Máster Universitario en Ingeniería Informática

SISTEMAS INTELIGENTES PARA LA GESTIÓN DE LA EMPRESA

Práctica 2:
Deep Learning para clasificación

UNIVERSIDAD DE GRANADA

Pablo Alfaro Goicoechea
Carlos Morales Aguilera
pabloalfaro@correo.ugr.es
carlos7ma@correo.ugr.es

Curso Académico 2020-2021

Índice

1. Introducción

1.1. Objetivo

El objetivo de esta práctica es resolver un problema de reconocimiento de patrones utilizando redes neuronales artificiales. Deberá evaluar el uso de varios tipos de redes neuronales para resolver un problema de reconocimiento de noticias falsas en la red social **Reddit** [1].

1.2. Fakeddit

El conjunto de datos **Fakeddit** [2] contiene una serie de noticias relacionadas con todo tipo de hilos tanto de política como cultura. Algunas de las noticias que contienen son falsas y afectan negativamente tanto a las redes sociales como a las comunidades off-line.

Normalmente los modelos requieren de conjuntos de noticias completos, en los que se incluya bastante información siendo realmente escasos, por lo que **Fakeddit** se presenta como un conjunto completo de datos con diferentes noticias que contienen tanto imágenes, texto y metadatos, los cuales se utilizan para poder determinar la veracidad o falsedad de las noticias.

Si bien el conjunto contiene una cantidad de datos, en este proyecto se utilizará el subconjunto proporcionado por el docente de la asignatura de 10.000 imágenes con el fin de definir una red y realizar la detección. En este caso se utilizará únicamente el modelo de dos clases: Noticia verdadera o falsa.

2. Redes Neuronales

Las redes neuronales son sistemas computacionales, bioinspirados en el sistema neuronal humano, tomando como ejemplo las neuronas que constituyen el cerebro, aportando a los ordenadores de una inteligencia artificial. Estas redes se encuentran formadas por una serie de unidades básicas denominadas neuronas que se comunican entre sí formando una red que adquiere conocimiento.

Debido a su similitud con el cerebro humano, la principal característica de estas redes neuronales es su capacidad de aprendizaje. Este modelo se puede aplicar tanto en problemas de aprendizaje supervisado, no supervisado como reforzado.

El problema de **Fakeddit** [2] se presenta como un problema real de detección de imágenes y procesamiento mediante redes neuronales, siendo este un problema de aprendizaje supervisado, donde se debe entrenar una red neuronal de forma que esta sea capaz de clasificar una noticia según su veracidad o falsedad, haciendo uso para ello de una red que se ajuste lo más fielmente posible a las características del problema.

Nota: Las redes implementadas al hacer uso de una GPU, son modelos *no determinísticos*, lo que quiere decir que los resultados obtenidos no serán reproducibles, por lo que se establecerá el criterio comentado en clase de realizar varias ejecuciones y obtener el

promedio de las mismas para establecer conclusiones (comprendiendo el fin académico del proyecto, ya que en un caso real y con otros recursos no sería válido este planteamiento).

3. Herramientas

Para la realización de este proyecto se han utilizado las siguientes herramientas:

3.1. Tensorflow

Tensorflow [3] es una biblioteca de código abierto para el aprendizaje automático multitarea. Ha sido desarrollado por Google para realizar las funciones de backend y entrenar redes neuronales para detectar, analizar y descifrar patrones y correlaciones. Emplea para ello unas técnicas de aprendizaje y razonamiento humanos. Hoy en día, se utiliza en la investigación y los productos de Google. **Tensorflow** ofrece varios niveles de abstracción para crear y entrenar modelos y brinda flexibilidad y control con características como la API funcional de **Keras**.

3.2. Keras

Keras [4], por otro lado, es una biblioteca de redes neuronales de alto nivel que se ejecuta en la parte superior de *TensorFlow*, *CNTK* o *Theano*. El uso de Keras en el aprendizaje profundo permite la creación de prototipos fácil y rápida, así como la ejecución sin problemas en la CPU y la GPU. Este marco está escrito en código Python, que es fácil de depurar y permite una fácil extensibilidad.

4. Realización del proyecto

4.1. Lectura de datos

Al tratarse de una gran cantidad de imágenes, por cuestiones de recursos limitados de memoria y computación se utilizarán generadores de flujo, tal y como se propone en la asignatura, para de forma dinámica ir generando una serie de lotes de imágenes y finalmente poder obtener el funcionamiento deseado. Cabe destacar que se escalan las imágenes a tamaño 255.

Los flujos se definen de las siguiente forma:

```
train_generator_flow <- flow_images_from_directory(
  directory = train_images_dir,
  generator = train_images_generator,
  class_mode = 'categorical',
  batch_size = 128,
  target_size = c(64, 64)      # (w x h) --> (64 x 64)
)

validation_generator_flow <- flow_images_from_directory(
  directory = val_images_dir,
  generator = val_images_generator,
  class_mode = 'categorical',
  batch_size = 128,
  target_size = c(64, 64)      # (w x h) --> (64 x 64)
)

test_generator_flow <- flow_images_from_directory(
  directory = test_images_dir,
  generator = test_images_generator,
  class_mode = 'categorical',
  batch_size = 128,
  target_size = c(64, 64)      # (w x h) --> (64 x 64)
)
```

En cuanto a la parametrización de los flujos definidos caben destacar tres puntos:

- El tamaño de **batch** se ha establecido a 128 ya que consideramos que es un tamaño lo suficientemente pequeño para ofrecer variabilidad y lo suficientemente grande para obtener conjuntos de un tamaño aceptable de imágenes.
- El modo de clasificación se ha establecido a *categorical*, pese a tener dos categorías, ya que debería poder definirse como *binary*, pero esto resulta en un bug que tras investigar no hemos obtenido una solución, para cuestiones del problema, realmente no varía, aunque exista una opción ideal para ello.
- El tamaño de **target** se ha mantenido el ofrecido inicialmente, ya que reducciones mayores obtienen malos resultados y reducciones menores conllevan problemas de memoria.

4.2. Análisis exploratorio de los datos

Para este problema partimos de un análisis de los datos que se nos ha facilitado. A partir de la información obtenida en ese análisis hemos ampliado el estudio analizando otras características.

4.2.1. Análisis propuesto

El análisis incial propuesto se hace para el problema con dos clases y 50 imágenes. Comienza revisando la distribución de las clases para comprobar si el problema esta balanceado.

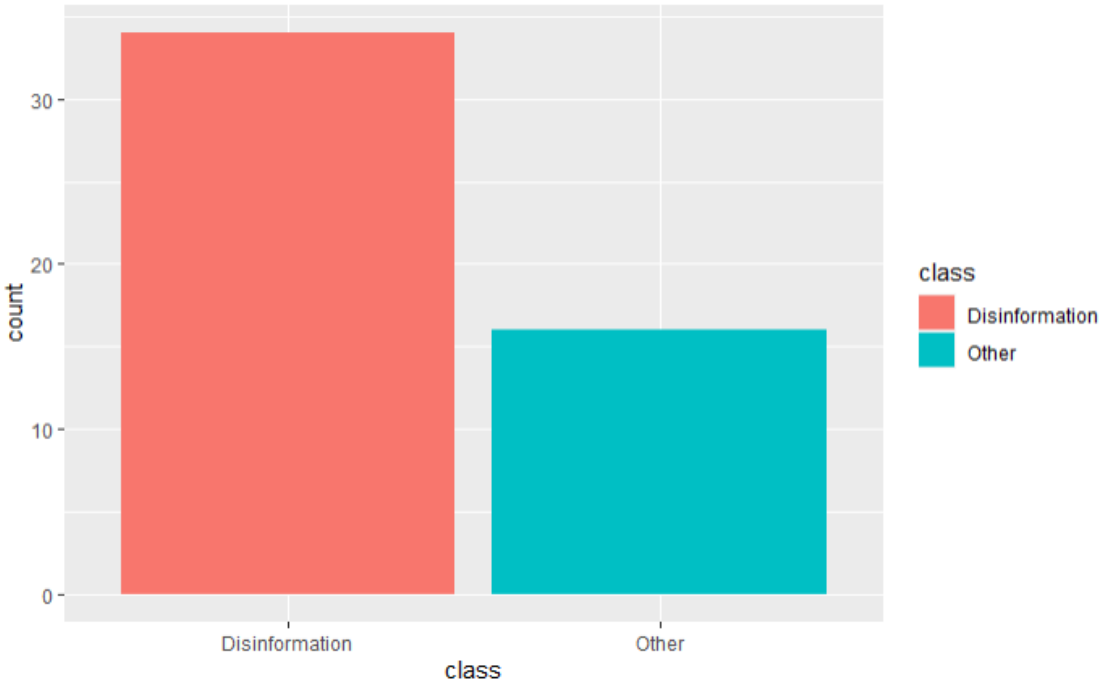

Imagen 1: Distribución de datos (50 imágenes)

Como se puede ver la clase de **Desinformación** tiene alrededor del doble de ejemplos de los que contiene la clase **Otros**. Esto se debe tener en cuenta durante el proceso de preprocesamiento de datos y a la hora de determinar las técnicas de evaluación de los modelos.

El siguiente paso plantea la evolución temporal de los datos. Esta evolución indica cómo ha variado la cantidad de artículos con desinformación. Esta información se obtiene con un grafo de frecuencia acumulada.

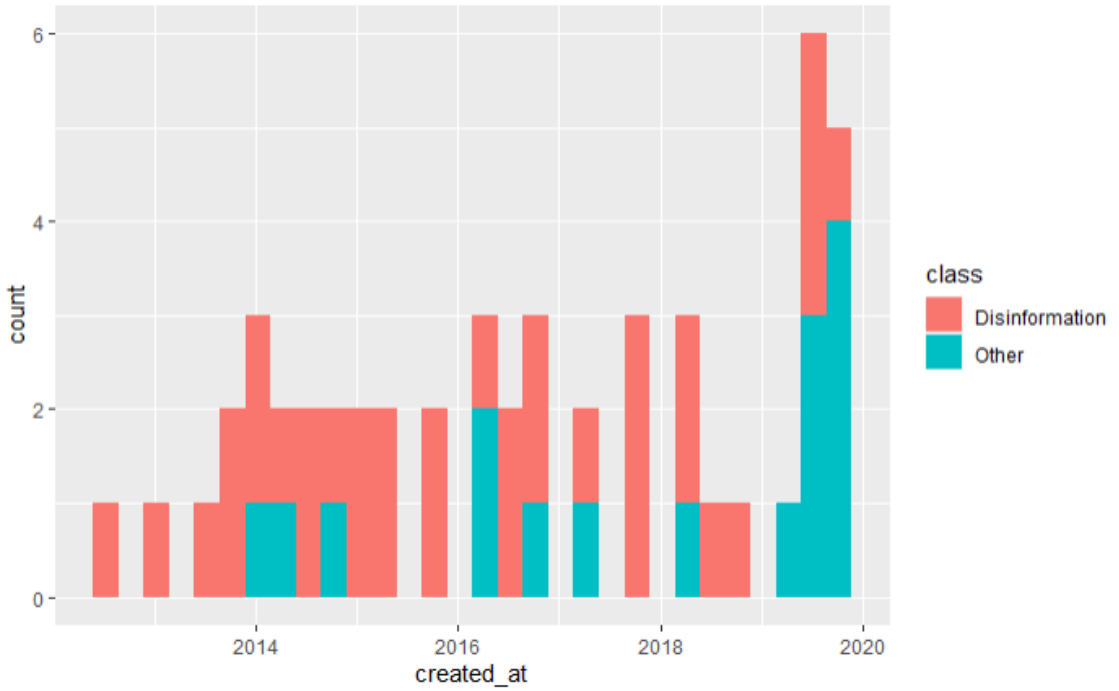

Imagen 2: Evolución temporal (50 imágenes)

Se pude ver que durante los primeros años la mayoría de estos posts eran de desinformación. Con el tiempo han ido aumentando las publicaciones de otro tipo hasta llegar incluso a superar a las de desinformación en los últimos años.

Otro aspectos que se ha estudiado ha sido el hecho de que aparezcan determinados carácteres en los posts. Por un lado se ha observado esta información en los títulos y por otro lado en el cuerpo del artículo. En los siguientes gráficos se ve la relación entre el número de mayúsculas que aparecen y la clase a la que pertecenecen.

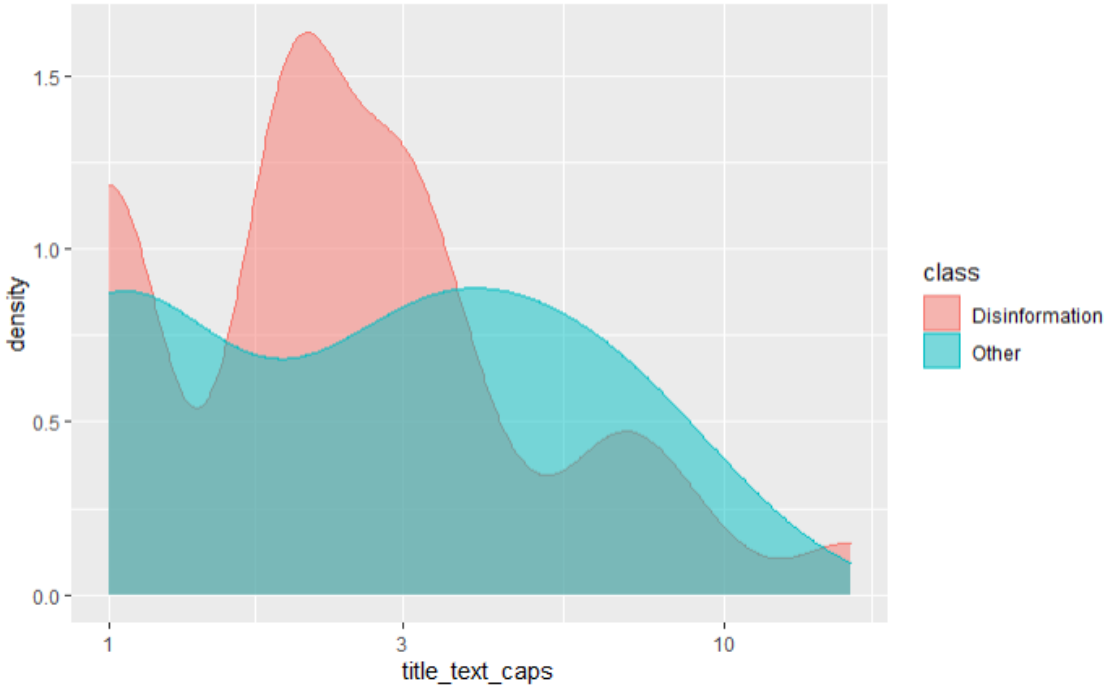

Imagen 3: Mayúsculas en títulos (50 imágenes)

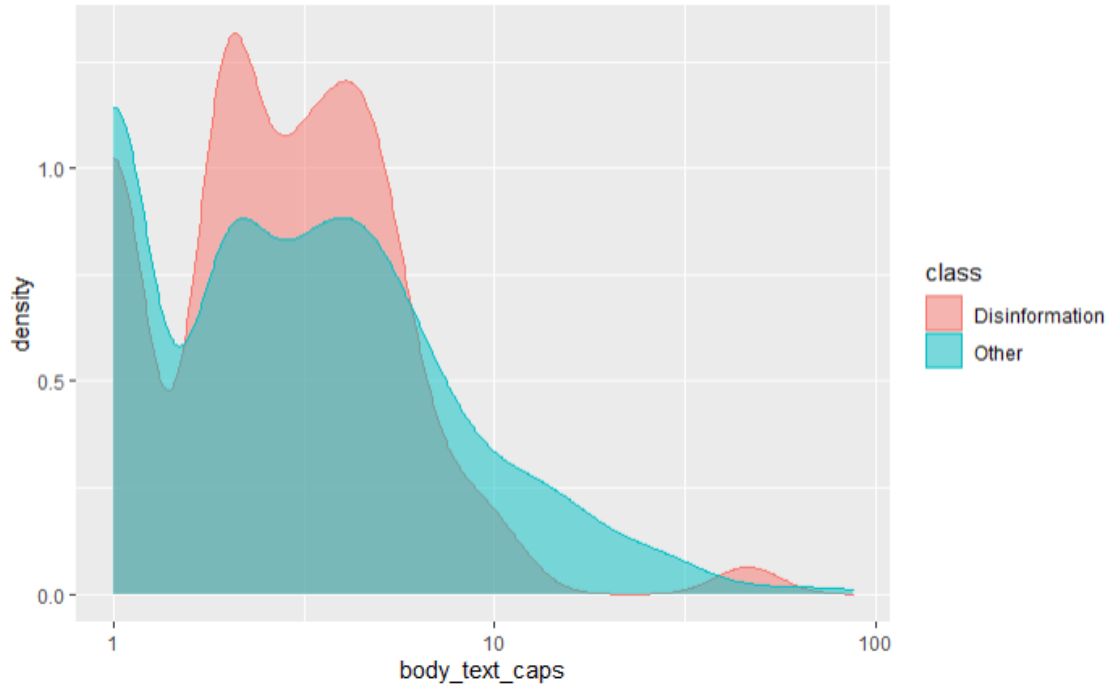

Imagen 4: Mayúsculas en el cuerpo del post (50 imágenes)

Un dato interesante que se ha investigado ha sido comprobar que usuarios estaban publicando más artículos con desinformación.

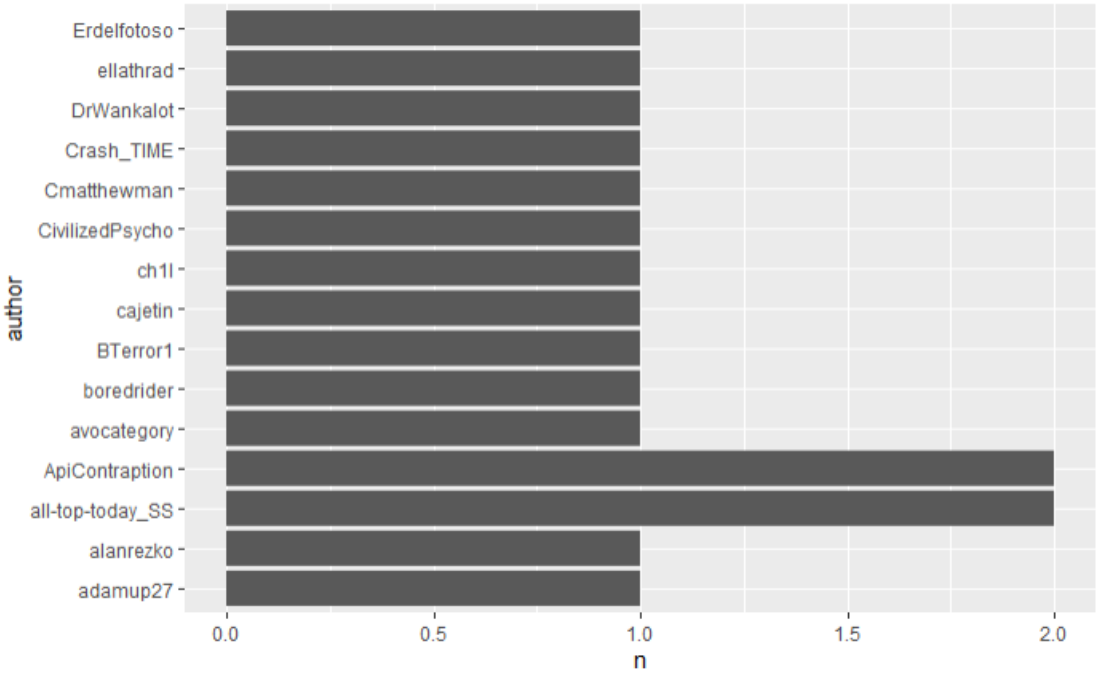

Imagen 5: Autores (50 imágenes)

4.2.2. Análisis extendido

Lo primero que se ha hecho ha estudiado de manera general el tipo de datos con los que se trabaja para el dataset de 10000 imágenes. Se ha utilizado la librería visdat que también indica los valores perdidos.

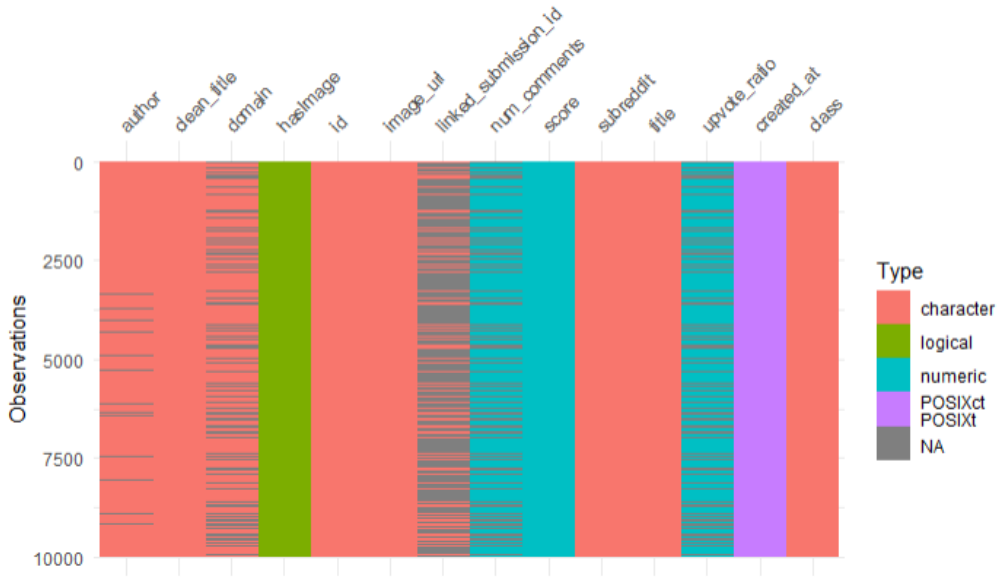

Imagen 6: Variables (10000 imágenes)

A continuación se han replicado los pasos anteriores para el dataset de 10000 imágenes.

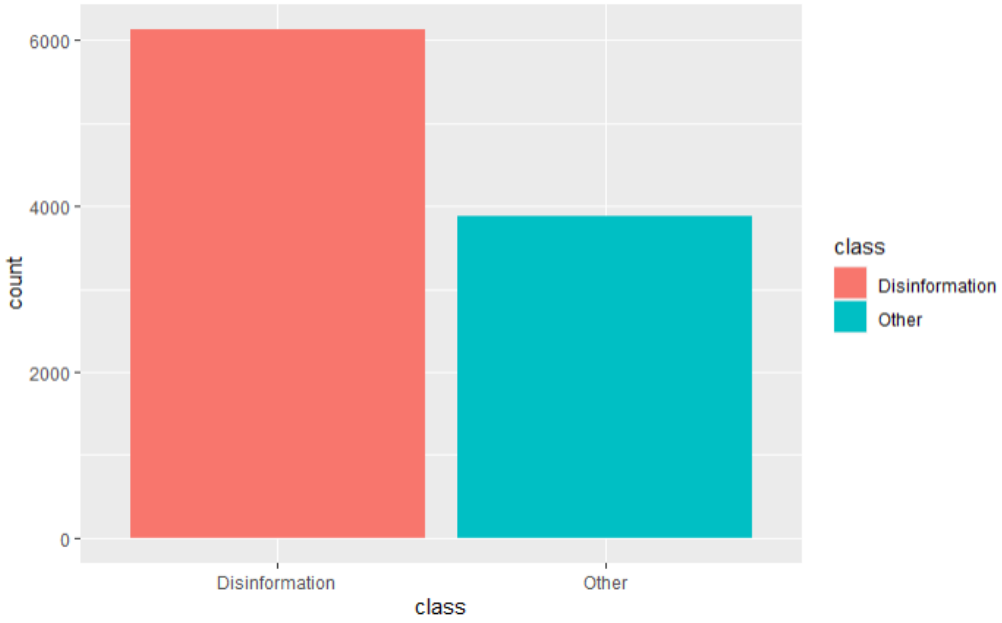

Imagen 7: Distribución de datos (10000 imágenes)

La diferencia entre las dos clases sigue siendo notable aunque se puede puede percibir que en este caso es menor en comparación con el anterior dataset.

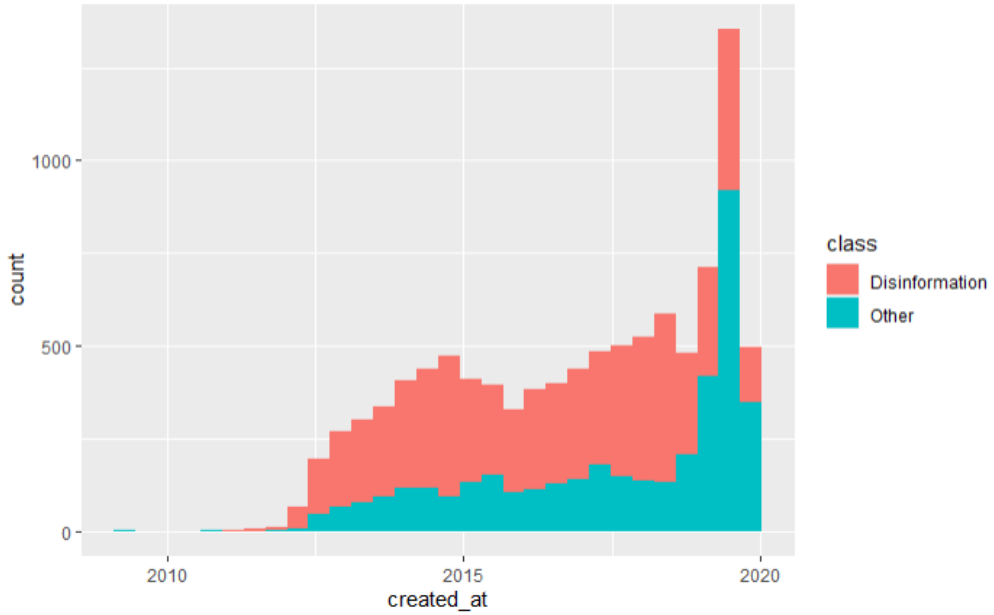

Imagen 8: Evolución temporal (10000 imágenes)

Permite ver más claramente la tendencia de la desinformación a lo largo de los años. La tendencia sigue siendo la misma que se ha observado previamente.

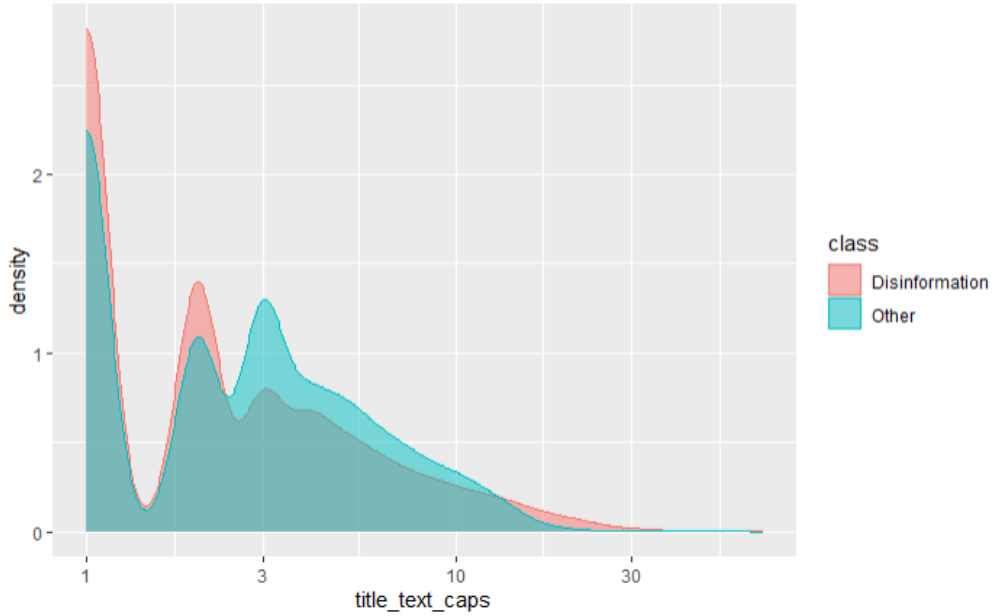

Imagen 9: Mayúsculas en títulos (10000 imágenes)

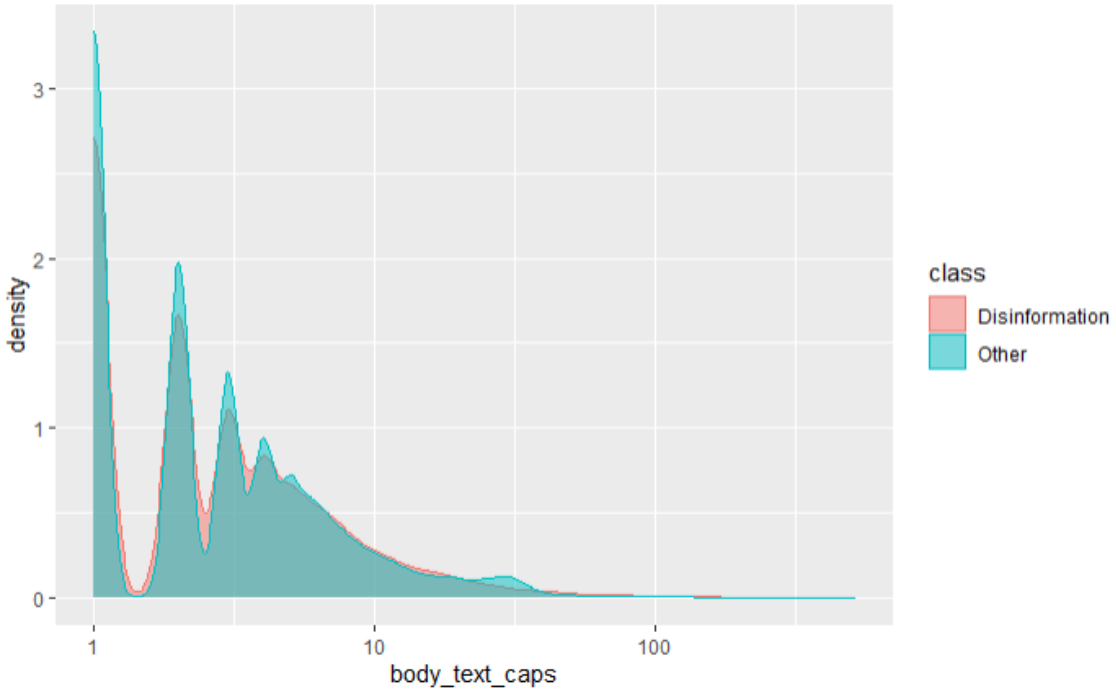

Imagen 10: Mayúsculas en el cuerpo del post (10000 imágenes)

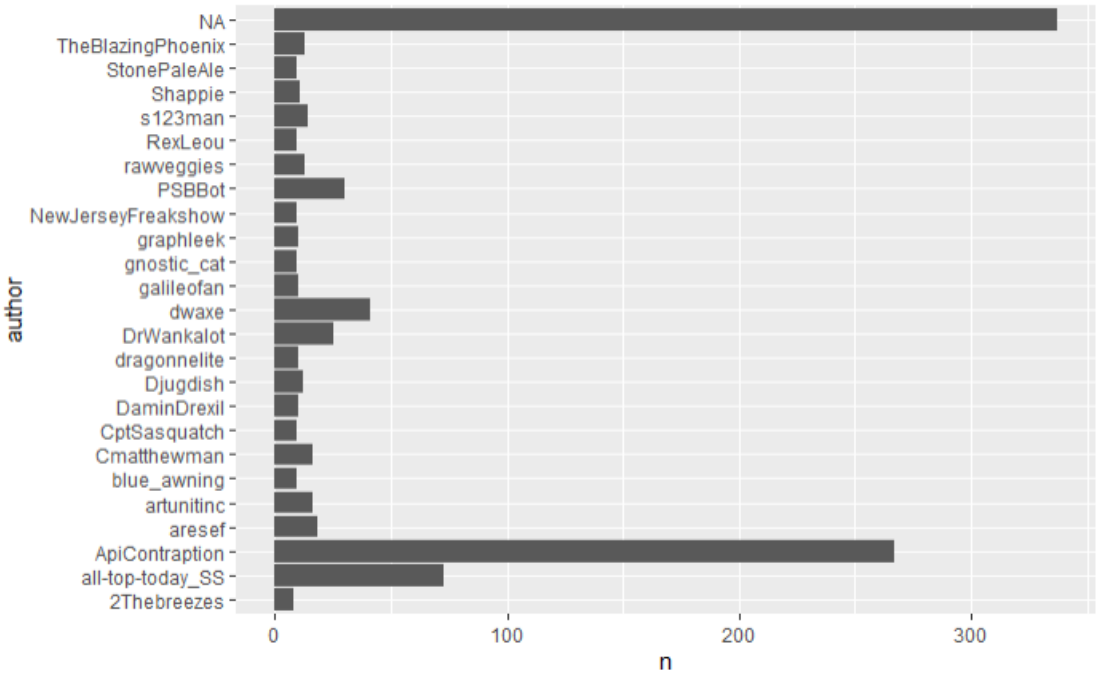

Imagen 11: Autores (10000 imágenes)

Los datos de este problema estan tomados de diferentes dominios de reddit. A raíz de esto, un dato interesante que podíamos investigar en qué dominios se publican más posts con desinformación. Esta información se ha obtenido del dataset de 50 imágenes y del de 10000.

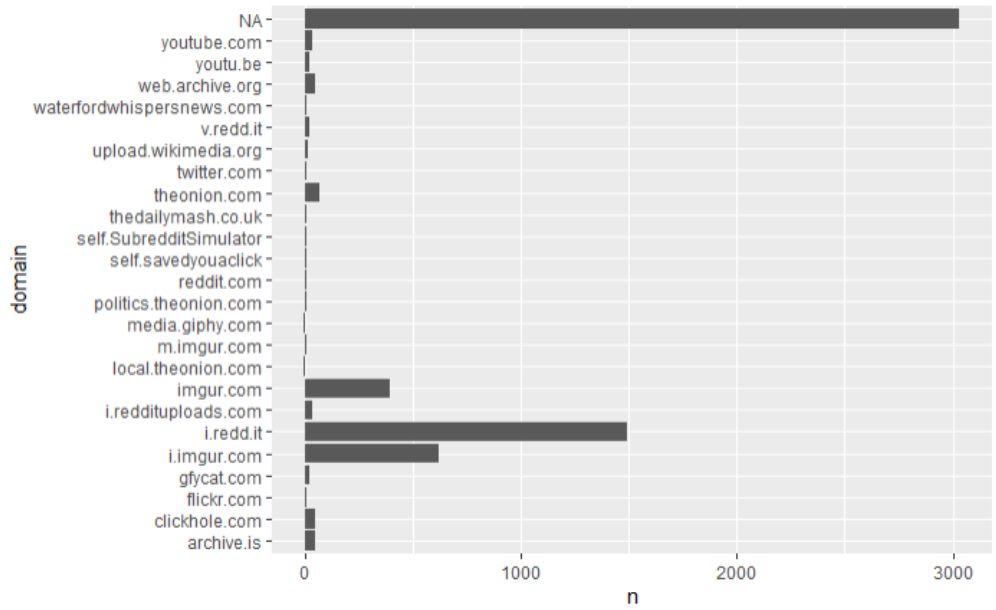

Imagen 12: Dominios (50 imágenes)

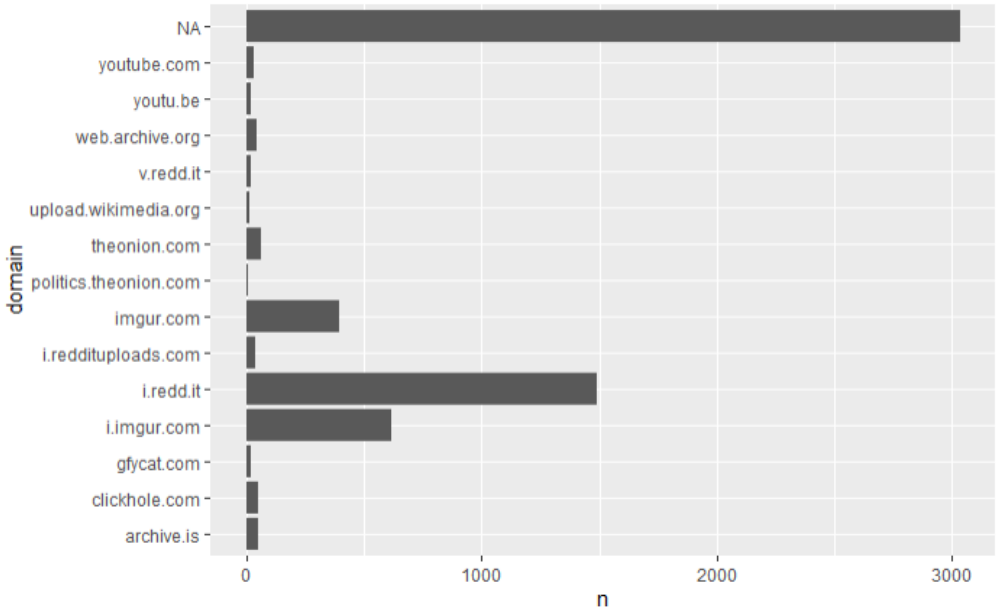

Imagen 13: Dominios (10000 imágenes)

Se puede ver que el dominio dónde más artículos de desinformación se publican es i.reddit.it.

Finalmente se ha investigado la relación que había entre el valor de las variables: número de comentarios y las valoraciones positivas con la variable objetivo de clasificación. Estas gráficas se han obtenido con la librería funModeling.

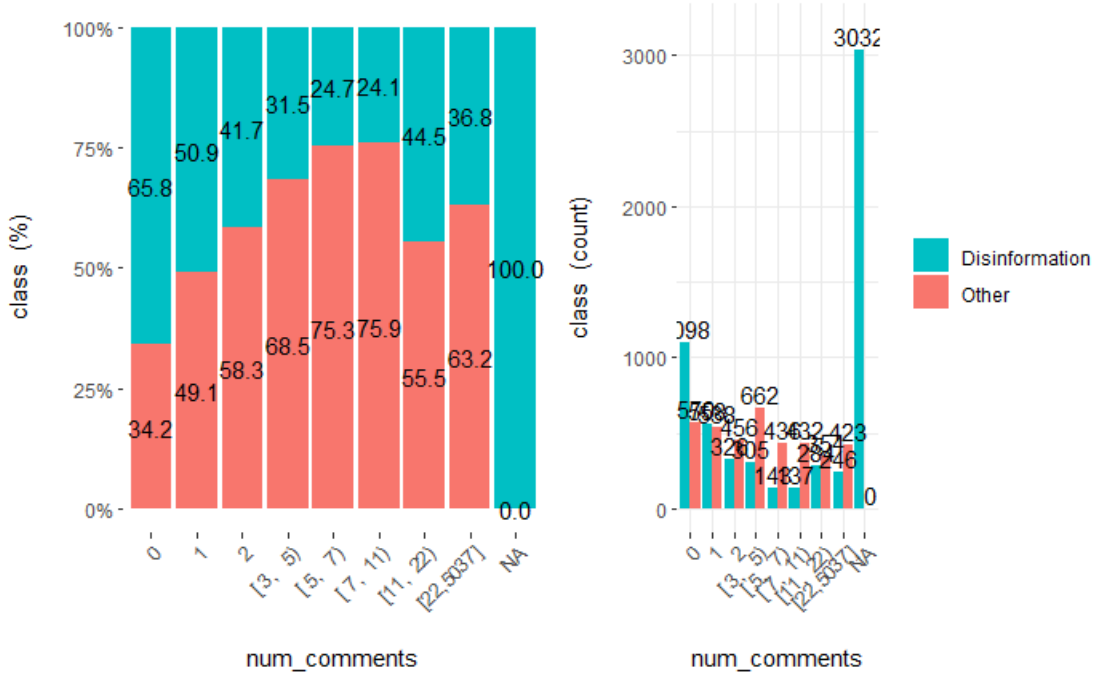

Imagen 14: Comentarios (10000 imágenes)

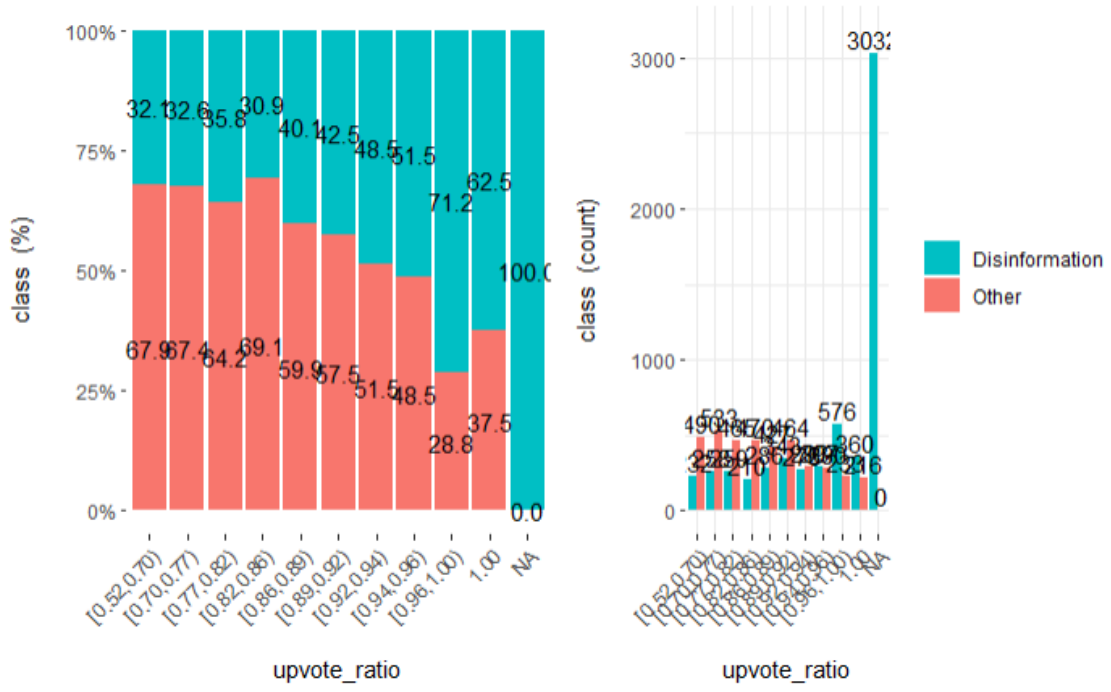

Imagen 15: Upvotes (10000 imágenes)

4.3. Modelo de red neuronal inicial

4.3.1. Diseño del modelo

El primer modelo que se plantea es el propuesto por el profesor de la asignatura D. Juan Gómez Romero, por lo que para empezar con el problema, analizaremos esta red y las adaptaciones que ha sufrido para llegar a obtener un modelo válido, ya que inicialmente esta red está planteada para el conjunto de datos más reducido.

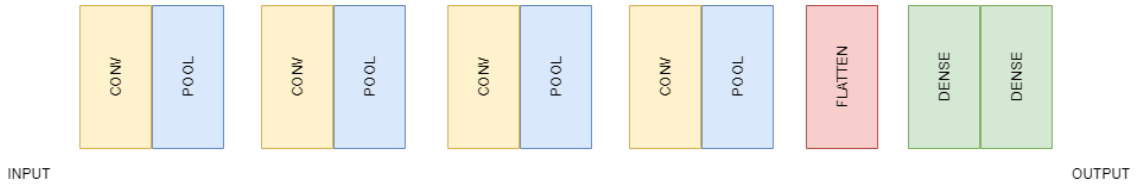

Imagen 16: Red Neuronal Inicial

El modelo propuesto es el de una **Red Neuronal Convolutiva**, que posee una serie de capas de convolución y otras capas de *pooling*, por lo que se necesitará crear un modelo **secuencial**.

Para la definición del modelo secuencial se utiliza la clase **Sequential** [5] perteneciente a los modelos que ofrece Keras. A continuación se añadirán las diferentes capas:

- Una capa de entrada de dimensiones (64x64x3 canales de color) con una convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 32, con una función de activación **RELU** [6] conocida como *Rectified Linear Unit*, consistente en que si la salida que realiza la operación es positiva, se activa y emite el valor positivo, si es negativa devuelve un 0 (no se activa). Para ello se utiliza la capa **Conv2D** [7] de las layers de Keras.
- Una capa de *pooling* de 2 dimensiones, que reduce el tamaño de la muestra tomando el valor máximo de una ventana definida como valor de representación de la ventana, en nuestro caso de un tamaño de 2x2. Para ello se utiliza la capa **MaxPooling2D** [8] de las layers de Keras.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 64, con una función de activación RELU.
- Una capa de pooling de 2 dimensiones, que utiliza una ventana de 2x2.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 128, con una función de activación RELU.
- Una capa de pooling de 2 dimensiones, que utiliza una ventana de 2x2.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 128, con una función de activación RELU.
- Una capa de pooling de 2 dimensiones, que utiliza una ventana de 2x2.

- Una capa de *flatten* o *aplanamiento*, que reduce las dimensiones. Para ello se utiliza la capa **Flatten** [9] de las layers de Keras.
- Una capa de conexión de neuronas, donde todas las neuronas se encuentran conectadas y se limitan a definirse en diferentes clases. Para ello se utiliza una capa **Dense** [10]. Esta capa sigue la estructura *Dense(units, activation)* realizando la operación *output=activation(dot(input,kernel) + bias)*, donde la función de activación indica si se activa la neurona y units representa el número de unidades de salida de la capa.
- Una capa de salida, que recibe la información de la capa previa y realiza una operación de softmax que convierte un vector real en un vector de probabilidades categóricas. Para ello se utiliza una capa Dense que utiliza como función de activación la función **softmax** [11] y como número de unidades de salida 2 unidades (que representan las dos clases existentes de clasificación).

El código asociado a la definición del modelo es el siguiente:

```
model <- 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")
```

4.3.2. Entrenamiento del modelo

Para poder entrenar el modelo, primero es necesario configurar las métricas que este va a emplear para evaluar tanto el *accuracy* como el *loss*, e indicar el método de aprendizaje, también llamado *optimizador*.

Para el cálculo de la pérdida (*loss*) se ha utilizado la **entropía cruzada** [12] mediante la función **categorical_crossentropy** [13].

Por otro lado, se ha utilizado el algoritmo de *Root Mean Square Propagation*, conocido como **RMSPROP** [14]. Este método, en lugar de mantener un acumulado de los gradientes, utiliza el concepto de ventana para únicamente considerar en el aprendizaje los gradientes más recientes. Para ello se ha utilizado la función **rmsprop** [15] de Keras.

Todo este procedimiento se realiza utilizando la función **compile** [16] de los modelos de Keras.

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

A continuación, ya se procede a entrenar o ajustar el modelo. Para ello se ha utilizado la función **fit** [17] de los modelos de Keras. Para ello además hay que decidir dos valores importantes en el ajuste de la red: **épocas** y **tamaño del bloque**.

- **Epochs**: Representa el número de iteraciones que debe realizar el algoritmo de entrenamiento sobre el conjunto de datos. Con este valor se puede separar el entrenamiento en varias fases y evaluarlas independientemente. Un valor pequeño no permitirá a la red lo suficiente, mientras que un valor excesivo provocará un sobreaprendizaje.
- **Batch size**: Representa la división que se realiza de los datos para ser procesados en paralelo, un conjunto grande ajustará mejor la red, pero puede llevar un sobreesfuerzo computacional, por lo que se recomienda distribuir correctamente los datos en conjuntos adecuados. (Se ha definido en los generadores).

Finalmente se utilizarán diferentes valores de épocas y un tamaño de bloque de 128. Al utilizar generadores hemos utilizado la función **fit_generator**.

```
start_time <- Sys.time()
history <- model %>%
  fit_generator(
    generator = train_generator_flow,
    validation_data = validation_generator_flow,
    steps_per_epoch = 10,
    epochs = 10
  )

plot(history)
```

4.3.3. Evaluación del modelo

Tras construir el modelo, se procede a evaluar el mismo, para el conjunto de datos de prueba. Para ello se ha utilizado la función **evaluate** [16], la cual nos permite predecir con la red creada el conjunto de datos que le indiquemos, esta además devuelve los valores *loss* y *accuracy* de la forma que se indicó en el proceso de compilación, realizando para el *accuracy* una validación cruzada.

Se utiliza nuevamente la versión asociada a generadores.

```
metrics <- model %>%
  evaluate_generator(test_generator_flow, steps = 1)

end_time <- Sys.time()

message(" loss: ", metrics[1])
message(" accuracy: ", metrics[2])
message(" time: ", end_time - start_time)
```

Por último cabe destacar la utilización de una **matriz de confusión** con el fin de comprobar los resultados mediante las predicciones de los modelos, para ello utilizamos la función **predict_generator** y comprobamos los resultados con el fin también de visualizarlos.

4.3.4. Resultados

Ejecución del modelo con 10 épocas:

Accuracy	Loss
0.5546875	0.665714
0.609375	0.649702
0.59375	0.668440
0.5859375	0.685877
0.6171875	0.637456

Ejecución del modelo con 20 épocas:

Accuracy	Loss
0.6640625	0.614704
0.59375	0.648627
0.59375	0.648626
0.6484375	0.636719
0.59375	0.637085

Ejecución del modelo con 50 épocas:

Accuracy	Loss
0.5859375	0.661067
0.5859375	0.638770
0.5625	0.779608
0.6328125	0.646691
0.6171875	0.629586

Resumen final:

Épocas	Accuracy	Loss	Tiempo
10	0.5921875	0.6614164	2.316241
20	0.61875	0.6371522	4.777219
50	0.596875	0.6711444	11.678452

4.4. Modelo de red neuronal propuesto

4.4.1. Diseño del modelo

Imagen 17: Red Neuronal Propuesta

El modelo propuesto es el de una **Red Neuronal Convolutiva**, que posee una serie de capas de convolución y otras capas de *pooling*, por lo que se necesitará crear un modelo **secuencial**. En esta red añadiremos novedades frente a la presentada inicialmente por el profesor de la asignatura.

Para la definición del modelo secuencial se utiliza la clase **Sequential** [5] perteneciente a los modelos que ofrece Keras. A continuación se añadirán las diferentes capas:

- Una capa de entrada de dimensiones (64x64x3 canales de color) con una convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 32, con una función de activación **RELU** [6] conocida como *Rectified Linear Unit*, consistente en que si la salida que realiza la operación es positiva, se activa y emite el valor positivo, si es negativa devuelve un 0 (no se activa). Para ello se utiliza la capa **Conv2D** [7] de las layers de Keras.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 32, con una función de activación RELU.
- Una capa de *pooling* de 2 dimensiones, que reduce el tamaño de la muestra tomando el valor máximo de una ventana definida como valor de representación de la ventana, en nuestro caso de un tamaño de 2x2. Para ello se utiliza la capa **MaxPooling2D** [8] de las layers de Keras.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 64, con una función de activación RELU.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 64, con una función de activación RELU.
- Una capa de pooling de 2 dimensiones, que utiliza una ventana de 2x2.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 128, con una función de activación RELU.
- Una capa de convolución de 2 dimensiones, que utiliza una ventana/kernel de 3x3, con un valor de filtrado de 128, con una función de activación RELU.
- Una capa de pooling de 2 dimensiones, que utiliza una ventana de 2x2.
- Una capa de **normalización de batch** [18] con un factor epsilon de 0.001 predeterminado utilizando para ello la capa **batch** [19] de Keras.

- Una capa de *flatten* o *aplanamiento*, que reduce las dimensiones. Para ello se utiliza la capa **Flatten** [9] de las layers de Keras.
- Una capa de conexión de neuronas de 512, donde todas las neuronas se encuentran conectadas y se limitan a definirse en diferentes clases. Para ello se utiliza una capa **Dense** [10]. Esta capa sigue la estructura *Dense(units, activation)* realizando la operación *output=activation(dot(input,kernel) + bias)*, donde la función de activación indica si se activa la neurona y units representa el número de unidades de salida de la capa.
- Una capa de *dropout* o de regularización, donde se le indica un porcentaje de exclusión de neuronas para evitar el *overfitting*, con un ratio de 0,2. Para ello se utiliza la capa **Dropout** [20] de las layers de Keras.
- Una capa de conexión de neuronas de 256, donde todas las neuronas se encuentran conectadas y se limitan a definirse en diferentes clases, mediante una capa **Dense**.
- Una capa de *dropout* o de regularización, donde se le indica un porcentaje de exclusión de neuronas para evitar el *overfitting*, con un ratio de 0,2.
- Una capa de conexión de neuronas de 128, donde todas las neuronas se encuentran conectadas y se limitan a definirse en diferentes clases, mediante una capa **Dense**.
- Una capa de *dropout* o de regularización, donde se le indica un porcentaje de exclusión de neuronas para evitar el *overfitting*, con un ratio de 0,2.
- Una capa de salida, que recibe la información de la capa previa y realiza una operación de softmax que convierte un vector real en un vector de probabilidades categóricas. Para ello se utiliza una capa Dense que utiliza como función de activación la función **softmax** [11] y como número de unidades de salida 2 unidades (que representan las dos clases existentes de clasificación).

El código asociado a la definición del modelo es el siguiente

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
      input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  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_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_batch_normalization() %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 2, activation = "softmax")
```

4.4.2. Entrenamiento del modelo

Para poder entrenar el modelo, primero es necesario configurar las métricas que este va a emplear para evaluar tanto el *accuracy* como el *loss*, e indicar el método de aprendizaje, también llamado *optimizador*.

Para el cálculo de la pérdida (*loss*) se ha utilizado la **entropía cruzada** [12] mediante la función **categorical_crossentropy** [13].

Por otro lado, se han utilizado distintos algoritmos de aprendizajes, pero finalmente se ha decidido mantener el algoritmo de **Adam** [21]. Este método, supone la combinación del algoritmo de **rmsprop** [14] junto a la aplicación del **gradiente descendente estocástico** [22] con **Momentum** [23]. Para ello se ha utilizado la función **adam** [24] de Keras. Se ha escogido un valor de tasa de aprendizaje de 0.005.

Todo este procedimiento se realiza utilizando la función **compile** [16] de los modelos de Keras.

```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_adam(lr=0.005),
  metrics = c('accuracy')
)
```

Finalmente se utilizarán diferentes valores de épocas y un tamaño de bloque de 128. Al utilizar generadores hemos utilizado la función **fit_generator**.

```
start_time <- Sys.time()
history <- model %>%
  fit_generator(
    generator = train_generator_flow,
    validation_data = validation_generator_flow,
    steps_per_epoch = 10,
    epochs = 10
  )

plot(history)
```

4.4.3. Evaluación del modelo

Tras construir el modelo, se procede a evaluar el mismo, para el conjunto de datos de prueba. Para ello se ha utilizado la función **evaluate** [16], la cual nos permite predecir con la red creada el conjunto de datos que le indiquemos, esta además devuelve los valores *loss* y *accuracy* de la forma que se indicó en el proceso de compilación, realizando para el *accuracy* una validación cruzada.

Se utiliza nuevamente la versión asociada a generadores.

```
metrics <- model %>%
  evaluate_generator(test_generator_flow, steps = 1)

end_time <- Sys.time()

message(" loss: ", metrics[1])
message(" accuracy: ", metrics[2])
message(" time: ", end_time - start_time)
```

Por último cabe destacar la utilización de una **matriz de confusión** con el fin de comprobar los resultados mediante las predicciones de los modelos, para ello utilizamos la función **predict_generator** y comprobamos los resultados con el fin también de visualizarlos.

4.4.4. Resultados

Ejecución del modelo con 10 épocas:

Accuracy	Loss
0.6796875	0.629045
0.625	0.641731
0.6640625	0.831907
0.6484375	0.732690
0.6171875	1.082894

Ejecución del modelo con 20 épocas:

Accuracy	Loss
0.5390625	1.247634
0.5625	0.962514
0.6484375	0.671609
0.5859375	0.952880
0.546875	0.828652

Ejecución del modelo con 50 épocas:

Accuracy	Loss
0.6188195	0.668192
0.625	0.666235
0.6480605	0.658890
0.6758755	0.651241
0.6868335	0.651002

Resumen final:

Épocas	Accuracy	Loss	Tiempo
10	0.646875	0.7836534	3.3259195
20	0.5765625	0.932657928	6.72148423
50	0.6509178	0.659112	16.311905

5. Mejoras de aprendizaje

5.1. Normalización por lotes (Batch)

5.1.1. Razonamiento

En este punto el objetivo es comprobar como afecta la normalización por lotes (**Batch normalization**) con la capa definida en Keras (explicada previamente). Para ello vamos a coger la red definida previamente y añadir una serie de normalizaciones por lote y comprobar como afectan a los resultados obtenidos y si mejoran los existentes.

La normalización por lotes básicamente implica agregar un paso adicional entre la neurona y la función de disparo para normalizar el disparo de salida. Idealmente, la media y la varianza de todo el conjunto de entrenamiento se usarán para la normalización, pero si aplicamos el descenso de gradiente estocástico para entrenar la red, se usará la media y la varianza de cada mini-lote de entrada.

Por lo tanto la implementación del modelo sería la siguiente:

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
      input_shape = c(64, 64, 3)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), 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(3, 3), activation = "relu") %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization(epsilon = 0.01) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_batch_normalization(epsilon = 0.01) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_batch_normalization() %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 2, activation = "softmax")
```

En cuanto a los resultados, se ha utilizado una configuración de tamaño de lote de 128 y una cantidad de 50 épocas, por mantener la uniformidad con pruebas anteriores.

5.1.2. Resultados

Accuracy	Loss
0.617185	0.660388
0.6484375	0.802477
0.6640625	0.660637
0.625	0.816283
0.635	2.391253

Finalmente, se obtienen los promedios:

Accuracy	Loss	Time
0.637937	1.0662076	22.1423626820246

5.2. Aumento de datos

5.2.1. Razonamiento

Otra de las técnicas más conocidas es la de **aumento de datos** [25], pero ante esto se nos plantea la pregunta ¿Cómo podemos lograr aumentar los datos si los que poseemos son finitos? Aquí entran técnicas de aumento de datos como son transformaciones sobre las imágenes, con el fin de crear imágenes muy similares pero a su vez diferentes y aumentar la cantidad de datos.

Cabría destacar la influencia de esta técnica pero razonando podemos encontrar una ventaja y una desventaja a simple vista:

- Ventaja: Poseemos más datos para entrenar, y en conjuntos con poca variedad de imágenes podemos añadir una gran variedad.
- Desventaja: Por el contrario, en conjuntos con una gran variedad de imágenes y donde no se sigan patrones simples en primera instancia, puede producir más bien un sobreajuste que realmente ayudar en su propósito.

En este problema, hemos permitido un rango determinado de rotación, posibles deformaciones mínimas que no estropeen la imagen, incluso voltear imágenes de forma que la variedad no aumente demasiado, ya que entendiendo el contexto del problema, esto podría conllevar a un problema de excesiva complejidad y malos resultados.

La implementación de esta técnica sería la siguiente:

```
train_images_generator <- 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"
)
```

5.2.2. Resultados

Accuracy	Loss
0.659375	1.077147
0.65625	0.650778
0.6415625	0.738627
0.68375	0.693101
0.6571875	0.669298

Finalmente, obtenemos los promedios:

Accuracy	Loss	Time
0.659625	0.7657902	20.235021619002

5.3. Ambas técnicas

5.3.1. Razonamiento

En esta sección el objetivo será comprobar si ambas técnicas pueden colaborar de forma que se optimicen los resultados y realmente se llegue a mejorar el enfoque inicial propuesto de forma que se pueda aprovechar la funcionalidad de cada una de las técnicas de mejora del aprendizaje propuestas.

5.3.2. Resultados

Accuracy	Loss
0.6328125	0.812493
0.6640625	0.646804
0.6640625	5.763518
0.625	1.331141
0.6328125	1.567976

Finalmente, obtenemos los promedios:

Accuracy	Loss	Time
0.64375	2.0243864	22.2698103149732

6. Técnicas extra

En esta sección observaremos dos de las técnicas comentadas en la asignatura para tratar de mejorar los resultados obtenidos previamente, aunque como es evidente, no se puede garantizar como tal, ya que esto depende del problema y su contexto.

6.1. Transferencia de aprendizaje

6.1.1. Razonamiento

En esta técnica, el enfoque consiste en obtener redes entrenadas previamente de forma que el aprendizaje de dichas redes pueda servir realmente de ayuda a la hora de clasificar en nuestro problema.

Tras probar una diferente series de redes, hemos decidido finalmente quedarnos con la red **VGG16**, la cual posee la siguiente forma:

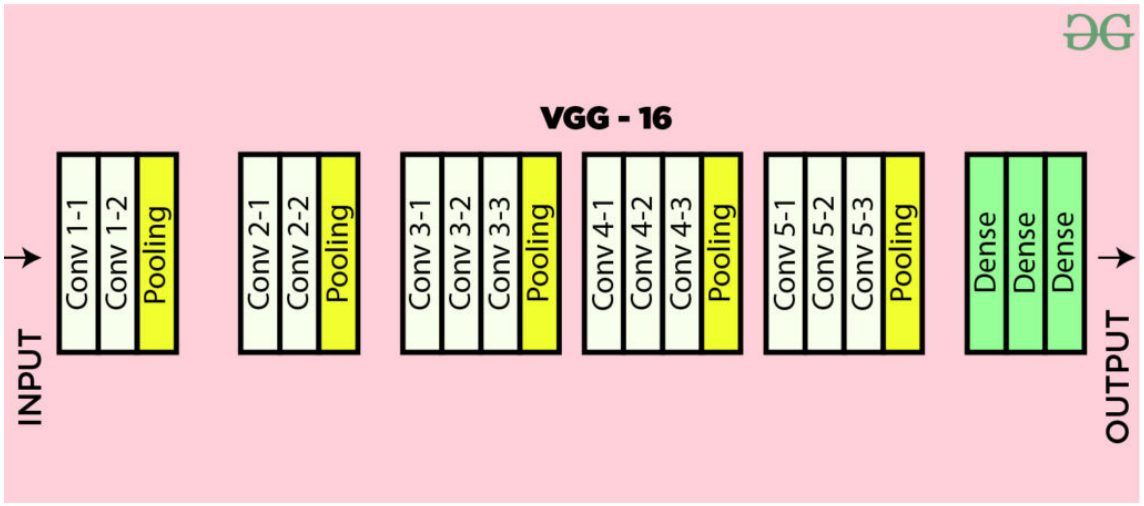

Imagen 18: Red VGG16 [26]

Para ello probaremos la red tal cual está definida para los pesos de **Imagenet** [27], y posteriormente liberaremos varias de las capas con el fin de comprobar si se podría aprovechar dicha red para este fin.

Por lo tanto, la implementación para la red **VGG16** sería la siguiente:

```
conv_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64, 64, 3)
)

freeze_weights(conv_base)

model <- keras_model_sequential() %>%
  conv_base %>%
  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

Por otro lado, la implementación que entrena ciertas capas de la red sería la siguiente:

```
unfreeze_weights(conv_base, from = "block2_conv1")
unfreeze_weights(conv_base, from = "block3_conv1")
```

6.1.2. Resultados con capas congeladas

Accuracy	Loss
0.6749995	0.676166
0.6440625	0.662526
0.6246875	0.677132
0.689375	0.663263
0.5921875	0.670350

Finalmente, se realiza el promedio:

Accuracy	Loss	Tiempo
0.6450624	0.6698874	30.2538813352585

6.1.3. Resultados con entrenamiento de capas

Accuracy	Loss
0.6125	0.643228
0.615625	0.650472
0.6778125	0.646939
0.6234375	0.637984
0.643125	0.646309

Finalmente, se realiza el promedio:

Accuracy	Loss	Tiempo
0.6345	0.6449864	66.7057226705

6.2. Ensamblamiento de diferentes redes

6.2.1. Razonamiento

El **ensamblamiento** [28] de redes neuronales es una de las técnicas empleadas ya que las redes neuronales son métodos no lineales, y esto se puede traducir en una gran varianza.

Tratando de paliar este efecto en los diferentes modelos, y con el fin de mejorar los resultados a obtener, se decide utilizar esta técnica que básicamente consiste en tomar diferentes modelos de redes neuronales y realizar un ensamblado de las mismas, tratando de reducir la varianza de las mismas.

Tras realizar pruebas con diferentes configuraciones, no hemos conseguido grandes resultados al ensamblar nuestra red propia con otras ya existentes, por lo que con el fin de mostrar la técnica hemos decidido mantener las dos redes que se ofrecen por defecto en la asignatura (VGG16 y MobileNet) y ver como se comportan en nuestro problema.

La implementación para ello es la siguiente:

```
vgg16_base <- application_vgg16(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64, 64, 3)
)

mobile_base <- application_mobilenet_v2(
  weights = "imagenet",
  include_top = FALSE,
  input_shape = c(64, 64, 3)
)

freeze_weights(vgg16_base)
freeze_weights(mobile_base)

model_input <- layer_input(shape=c(64, 64, 3))

model_list  <- c(vgg16_base(model_input) %>% layer_flatten(),
                 mobile_base(model_input) %>% layer_flatten())
```

6.2.2. Resultados

Accuracy	Loss
0.676875	0.653509
0.6215625	0.673077
0.659375	0.684830
0.5921875	0.676067
0.628125	0.685192

Finalmente, se realiza el promedio:

Accuracy	Loss	Tiempo
0.635625	0.674535	41.66134317

7. Ficheros del proyecto

Para facilitar la comprensión del proyecto y las diferentes ejecuciones propuestas, se ha descompuesto el problema en diferentes ficheros que son los siguientes:

- **p2-eda.Rmd**: Este fichero *Rmarkdown* contiene toda la información del análisis exploratorio tanto inicial proporcionado por el profesor como el propio extendido.
- **p2-inicial.R**: Este fichero es básicamente una adaptación del modelo propuesto por el profesor de la asignatura y adaptado para el conjunto de datos con el que se trabaja.
- **p2-modelo.R**: Este fichero contiene el modelo definido para el proyecto con sus diferentes parametrizaciones pero sin ninguna técnica añadida.
- **p2-batch.R**: Este fichero contiene el modelo propuesto haciendo uso de la técnica de normalización por lotes o *Batch Normalization*.
- **p2-augmentation.R**: Este fichero contiene el modelo propuesto haciendo uso de la técnica de aumento de datos o *Data Augmentation*.
- **p2-batch-augmentation.R**: Este fichero contiene el modelo propuesto haciendo uso de las técnicas de *Batch Normalization* y *Data Augmentation*.
- **p2-transfer.R**: Este fichero contiene el modelo con uso de la técnica de transferencia de aprendizaje y *Fine tuning*.
- **p2-ensemble.R**: Este fichero contiene el modelo con uso de la técnica de ensamblado de modelos.

8. Resultados finales

Modelo	**Accuracy**	**Loss**	**Tiempo**
Inicial	0.596875	0.6711444	11.678452
Propuesto	0.6509178	0.659112	16.311905
Propuesto-Batch	0.637937	1.0662076	22.1423626820246
Propuesto-DataAugmentation	0.659625	0.7657902	20.235021619002
Propuesto-Batch-DataAug	0.64375	2.0243864	22.2698103149732
VGG16-Congelado	0.6450624	0.6698874	30.2538813352585
VGG16-Entrenamiento	0.6345	0.6449864	66.7057226705
Ensamblado	0.635625	0.674535	41.66134317

Se puede observar que finalmente el modelo con mejores resultados obtenidos es el propio utilizando la técnica de mejora del aprendizaje de **aumento de datos**. Esto puede deberse a que realmente el conjunto de datos posee información variable según interpretación por lo que un modelo exacto quizás tenga determinados problemas para adaptarse y la ampliación suponga una gran mejora.

Por otro lado, examinando los diferentes tiempos, está claro que los modelos más complejos, aunque pueden mejorar el resultado inicial, no llegan a compensar debido a la cantidad de tiempo y exigencia computacional que requieren.

Se puede observar que modelos más complejos, no tienen por qué obtener mejores resultados y que la entroia cruzada, si bien es indicativa, no llega a ser determinista en este problema.

Finalmente se observa que el modelo propuesto se adapta considerablemente mejor al problema que el modelo inicial proporcionado, por lo que consideramos que es más adecuado y la diferencia de tiempo llega a ser despreciable.

9. Conclusiones

Tras realizar el proyecto hemos podido comprobar una serie de aspectos dentro del conjunto de datos:

- Uno de los principales problemas que encontramos en la división de los datos ofrecida es el gran desbalanceo existente en los datos, quizás en otro contexto podríamos haber realizado un preprocesamiento con técnicas de balanceo de datos para obtener mejores resultados.
- Se puede ver una clara tendencia a la desinformación, por lo que realmente es un problema complejo de tratar, ya que la veracidad de las noticias, por lo que hemos podido observar, es más bien infrecuente en este tipo de foros.
- Se puede observar una clara tendencia a que los artículos sean de desinformación cuando poseean una mayor actividad en los comentarios del mismo y cuando las valoraciones positivas son altas.
- Se puede observar claramente que este tipo de foro no está orientado a ofrecer noticias reales, sino a realizar procesos de desinformación tales como noticias con fines malintencionados, parodias u otro tipo de actividades menos serias que la información como tal.

Tras realizar el problema, en cuanto a las cuestiones técnicas, se analizan los siguientes puntos:

- Realmente las herramientas que poseemos son muy potentes y flexibles para este tipo de problemas y ofrecen una gran variedad de alternativas con el fin de resolerlos, si bien en este caso, dadas las circunstancias tanto a nivel computacional como a nivel conjuntos de datos los resultados no son los más deseables.
- Se puede observar que abordar un problema de reconocimiento de imágenes no es trivial ya que requiere no solo el procesamiento de los datos, sino un análisis y transformación de los mismos, ya que los datos no son simplemente números y necesitamos en este problema la definición de flujos por cuestiones de escasez de memoria y el procesamiento y reescalado de los mismos, lo cual no se había visto con anterioridad en otros proyectos realizados.

A continuación, en cuanto a las decisiones tomadas, podemos observar que (considerando que tomamos como fuente de verdad una serie de ejecuciones aisladas en un ámbito no determinístico) existe una tendencia a mejorar los resultados con una red que posee un mayor número de convoluciones que no utilizan unos filtros muy altos, por lo que con un kernel de 3x3 es más que suficiente para abordar un problema de tal tamaño.

Hemos probado diferentes configuraciones pero al final nos hemos decantado por tratar de simplificar el modelo, tratando de evitar posibles sobreajustes con técnicas como la normalización por lotes o el dropout. Finalmente el modelo que poseemos obtiene mejores resultados sin crecer en el tiempo de ejecución una cantidad que consideremos inviable.

También hemos podido observar que las diferentes técnicas de mejora del aprendizaje no tienen por qué mejorar los resultados por norma general, si bien en ciertos casos son convenientes y sirven de gran ayuda, como el caso de el aumento de datos para nuestro modelo. Y podemos observar también que una red más compleja o incluso entrenada (véase **VGG16**) no tiene por que producir mejores resultados.

Tras analizar toda la información previa, podemos establecer unas conclusiones finales:

- Cada problema de **Deep Learning** requiere un estudio previo, específico y amplio sobre el contexto del problema y los datos que se poseen, ya que las herramientas propuestas son útiles, pero no universales.
- Las herramientas utilizadas como **keras** y **tensorflow** suponen un gran avance y la flexibilidad a la hora de configurar redes neuronales donde poder centrarnos en mayor forma en el problema más que en la implementación, resultando bastante sencilla.
- El problema propuesto no es un problema trivial ya que requiere estudiar tanto el conjunto de datos, saber como tratarlos, como el comportamiento de las diferentes redes que configuremos para poder solventarlo.

Finalmente, nos gustaría señalar la importancia de este tipo de técnicas en problemas relacionados con la detección de diferentes patrones dentro de problemas o ámbitos de diferentes plataformas informativas o de tipo foro de entretenimiento, para poder evitar usos malintencionados (considerando únicamente con este fin y no posibles parodias) y poder advertir a los diferentes usuarios que la información mostrada no es real.

10. Bibliografía

[1] Reddit. https://www.reddit.com/.

[2] Kai Nakamura, Sharon Levy, and William Yang Wang. r/fakeddit: A new multimodal benchmark dataset for fine-grained fake news detection. *arXiv preprint arXiv:1911.03854*, 2019.

[3] Tensorflow. https://www.tensorflow.org/.

[4] Keras. https://keras.io/.

[5] Keras. The sequential class. https://keras.io/api/models/sequential/.

[6] Wikipedia. Rectificador (redes neuronales). https://es.wikipedia.org/wiki/Rectificador_(redes_neuronales).

[7] Keras. Conv2d layer. https://keras.io/api/layers/convolution_layers/convolution2d/.

[8] Keras. Maxpooling2d layer. https://keras.io/api/layers/pooling_layers/max_pooling2d/.

[9] Keras. Flatten layer. https://keras.io/api/layers/reshaping_layers/flatten/.

[10] Keras. Dense layer. https://keras.io/api/layers/core_layers/dense/.

[11] Keras. Layer activation functions. https://keras.io/api/layers/activations/.

[12] Wikipedia. Entropía cruzada. https://es.wikipedia.org/wiki/Entrop%C3%ADa_cruzada.

[13] Keras. Probabilistic losses. https://keras.io/api/losses/probabilistic_losses/#categorical_crossentropy-function.

[14] Andrea Perlato. Root mean square propagation. https://www.andreaperlato.com/aipost/root-mean-square-propagation/.

[15] Keras. Rmsprop. https://keras.io/api/optimizers/rmsprop/.

[16] Keras. The model class. https://keras.io/api/models/model/.

[17] Keras. Model training apis. https://keras.io/api/models/model_training_apis/.

[18] Jaime Durán. Técnicas de regularización básicas para redes neuronales. https://medium.com/metadatos/t%C3%A9cnicas-de-regularizaci%C3%B3n-b%C3%A1sicas-para-redes-neuronales-b48f396924d4#:~:text=Normalizaci%C3%B3n%20por%20lotes%20(Batch%20normalization)&text=La%20normalizaci%C3%B3n%20en%20lotes%20consiste,normalizar%20las%20activaciones%20de%20salida.

[19] Keras. Batchnormalization layer. https://keras.io/api/layers/normalization_layers/batch_normalization/.

[20] Keras. Dropout layer. https://keras.io/api/layers/regularization_layers/dropout/.

[21] Vitaly Bushaev. Adam — latest trends in deep learning optimization. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

[22] Wikipedia. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[23] Wikipedia. Momentum. https://en.wikipedia.org/wiki/Momentum.

[24] Keras. Adam. https://keras.io/api/optimizers/adam/.

[25] Arun Gandhi. Data augmentation | how to use deep learning when you have limited data. https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/.

[26] Geeksforgeeks. Vgg-16 | cnn model. https://www.geeksforgeeks.org/vgg-16-cnn-model/.

[27] Imagenet. https://www.image-net.org/.

[28] Jason Brownlee. Ensemble learning methods for deep learning neural networks. https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/.