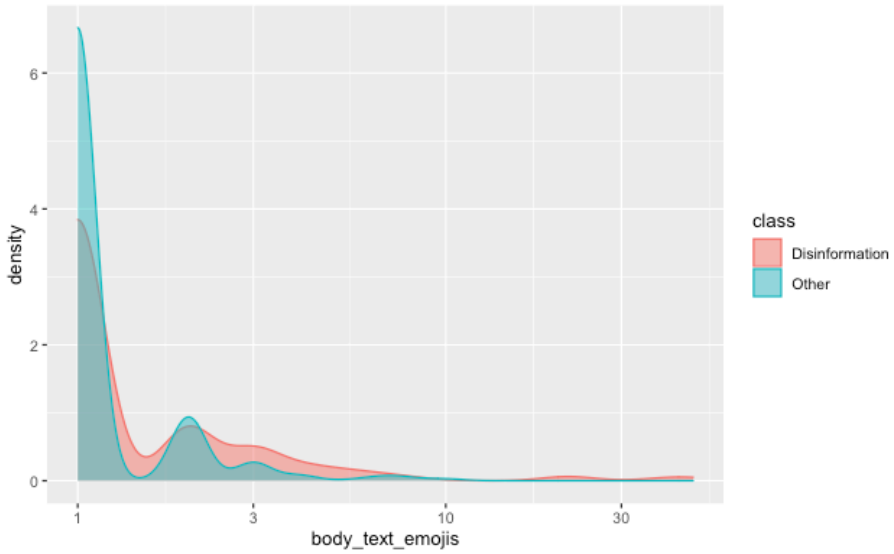

Figura 2.10: Se puede apreciar una mayor densidad de desinformación en una tasa de emojis dentro del cuerpo media-alta

Capítulo 3

Clasificación

En esta sección pretendemos experimentar con diferentes arquitecturas y técnicas de redes, buscando una optimización de nuestra clasificación. Nosotros nos centraremos en las imágenes, pero cabe mencionar que existen técnicas en las que se podrían combinar clasificadores (en este caso, uno para los comentarios y otro para las imágenes).

3.1 Modelo inicial

Definimos generadores de imágenes a cada conjunto, al de entrenamiento, validación y testeo, reescalando cada imagen al factor 1/255 lo que se denomina como **normalización**.

```
train_images_generator <- image_data_generator(rescale = 1/255)
val_images_generator   <- image_data_generator(rescale = 1/255)
test_images_generator  <- image_data_generator(rescale = 1/255)
```

Definimos un flujo de imágenes para cada conjunto. Entre los parámetros, indicamos que cada flujo introducirá al modelo imágenes de 64 píxeles por 64 píxeles en lotes de 128. Necesitamos un flujo para el entrenamiento, otro para la validación y finalmente otro para test.

```
train_generator_flow_1 <- flow_images_from_directory(
  directory = train_images_dir,
  generator = train_images_generator,
  class_mode = 'categorical',
```

```
  batch_size = 128,
  target_size = c(64, 64)
)

validation_generator_flow_1 <- flow_images_from_directory(
  directory = val_images_dir,
  generator = val_images_generator,
  class_mode = 'categorical',
  batch_size = 128,
  target_size = c(64, 64)
)

test_generator_flow_1 <- flow_images_from_directory(
  directory = test_images_dir,
  generator = test_images_generator,
  class_mode = 'categorical',
  batch_size = 128,
  target_size = c(64, 64)
)
```

Creamos la arquitectura del primer modelo (inspirado del proporcionado por el profesor), con capas convolucionales de dos dimensiones y con funciones de activación *relu*. Como la clasificación propuesta es binaria, la última capa es una capa densa de dos unidades con función de activación *softmax*.

```
model_1 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation =
   ↪  "relu", input_shape = c(64, 64, 3)) %>%
  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 = 2, activation = "softmax")
```

Le indicamos al marco **Keras** que compile el modelo anterior:

```
model_1 %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
```

Y estamos listos para lanzar el entrenamiento y la posterior evaluación de dicho entrenamiento de nuestro modelo. Para ello lanzamos cinco ejecuciones y nos quedamos con la media debido a la naturaleza no determinística de estos modelos. Esta métrica la aplicaremos a todos los modelos de aquí en adelante para poder compararlos.

```
NEXEC = 5
loss <- 0
precision <- 0
time <- 0

for (i in 1:NEXEC){
 start_model_1 <- Sys.time()

  history <- model_1 %>%
  fit_generator(
    generator = train_generator_flow_1,
    validation_data = validation_generator_flow_1,
    steps_per_epoch = 10,
    epochs = 10
  )

  end_model_1 <- Sys.time()

  plot(history)

  metrics <- model_1 %>%
    evaluate_generator(test_generator_flow_1, steps = 5)

  time <- time + (end_model_1 - start_model_1)
  precision <- precision + as.numeric(metrics[2])
  loss <- loss + as.numeric(metrics[1])

  message("Iteracion: ",i )
```

```
29    message("\t Tiempo: ", specify_decimal((end_model_1 -
      ↪  start_model_1), 2))
30    message("\tPrecisión: ", specify_decimal(as.numeric(metrics[2]), 2))
31    message("\tPerdida: ", specify_decimal(as.numeric(metrics[1]), 2))
32  }
33
34  time <- time / NEXEC
35  precision <- precision / NEXEC
36  loss <- loss / NEXEC
37
38  message("Total: ",i )
39  message("\t Tiempo: ", specify_decimal(time,2))
40  message("\tPrecisión: ", specify_decimal(as.numeric(precision),2))
41  message("\tPerdida: ", specify_decimal(as.numeric(loss),2))
```

Los resultados podemos observarlos a continuación. La gráfica de la progresión del error/precisión queda como sigue:

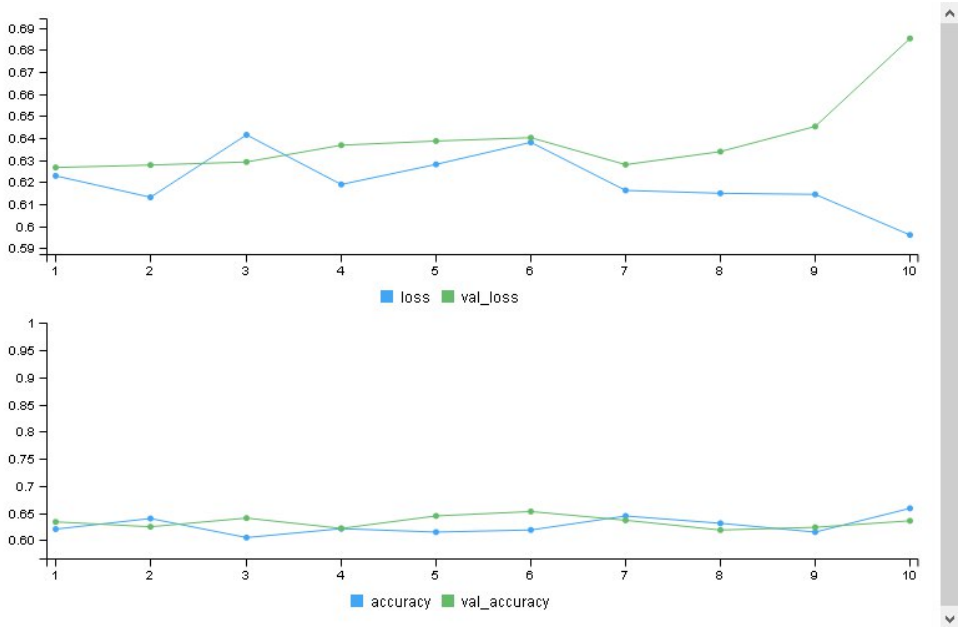

Figura 3.1: Evolución del proceso de aprendizaje sobre los parámetros de pérdida y precisión

Observamos el tiempo de cada ejecución y la precisión obtenida.

Iteración 1	
Tiempo	5.39 min
Precisión	0.58
Pérdida	0.67
Iteración 2	
Tiempo	4.71 min
Precisión	0.62
Pérdida	0.64
Iteración 3	
Tiempo	4.69 min
Precisión	0.56
Pérdida	0.65
Iteración 4	
Tiempo	4.69 min
Precisión	0.62
Pérdida	0.62
Iteración 5	
Tiempo	4.76 min
Precisión	0.58
Pérdida	0.67

Cuadro 3.1: Tiempo, precisión y pérdida de las cinco iteraciones

La media de las cinco iteraciones es:

Tiempo	4.85 min
Precisión	0.59
Pérdida	0.65

Cuadro 3.2: Resultado de evaluación del modelo en 5 ejecuciones

Esta será nuestro modelo base y en las secciones posteriores intentaremos mejorarlo.

3.2 Propuesta de otro modelo

Definimos un nuevo modelo con capas convolucionales, donde hemos aumentado el número de neuronas de cada capa además de incrementar el número de capas. En cuanto a la función de activación, hemos mantenido la activación *relu*. Finalmente hemos

aumentado el tamaño de la matriz con la que aplicamos las redes convolucionales (a 5x5). En resumen, la arquitectura de nuestra red consta de:

- Dos capas convolucionales iniciales de 32 filtros cada una. Este tipo de capas se utilizan para extraer la información más relevante de las imágenes.
- Típicamente, tras las capas convolucionales se suele aplicar **MaxPooling**. Consiste en reducir la dimensionalidad de la matriz que representa a la imagen.
- De nuevo alternamos dos pares de capas convolucionales con **max pooling**, de 64 y 128 respectivamente.
- Introducimos una capa **flatten**. Su función es para .ªplanar"los datos (su dimensión).
- Continuamos con dos capas de 256 y 512 neuronas respectivamente.
- Finalizamos con una capa de dos unidades (ya que nuestro problema de clasificación binaria) y de activación *softmax*.

```
model_2 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "relu", input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "relu") %>%
  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_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 2, activation = "softmax")
```

Las gráficas de la evolución del *loss* y la precisión se pueden observar a continuación.

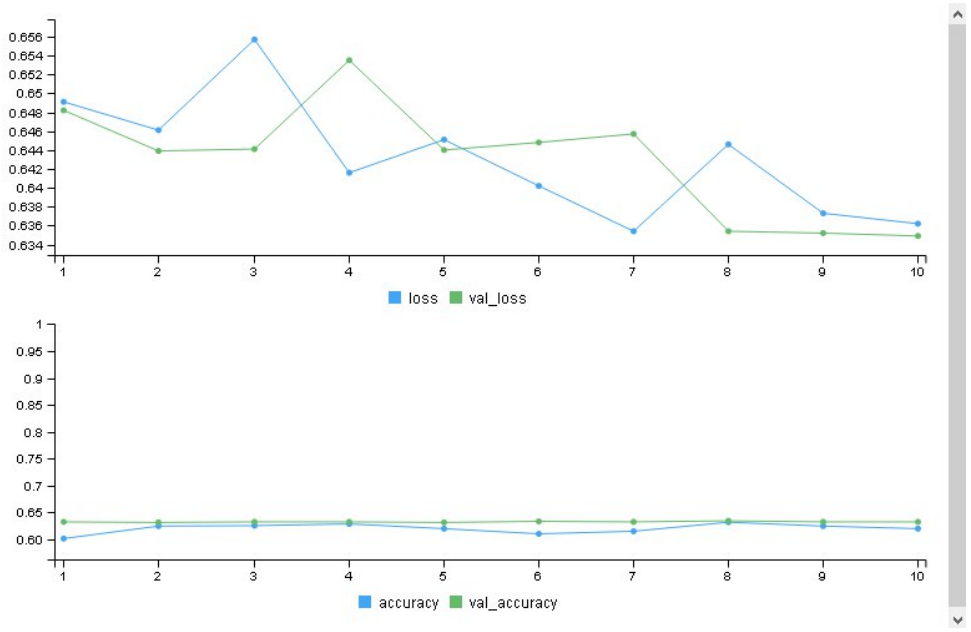

Figura 3.2: Evolución del proceso de aprendizaje sobre los parámetros de pérdida y precisión

Observamos el tiempo de cada ejecución y la precisión obtenida.

Iteración 1	
Tiempo	7.57 min
Precisión	0.61
Pérdida	0.68
Iteración 2	
Tiempo	7.40
Precisión	0.59
Pérdida	0.67
Iteración 3	
Tiempo	7.42 min
Precisión	0.57
Pérdida	0.65
Iteración 4	
Tiempo	7.35 min
Precisión	0.62
Pérdida	0.66
Iteración 5	
Tiempo	7.30 min
Precisión	0.62
Pérdida	0.63

Cuadro 3.3: Tiempo, precisión y pérdida de las cinco iteraciones

Vemos que los tiempos aumentan ligeramente con respecto al modelo anterior. La media de las cinco iteraciones es:

Tiempo	7.30 min
Precisión	0.60
Pérdida	0.66

Cuadro 3.4: Resultado de evaluación del modelo en 5 ejecuciones

Prácticamente no conseguimos mejorar nuestro modelo inicial (solo aumentamos la precisión en 0.01).

3.2.1 Intento de mejora

En este intento de mejora, hemos cambiado la función de activación **relu** por una de tipo sigmoidal **sigmoid**.

```
model_3 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "sigmoid", input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "sigmoid") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation =
   ↪  "sigmoid") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(5, 5), activation =
   ↪  "sigmoid") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "sigmoid") %>%
  layer_dense(units = 512, activation = "sigmoid") %>%
  layer_dense(units = 2, activation = "softmax")
```

Y volvemos a ejecutar el entrenamiento y la validación del modelo 5 veces haciendo la media. Los resultados se recogen en la siguiente tabla.

Iteración 1	
Tiempo	7.50 min
Precisión	0.58
Pérdida	0.69
Iteración 2	
Tiempo	7.47 min
Precisión	0.62
Pérdida	0.68
Iteración 3	
Tiempo	7.47 min
Precisión	0.62
Pérdida	0.68
Iteración 4	
Tiempo	7.52 min
Precisión	0.57
Pérdida	0.70
Iteración 5	
Tiempo	5.05 min
Precisión	0.61
Pérdida	0.67

Cuadro 3.5: Tiempo, precisión y pérdida de las cinco iteraciones

Tiempo	7.60 min
Precisión	0.60
Pérdida	0.68

Cuadro 3.6: Resultado de evaluación del modelo en 5 ejecuciones

Obtenemos una precisión idéntica a la anterior, por lo tanto continuaremos con la sigmoidal para las siguientes mejoras.

3.3 Aplicación de técnicas de mejora

En este apartado tratamos de aplicaremos las siguientes técnicas de mejoras.

- Capas **dropout**. Consiste en poner a 0 pesos de la red durante el entrenamiento para aumentar resistencia al aprendizaje.

- **Batch normalization**. Normalizar generalmente consiste en establecer los datos en un rango (normalmente entre 0 y 1). Aplicado al *batch*, se normalizan las activaciones de una neurona obtenidas con un mini-lote de tamaño k antes de pasarlas a la siguiente capa.
- **Data amplification**. Consite en aumentar el conjunto de datos, aplicando pequeñas variaciones como rotaciones, escalamientos y funciones espejo.

A continuación mostramos la arquitectura de nuestro siguiente modelo a probar. Al comienzo podemos observar el generador de imágenes mediante el cual llevaremos a cabo el *data amplification*. Finalmente, hemos añadido dos nuevas capas de *batch normalization* al 1 % y dos capas de *dropout*. Puesto que la primera capa de dropout viene tras una max pooling, le asignamos un rate de 0'01 y a la segunda que se sitúa tras una capa densa, proponemos un rate de 0'5.

```
# Aumento de datos de entrenamiento
train_images_generator <- image_data_generator(
  rescale = 1/255,
  rotation_range = 28,
  width_shift_range = 0.2,
  height_shift_range = 0.2,
  shear_range = 0.15,
  zoom_range = 0.15,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)

model_4 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "sigmoid", input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "sigmoid") %>%
  layer_batch_normalization(epsilon = 0.01) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(5, 5), activation =
   ↪  "sigmoid") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(5, 5), activation =
   ↪  "sigmoid") %>%
  layer_batch_normalization(epsilon = 0.01) %>%
```

```
22    layer_max_pooling_2d(pool_size = c(2, 2)) %>%
23    layer_flatten() %>%
24    layer_dropout(rate = 0.1) %>%
25    layer_dense(units = 256, activation = "sigmoid") %>%
26    layer_dropout(rate = 0.5) %>%
27    layer_dense(units = 512, activation = "sigmoid") %>%
28    layer_dense(units = 2, activation = "softmax")
```

Y volvemos a ejecutar el entrenamiento y la validación del modelo 5 veces haciendo la media. Los resultados se recogen en la siguiente tabla.

Iteración 1	
Tiempo	8.69 min
Precisión	0.57
Pérdida	0.68
Iteración 2	
Tiempo	8.78 min
Precisión	0.59
Pérdida	0.69
Iteración 3	
Tiempo	8.83 min
Precisión	0.62
Pérdida	0.68
Iteración 4	
Tiempo	8.84 min
Precisión	0.61
Pérdida	0.68
Iteración 5	
Tiempo	8.92 min
Precisión	0.62
Pérdida	0.68

Cuadro 3.7: Tiempo, precisión y pérdida de las cinco iteraciones

Tiempo	8.81 min
Precisión	0.61
Pérdida	0.68

Cuadro 3.8: Resultado de evaluación del modelo en 5 ejecuciones

Seguimos sin obtener una gran mejora en la precisión (solo 0'1).

3.3.1 Cambio de la capa de activación

El último cambio que vamos a aplicar en nuestra experimentación de nuevo la capa de activación **relu** y vamos a añadir 10 épocas más, es decir, 20 en total.

```
model_5 <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "relu", input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(filters = 32,  kernel_size = c(5, 5), activation =
   ↪  "relu") %>%
  layer_batch_normalization(epsilon = 0.01) %>%
  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_batch_normalization(epsilon = 0.01) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 2, activation = "softmax")

model_5 %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)
}
```

Los resultados se pueden analizar en la tabla correspondiente.

Iteración 1	
Tiempo	17.52 min
Precisión	0.63
Pérdida	0.68
Iteración 2	
Tiempo	17.64 min
Precisión	0.64
Pérdida	0.69
Iteración 3	
Tiempo	17.60 min
Precisión	0.62
Pérdida	0.68
Iteración 4	
Tiempo	17.58 min
Precisión	0.63
Pérdida	0.68
Iteración 5	
Tiempo	17.65 min
Precisión	0.65
Pérdida	0.68

Cuadro 3.9: Tiempo, precisión y pérdida de las cinco iteraciones

Tiempo	17.6 min
Precisión	0.63
Pérdida	0.68

Cuadro 3.10: Resultado de evaluación del modelo en 5 ejecuciones

Obtenemos una ligera mejora hasta el 0’63.

Capítulo 4

Conclusiones

Al concluir el trabajo realizado podemos comentar algunos aspectos que pueden ser interesante:

- Diseñar modelos de aprendizaje profundo requiere de un alto grado de conocimiento sobre el problema propuesto. Aunque en un principio se pueda reutilizar ciertos tipos de patrones para ciertos tipos de problemas, como en nuestro caso, el uso de capas convolutivas de dos dimensiones, es necesario conocer bien la fuente de datos a generalizar.

- La realización de modelos, requieren de una alta capacidad de computación que den soporte a la etapa de entrenamiento, por lo que se ve necesario obtener hardware a la altura de las circunstancias. En nuestro caso, hemos usado una instancia de máquina virtual remota en **Google Cloud** con **Windows Server 2019** y una sesión **RDP** para realizar las prácticas sobre una instancia optimizada para la computación ubicada en Países bajos. Esto nos ha aligerado muchísimo el entrenamiento de los modelos, pero no todo el mundo tiene acceso a estos recursos o directamente recursos económicos, por lo que concluimos que la generación de modelos sobre este tipo de datos, no es para todo el mundo.

- Con respecto a las herramientas utilizadas, cabe destacar que **Keras** permite una alta modularización y personalización de la arquitectura de las redes, permitiendo así un amplio margen de pruebas para detectar qué aspectos de las redes mejoran la eficiencia de nuestro clasificador.

- Los diferentes modelos realizados, no obtienen una mejora notoria ni alcanzan cotas de precisión muy altas, por lo que utilizar estos modelos en entornos de

producción puede no ser muy adecuado. Dejamos entrever, que usar modelos ya entrenados para la clasificación de este tipo de imágenes, podría ser mucho mejor idea que hacerlo por nuestra cuenta. De las mejoras en concreto, hemos podido comprobar como el aumento de épocas en el entrenamiento de las redes es lo que más impacto ha tenido en la mejora de los modelos.

- En ejecuciones anteriores, hemos tratado de variar el tamaño del kernel e incluso aplicar otro tipos de filtros, pero los resultados han sido exactamente igual de precisos que sin estas modificaciones que sólo dificultan el modelo, por lo que hemos optado por rechazarlos. Además, el hecho de que los experimentos no sean determinísticos hace que las ejecuciones tengan una faceta bastante aleatoria a la hora de comparar los modelos.

Bibliografía

[1] Keras. https://keras.io/api/

[2] Repositorio para la asignatura del profesor Juan Gómez Romero. https://github.com/jgromero/sige2021