ugr | Universidad de Granada

PRÁCTICA 2:

DEEP LEARNING PARA MULTI-CLASIFICACIÓN

Máster Profesional en Ingeniería Informática

Sistemas Inteligentes para la Gestión en la Empresa

Autores:

Freddy Javier Frere Quintero

Ramón Gago Carrera

02/06/2018

ÍNDICE:

1. INTRODUCCIÓN

Como introducción a esta práctica se debe definir en primer lugar el término red neuronal **[1]**.

Las redes neuronales son un campo de estudio fundamental dentro de la Inteligencia Artificial. Se inspiran en el comportamiento hasta ahora conocido del cerebro humano (en su mayoría el referido a las neuronas y sus conexiones) y trata de crear modelos artificiales que solucionen problemas difíciles de resolver mediante técnicas algorítmicas convencionales.

Se entiende, por tanto, que una red neuronal se trata de un procesador distribuido en paralelo de forma masiva que tiene una propensión a almacenar contenidos de conocimiento experimental para después poder proveerse del mismo, semejando así el funcionamiento del cerebro humano.

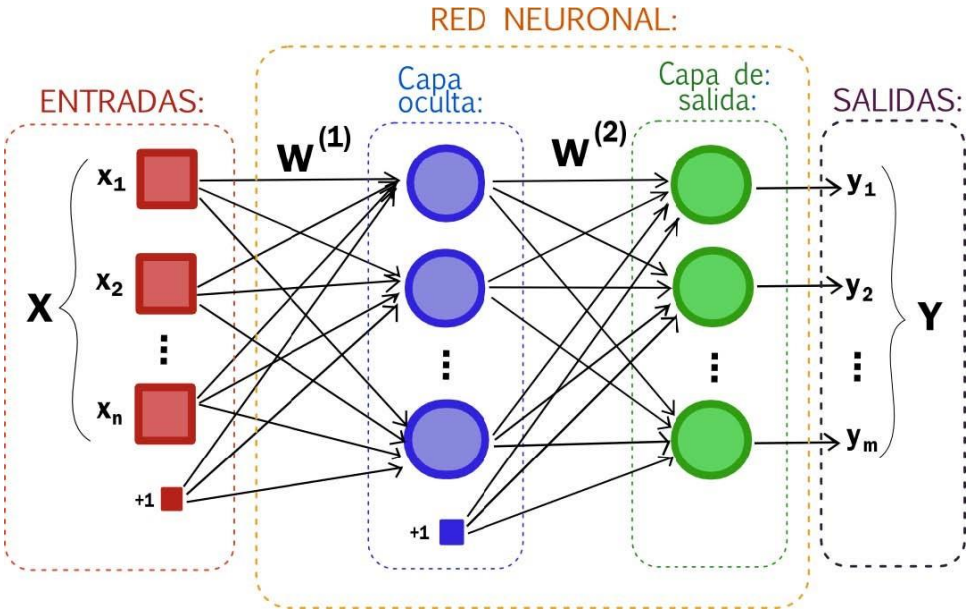

Figura 1: Red Neuronal

Esto se puede afirmar debido a que el conocimiento que adquiere la red, se realiza mediante un proceso de aprendizaje que dados unos parámetros existe una forma de combinarlos para predecir un cierto resultado. Por ejemplo, en este caso sabiendo los píxeles de una imagen se realiza una predicción para saber a qué subtipo de células pertenecen.

Estas redes forman parte de la inteligencia artificial, pues su respuesta ante estímulos es lo que propone una idea de inteligencia y, aunque aún se está muy lejos de poder otorgar la inteligencia real a una máquina, las posibilidades se acrecientan cada año a pasos agigantados gracias a la inteligencia artificial.

2. FUNDAMENTOS TEÓRICOS

En esta práctica se tratará de resolver el problema asociado a la clasificación de imágenes obtenidas de muestras de sangre.

Este problema obtenido a través de la plataforma de Kaggle **[2]**, pretende que se desarrollen distintos algoritmos de optimización de redes neuronales que consigan clasificar con el mayor grado de acierto el subtipo de célula que aparece en las muestras de sangre. Estos subtipos de células son los siguientes:

- **Eosinófilos:**

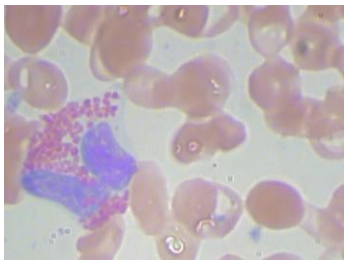

Figura 2: Eosinófilos

- **Linfocitos:**

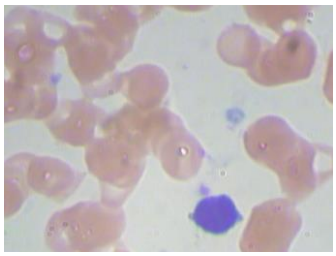

Figura 3: Linfocitos

- **Monocitos:**

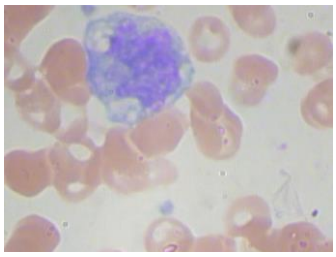

Figura 4: Monocitos

- **Neutrófilos:**

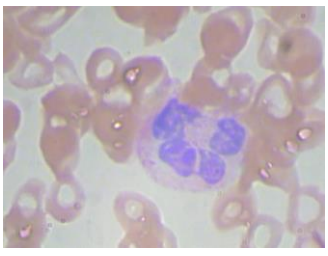

Figura 5: Neutrófilos

Este conjunto de datos contiene 12.500 imágenes aumentadas de células sanguíneas (JPEG) con etiquetas de tipo celular (CSV). Hay aproximadamente 3.000 imágenes para cada uno de los 4 tipos de células diferentes agrupadas en 4 carpetas diferentes.

En esta práctica se aplicarán técnicas de aprendizaje profundo (Deep Learning) el cual se refiere a redes neuronales con múltiples capas ocultas que pueden aprender representaciones cada vez más abstractas de los datos de entrada.

En general, cualquier técnica de Machine Learning **[3]** trata de realizar la asignación de entradas (por ejemplo, imágenes) a salidas objetivo (por ejemplo, la etiqueta "Linfocito"), mediante la observación de un gran número de ejemplos de entradas y salidas. El Deep Learning realiza este mapeo de entrada-objetivo por medio de una red neuronal artificial que está compuesta de un número grande de capas dispuestas en forma de jerarquía. La red aprende algo simple en la capa inicial de la jerarquía y luego envía esta información a la siguiente capa. La siguiente capa toma esta información simple, lo combina en algo que es un poco más complejo, y lo pasa a la tercera capa. Este proceso continúa de forma tal que cada capa de la jerarquía construye algo más complejo de la entrada que recibió de la capa anterior. De esta forma, la red irá aprendiendo por medio de la exposición a los datos de ejemplo.

Debe tenerse en cuenta que las primeras capas ocultas solo pueden aprender patrones de bordes locales. Luego, cada capa (o filtro) posterior aprende representaciones más complejas.

Finalmente, la última capa puede clasificar las imágenes según el problema concreto que se está abordando. A este tipo de redes neuronales profundas se llaman redes neuronales convolucionales (CNN).

Las CNN son redes neuronales multicapa (en este caso se han utilizado 13 capas) que suponen que los datos de entrada son imágenes y su funcionamiento es el siguiente:

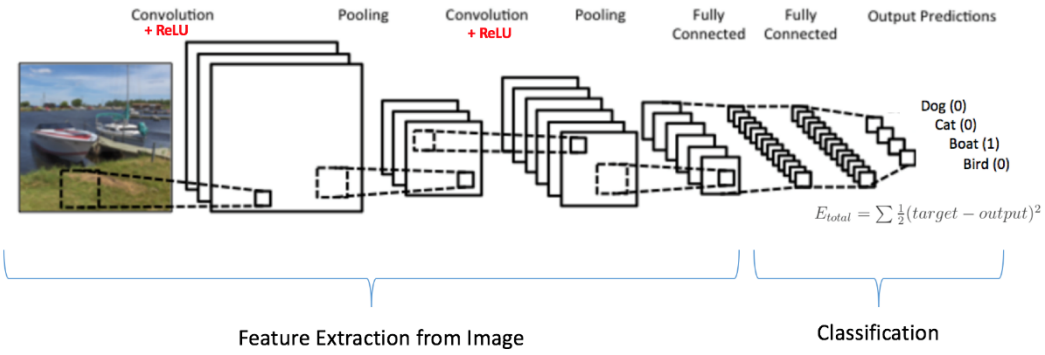

Figura 6: Ejemplo de CNN multicapa

La especificación de lo que cada capa hace a la entrada que recibe es almacenada en los pesos de la capa, que, en esencia, no son más que números. Se puede decir que la transformación de datos que se produce en la capa es parametrizada por sus pesos. Para que la red aprenda debemos encontrar los pesos de todas las capas de forma tal que la

red realice un mapeo perfecto entre los ejemplos de entrada con sus respectivas salidas objetivo. Pero el problema reside en que una red de Deep Learning puede tener millones de parámetros, por lo que encontrar el valor correcto de todos ellos puede ser una tarea realmente muy difícil, especialmente si la modificación del valor de uno de ellos afecta a todos los demás.

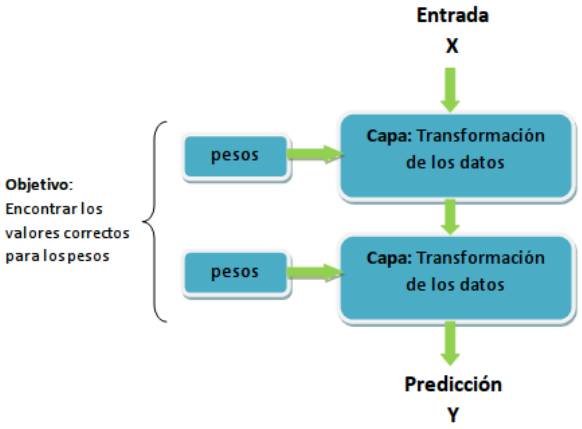

Figura 7: Modificación de los pesos

En el apartado 3 se llevará a cabo la descripción de la red empleada para intentar resolver el problema propuesto. Sin embargo, es en este apartado donde se hará una introducción de los conceptos clave que han supuesto unos buenos resultados que se comentarán más adelante.

Tras los ajustes de la topología de la red y las múltiples pruebas realizadas con distintos hiperparámetros se llegó a la conclusión de que además de la construcción correcta de la CNN los parámetros clave para la obtención de buenos resultados eran dos:

- La función de pérdida
- El optimizador

Con esta información se procedió a la investigación de ambos parámetros.

2.1. Función de pérdida

Una función de pérdida (o función objetivo, o función de puntuación de optimización) es uno de los dos parámetros necesarios para compilar un modelo junto con el optimizador.

Para controlar la salida de la red neuronal, se debe medir la distancia de salida obtenida con respecto a la esperada. Este es el trabajo de la función de pérdida de la red. Esta función utiliza las predicciones que realiza el modelo y los valores objetivos (lo que se espera que la red prediga), y calcula la distancia a ese valor, de esta manera, se puede saber con certeza la exactitud del modelo. Para ello se utiliza el valor que devuelve esta función de pérdida para retroalimentar la red y ajustar los pesos en la dirección que

vayan reduciendo la pérdida del modelo para cada ejemplo. De este ajuste, se encarga el optimizador, el cuál implementa la propagación hacia atrás.

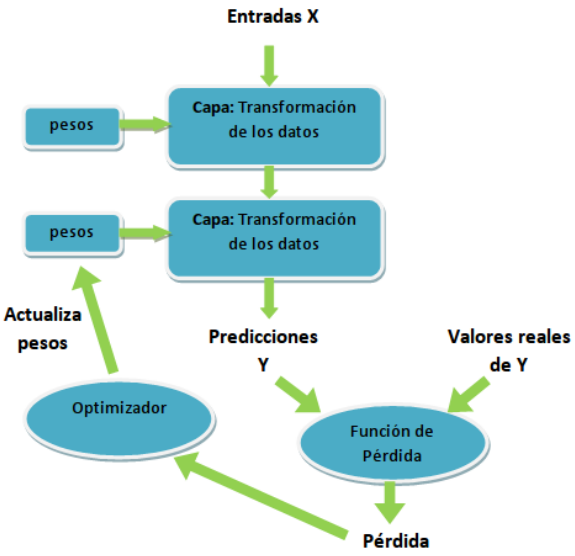

Figura 8: Función de pérdida

Por tanto, el funcionamiento sería el siguiente: inicialmente, los pesos de cada capa son asignados en forma aleatoria, por lo que la red simplemente implementa una serie de transformaciones aleatorias. En este primer paso, obviamente la salida del modelo dista bastante del ideal que se desea obtener, por lo que el valor de la función de pérdida va a ser bastante alto. Pero a medida que la red va procesando nuevas imágenes, los pesos se van ajustando de tal forma que se va reduciendo cada vez más el valor de la función de pérdida. Este proceso es el que se conoce como entrenamiento de la red, el cual, repetido una suficiente cantidad de veces, logra que los pesos se ajusten a los que minimizan la función de pérdida. Una red que ha minimizado la pérdida es la que logra los resultados que mejor se ajustan a las salidas objetivo, es decir, que el modelo se encuentra entrenado.

2.2. Algoritmos de optimización

El uso técnico de los algoritmos de optimización sobre la CNN utilizada se detallará en el siguiente apartado. Sin embargo, en este subapartado se tratarán a nivel teórico los algoritmos, que tras diversas pruebas, han dado mejor resultado. Aunque el utilizado por excelencia es el algoritmo de Gradiente Descendiente Estocástico (SGD, por sus siglas en inglés), que popularmente se impone sobre los demás por su facilidad y extensión de uso, hay muchos otros que pueden resultar interesantes, presentando diferentes características y rendimiento que los pueden hacer más, o menos, adecuados dependiendo de las características del problema concreto al que nos enfrentemos.

Para explicar estos algoritmos de optimización **[4]** en primer lugar debe destacarse que el problema de aprendizaje en las redes neuronales se formula en términos de la

minimización de la función de error (o pérdida) asociada, y que se notará en esta entrada por medio de "*f*".

Normalmente, esta función está compuesta por dos términos, uno que evalúa cómo se ajusta la salida de la red neuronal al conjunto de datos de que disponemos, y que se denomina término de error, y otro que se denomina término de regularización, y que se utiliza para evitar el sobreaprendizaje por medio del control de la complejidad efectiva de la red neuronal. Sobre el sobreaprendizaje se volverá a hablar más adelante ya que también se han aplicado métodos como el dropout para evitarlo.

Por supuesto, el valor de la función de error depende por completo de los parámetros de la red neuronal. En este sentido, podemos escribir *"f (w)"* para indicar que el valor del error que comete la red neuronal depende de los pesos asociados a la misma. El objetivo es encontrar el valor "*w*"* para el que se obtiene un mínimo global de la función "*f*", convirtiendo el problema de aprendizaje en un problema de optimización.

En general, la función de error es una función no lineal, por lo que no se dispone de algoritmos sencillos y exactos para encontrar sus mínimos. En consecuencia, se deberá hacer uso de una búsqueda a través del espacio de parámetros que, idealmente, se aproxime de forma iterada a un error mínimo de la red para los parámetros adecuados.

Aunque el proceso de optimización es inherentemente multidimensional (ya que siempre tendremos más de un parámetro en la función de error), muchos de los algoritmos de optimización que se utilizan en el entrenamiento de redes neuronales hacen uso de una optimización unidimensional

A continuación, se explicarán algoritmos de optimización más usuales para redes neuronales. La mayoría hacen uso de aproximaciones numéricas basadas en propiedades diferenciales de la función de error, concretamente, aquellos basados en el método del Descenso del Gradiente. En todo caso, solo se describirán los que se han utilizado y no las variadas implementaciones ni mejoras que se han realizado y que suelen tener a alguno de ellos **[5]**.

2.2.1. Gradiente Descendiente Estocástico (SGD):

El Descenso del Gradiente Estocástico es uno de los algoritmos de entrenamiento más simple y también más extendidos y conocidos **[6]**.

El punto clave está en que reevalúa todo el conjunto de datos cada vez que se realiza una época, lo cual propicia que, por un lado, los pesos se actualicen con más frecuencia (provocando una convergencia más rápida) y por otro, que al usar una sola muestra da menos precisión a la actualización, así que se introduce más ruido en los pesos. Esto no es necesariamente malo, ya que facilita salir de mínimos locales que no se corresponden con la minimización de nuestra función de costos. Otra ventaja de este algoritmo es que resulta adecuado para el aprendizaje online, es decir, aquel en el que el modelo tiene que adaptarse rápidamente a cambios en los datos.

Al contrario que Gradiente Descendente de Lote, que realiza cálculos redundantes para grandes conjuntos de datos, recalculando los gradientes para ejemplos similares antes de cada actualización de parámetro. SGD elimina esta redundancia al realizar una actualización a la vez. Por lo tanto, generalmente es mucho más rápido y también se puede usar para aprender en línea.

SGD realiza actualizaciones frecuentes con una gran variación que hace que la función objetivo fluctúe mucho como se mostrará en los ejemplos del siguiente apartado.

2.2.2. AdaGrad

Es un algoritmo para la optimización basado en gradiente que adapta la velocidad de aprendizaje a los parámetros, realizando pequeñas actualizaciones (es decir, tasas de aprendizaje bajas) para parámetros asociados con características que se producen con frecuencia, y actualizaciones más frecuentes (es decir, tasas de aprendizaje altas) para parámetros asociados con características poco frecuentes. Por esta razón, es adecuado para tratar con datos dispersos. Adagrad mejoró en gran medida la solidez del SGD y se utilizó para entrenar redes neuronales a gran escala en Google, que, entre otras cosas, facilitó el aprendizaje de reconocimiento de gatos en los videos de Youtube. Por otra parte, Adagrad también se ha utilizado para entrenar las incrustaciones de palabras GloVe, ya que las palabras infrecuentes requieren actualizaciones mucho más grandes que las frecuentes.

Uno de los principales beneficios de Adagrad es que elimina la necesidad de ajustar manualmente la tasa de aprendizaje. La mayoría de las implementaciones usan un valor predeterminado de 0.01 y lo dejan así, lo cual mejora el rendimiento en problemas con gradientes dispersos (por ejemplo, lenguaje natural y problemas de visión por computadora).

Sin embargo, la principal debilidad de Adagrad es su acumulación de los gradientes al cuadrado en el denominador: dado que cada término agregado es positivo, la suma acumulada sigue creciendo durante el entrenamiento. Esto, a su vez, hace que la tasa de aprendizaje se reduzca y eventualmente se vuelva infinitamente pequeña, momento en el cual el algoritmo ya no puede adquirir conocimiento adicional.

Para ello, los siguientes algoritmos tratan de resolver este defecto.

2.2.3. Adadelta

Adadelta es una extensión de Adagrad que busca reducir su ritmo de aprendizaje agresivo y monotónicamente decreciente. En lugar de acumular todos los gradientes cuadrados anteriores, Adadelta restringe la ventana de gradientes pasados acumulados a un tamaño fijo "w".

Así, en vez de almacenar ineficientemente los gradientes cuadrados anteriores, la suma de los gradientes se define recursivamente como un promedio decreciente de todos los

gradientes cuadrados anteriores. Con Adadelta, ni siquiera es necesario establecer una tasa de aprendizaje predeterminada, ya que se ha eliminado de la regla de actualización.

2.2.4. Adam

Adam es un algoritmo de optimización que se puede usar en lugar del clásico SGD para actualizar las ponderaciones de red de forma iterativa en función de los datos de entrenamiento **[7]**.

Se puede decir que aprovecha los beneficios de AdaGrad y RMSProp. En lugar de adaptar las tasas de aprendizaje de parámetros basadas en el primer momento promedio (la media) como en RMSProp, Adam también utiliza el promedio de los segundos momentos de los gradientes (la varianza descentrada).

Específicamente, el algoritmo calcula un promedio móvil exponencial del gradiente y el gradiente cuadrado, y los parámetros beta1 y beta2 controlan las tasas de disminución de estos promedios móviles.

El valor inicial de los promedios móviles y los valores beta1 y beta2 cercanos a 1.0 (recomendado) dan como resultado un sesgo de estimaciones de momento hacia cero. Este sesgo se supera al calcular primero las estimaciones sesgadas antes de calcular las estimaciones corregidas por sesgo.

2.2.5. Rmsprop

RMSprop y Adadelta se desarrollaron de forma independiente en la misma época, debido a la necesidad de resolver disminución radical de la tasa de aprendizaje de Adagrad. De hecho, RMSprop es idéntico al primer vector de actualización de Adadelta.

También divide la tasa de aprendizaje por un promedio exponencialmente decreciente de gradientes cuadrados. Además aporta varias ventajas. Por un lado, es un optimizador muy robusto que tiene información de pseudo curvatura y por otro lado, puede tratar los objetivos estocásticos muy bien, por lo que es aplicable al mini aprendizaje por lotes.

3. DESCRIPCIÓN DE LAS REDES EMPLEADAS

Las redes neuronales convolucionales son muy similares a las redes neuronales ordinarias como el perceptron multicapa. Lo que diferencia a las redes neuronales convolucionales es que suponen explícitamente que las entradas son imágenes, lo que nos permite codificar ciertas propiedades en la arquitectura; permitiendo ganar en eficiencia y reducir la cantidad de parámetros en la red.

En general, las redes neuronales convolucionales van a estar construidas con una estructura que contendrá 3 tipos distintos de capas:

- Una capa convolucional, que es la que le da le nombre a la red.

- Una capa de reducción o de pooling, la cual va a reducir la cantidad de parámetros al quedarse con las características más comunes.
- Una capa clasificadora totalmente conectada, la cual nos va dar el resultado final de la red.

En este apartado se realizará una descripción de la CNN empleada para abordar el problema explicado en apartados anteriores.

Se ha procedido a utilizar el conjunto de datos "Blood Cell Images", para crear un modelo de clasificación de imágenes basado en redes neuronales profundas con la ayuda de las bibliotecas de Keras y TensorFlow.

La estructura de datos principal en Keras es un modelo, que está pensada para facilitar la organización de capas, ya que dichos modelos en Keras son básicamente una secuencia de capas. Además deduce automáticamente la forma de todas las capas después de la primera capa, teniendo que simplemente establecer las dimensiones de entrada para la primera capa.

Una vez creado el modelo con todas sus capas se configura su proceso de aprendizaje. La compilación del modelo utiliza la librería numérica de la que dispone el nivel inferior en este caso, Tensorflow. El backend selecciona automáticamente la mejor forma de representar la red para el entrenamiento, y hacer que las predicciones se ejecuten en el hardware.

Para realizar la implementación se ha tomado como referencia el script proporcionado en las prácticas de la asignatura. A partir de ahí se ha decidido diseñar una red neuronal con la siguiente estructura; se declara un formato de modelo secuencial y a continuación las siguientes capas **[8]**:

```
# Definición de la arquitectura
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation = "relu", input_shape = c(150,150, 3)) %>%
  layer_activation('relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation = "relu") %>%
  layer_activation('relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64,  kernel_size = c(3, 3), activation = "relu") %>%
  layer_activation('relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 4, activation = "softmax")
```

Figura 9: Modelo de la CNN

- **Convolution2D:**

Los primeros 3 parámetros corresponden a la cantidad de filtros de convolución que se usarán, el número de filas en cada kernel de convolución y el número de columnas en cada kernel de convolución, respectivamente, esta capa de entrada establece el parámetro (150,150, 3) que corresponde a profundidad, ancho y alto de la imagen a procesar.

Cabe recalcar que Keras establece automáticamente la forma de entrada como la forma de salida de la capa anterior, pero para la primera capa, tendrá que establecer eso como un parámetro.

- **Activation "Relu":**

Estas capas permiten el paso de todos los valores positivos sin cambiarlos, pero asigna todos los valores negativos a 0, siendo muy sencilla de calcular, por ende es muy rápida.

- **MaxPooling2D:**

Ayuda a reducir el número de parámetros en el modelo, aplicando un filtro de agrupamiento (2 x 2) a través de la capa previa.

- **Flatten:**

Permite aplanar las imágenes, hacerlas de 1 dimensión, es necesario realizar esto antes de pasar los valores de las capas "Convolution2D" a las capas "Dense".

- **Dense:**

El primer parámetro es el tamaño de salida de la capa, teniendo en cuenta que la capa final tiene un tamaño de salida de 4, que corresponde a las 4 clases de imágenes.

- **Dropout:**

Su propósito es el de evitar que ciertas neuronas se adapten demasiado entre sí (haya un sobre aprendizaje) y se le aplica a la capa que le precede, reduciendo significativamente el error. Para cada caso de entrenamiento, se omite aleatoriamente cada neurona oculta con probabilidad 0.4 en este caso, es decir, casi la mitad de las neuronas de las capas ocultas no se utilizan para cada caso del conjunto de entrenamiento.

De esta forma se evita que las neuronas ocultas dependan o confíen demasiado en el trabajo de otras neuronas ocultas de su misma capa. Si una neurona oculta sabe qué otras neuronas ocultas existen, pueden co-adaptarse sobre el conjunto de entrenamiento, pero las co-adaptaciones no funcionarán bien sobre el conjunto de prueba **[9]**.

A continuación, se procede a compilar el modelo y para ello se deben especificar algunas propiedades adicionales requeridas para entrenar la red. En este caso se especifica la función de pérdida (loss). Esta métrica ha sido probada con tres funciones diferentes:

- **"binary_crossentropy"**
- **"categorical_crossentropy"**
- **“mean_squared_logarithmic_error”**

Y finalmente debido a sus buenos resultados se ha decidido utilizar este último como una variación del error cuadrático medio que, aunque suele utilizarse para problemas de regresión, tras consultarlo se utilizó para este problema concreto a pesar de que los

valores absolutos de error o pérdida no conllevan necesariamente mejora de la métrica "accuracy".

Por otro lado, para compilar el modelo también se han realizado distintas pruebas con varios algoritmos de optimización comentados en el apartado anterior.

```
# Compilación del modelo
model %>% compile(
  loss = "mean_squared_logarithmic_error",
  optimizer = 'adadelta',
  #optimizer = 'adam',
  #optimizer = 'adagrad',
  #optimizer = 'sgd',
  #optimizer = optimizer_rmsprop(lr = 1e-4),
  metrics = c("accuracy")
)
```

Figura 10: Compilación del modelo

Una vez realizada la compilación, el modelo está listo para el entrenamiento.

Al entrenar con Keras, se ha comprobado que el valor de "loss" se calcula como el promedio de pérdidas sobre los minilotes y el valor de "val_loss" se calcula con los parámetros obtenidos al final de la época.

Por tanto, "loss" tiende a ser menor que lo que obtendríamos evaluando la pérdida de entrenamiento al final de la época, y puede ser en ocasiones más grande que "val_loss".

Adicionalmente, también se ha observado que las diferencias en las muestras de entrenamiento y validación pueden producir también valores de validación ligeramente menores de entrenamiento.

En cuanto al manejo de los hiperparámetros, para su modificación se ha realizado una búsqueda manual, intentando ajustar los parámetros a base de experimentos de prueba sucesivos, monitorizando el error de entrenamiento y de test. Los principios básicos que tenemos que entender son los de sesgo (rigidez) y varianza (flexibilidad) de los modelos y cómo se comportan estas cantidades cuando cambiamos parámetros, aunque es también importante experiencia e intuición.

Otra parte importante de la red es el "data aumentation", encargada de incrementar el número de datos transformándolos para aumentar la capacidad de generalización.

Una red neuronal convolucional puede clasificar robustamente objetos incluso si se coloca en orientaciones diferente, esta propiedad es conocida como invarianza. Más específicamente, una CNN puede ser invariante para la traducción, el punto de vista, el tamaño o la iluminación (o una combinación de todos).

Esta es esencialmente la premisa del aumento de datos. En el escenario del mundo real, se puede tener un conjunto de datos de imágenes tomadas en un conjunto limitado de condiciones. Sin embargo, para la aplicación de destino pueden existir una variedad de condiciones, como diferente orientación, ubicación, escala, brillo, etc. Por ello se realiza

una modificación y un aumento de los datos tratando de aportar a la red estas situaciones sintéticamente adicionales.

El esquema utilizado, aportado por el profesor, se muestra a continuación:

```
## ------------------------------------------------------------
## Data augmentation
## ------------------------------------------------------------

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"
)

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 = 32,                        # grupos de 32 imágenes
  class_mode = "categorical"              # etiquetas categorical
)
```

Figura 11: Data augmentation

4. DISCUSIÓN DE RESULTADOS

Una vez explicada la red, en este apartado se procederá a explicar las distintas pruebas que se han realizado y los resultados obtenidos.

Debido a que se observó que los optimizadores jugaban un papel fundamental en los resultados obtenidos, se decidió llevar a cabo una investigación para probar cuales de ellos se utilizarían. Una vez decididos, tras diversos ensayos se estableció que las pruebas para fijar los mejores resultados cambiando el algoritmo de optimización, se realizarían con 30 épocas.

Además, tras haber probado distintas funciones de pérdida, primero erróneamente "binary_crossentropy" se realizaron todas las pruebas con sus respectivos resultados, hasta que contrastando los resultados se detectó el error. La siguiente función de pérdida utilizada fue "categorical_crossentropy" al tratarse de un problema de multiclasificación, sin embargo debido a los resultados bajos obtenidos, se estableció que la que se utilizaría sería la función "mean_squared_logarithmic_error", una variación del error cuadrático medio.

A continuación, se mostrarán los resultados utilizando la misma estructura para cada optimizador según el orden en que se realizaron las pruebas. Tras ajustar correctamente los hiperparámetros el único elemento a cambiar para realizar las pruebas fue el número de épocas y el algoritmo de optimización. En todos los casos se muestran los resultados obtenidos en entrenamiento y luego en test.

4.1. Optimizador RMSProp

- **20 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.05111339

$acc
[1] 0.68625
```

Figura 12: Entrenamiento RMSProp 20 épocas

  - **Resultados test:**

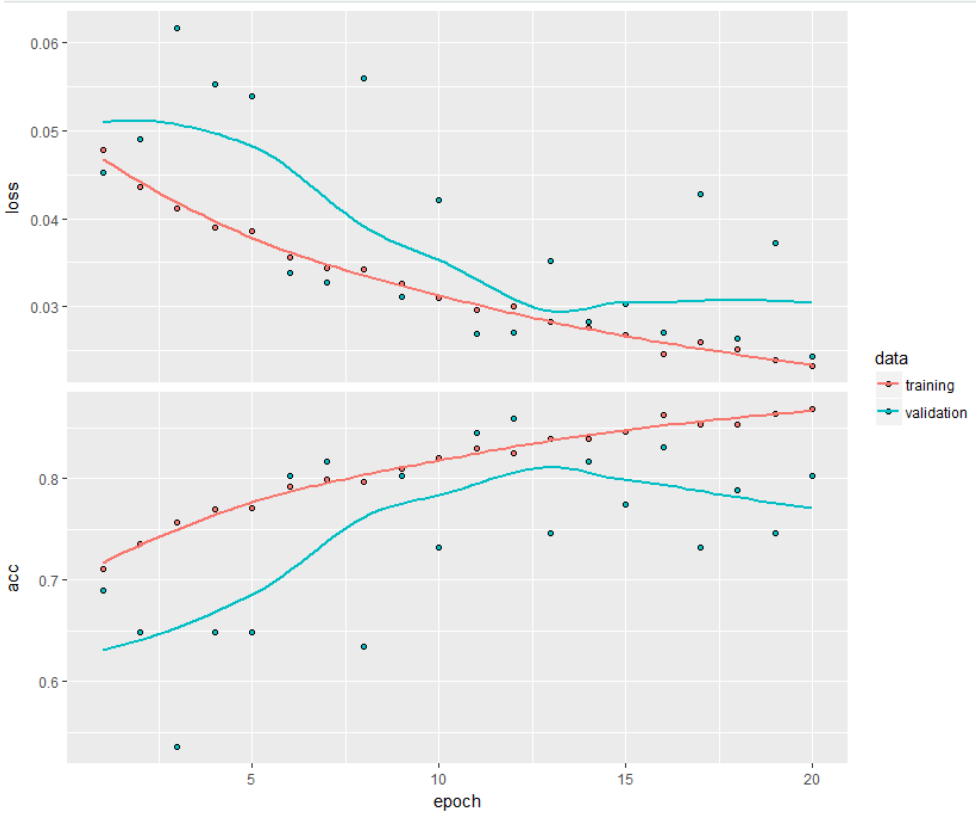

Figura 13: Plot Test RMSProp 20 épocas

```
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.0316983

$acc
[1] 0.8183333
```

Figura 14: Resultados test RMSProp 20 épocas

- **30 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.04458109

$acc
[1] 0.74875
```

Figura 15: Entreamiento RMSProp 30 épocas

  - **Resultados test:**

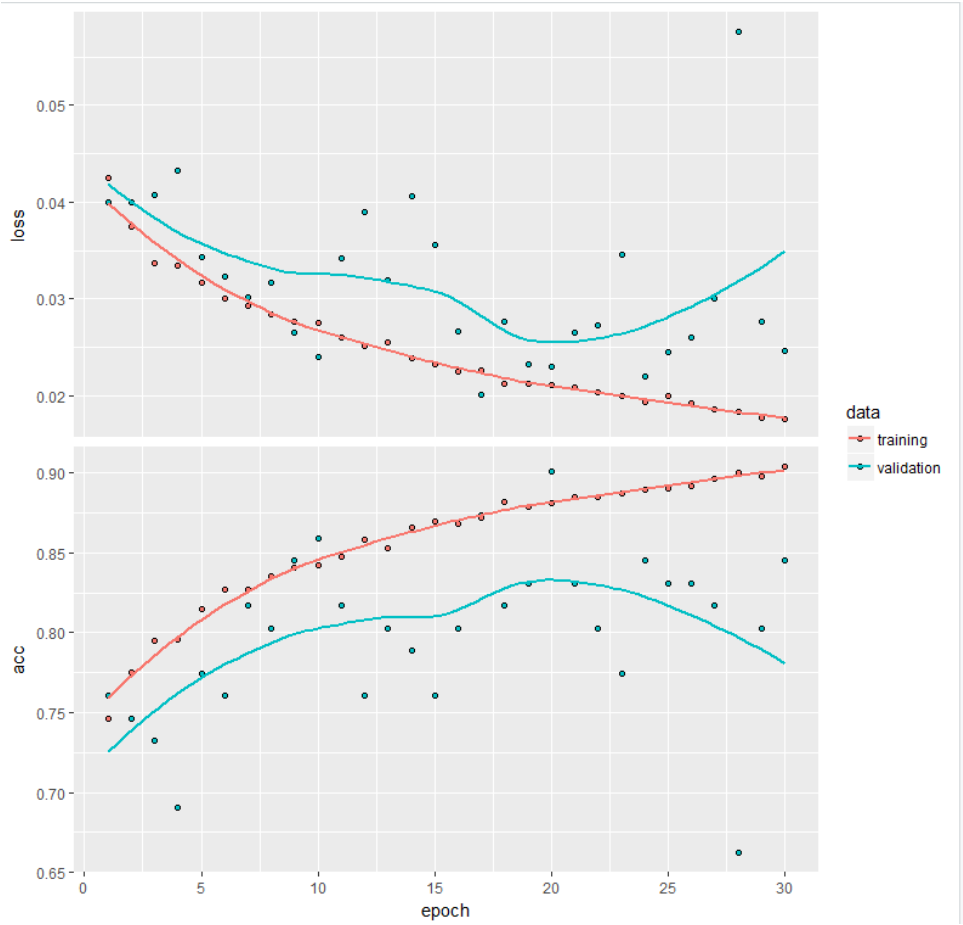

Figura 16: Plot Test RMSProp 30 èpocas

```
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.02791612

$acc
[1] 0.83125
```

Figura 17: Resultados Test RMSProp 30 épocas

4.2. Optimizador Adadelta

- **20 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.03840559

$acc
[1] 0.78
```

Figura 18: Entrenamiento Adadelta 20 épocas

  - **Resultados test:**

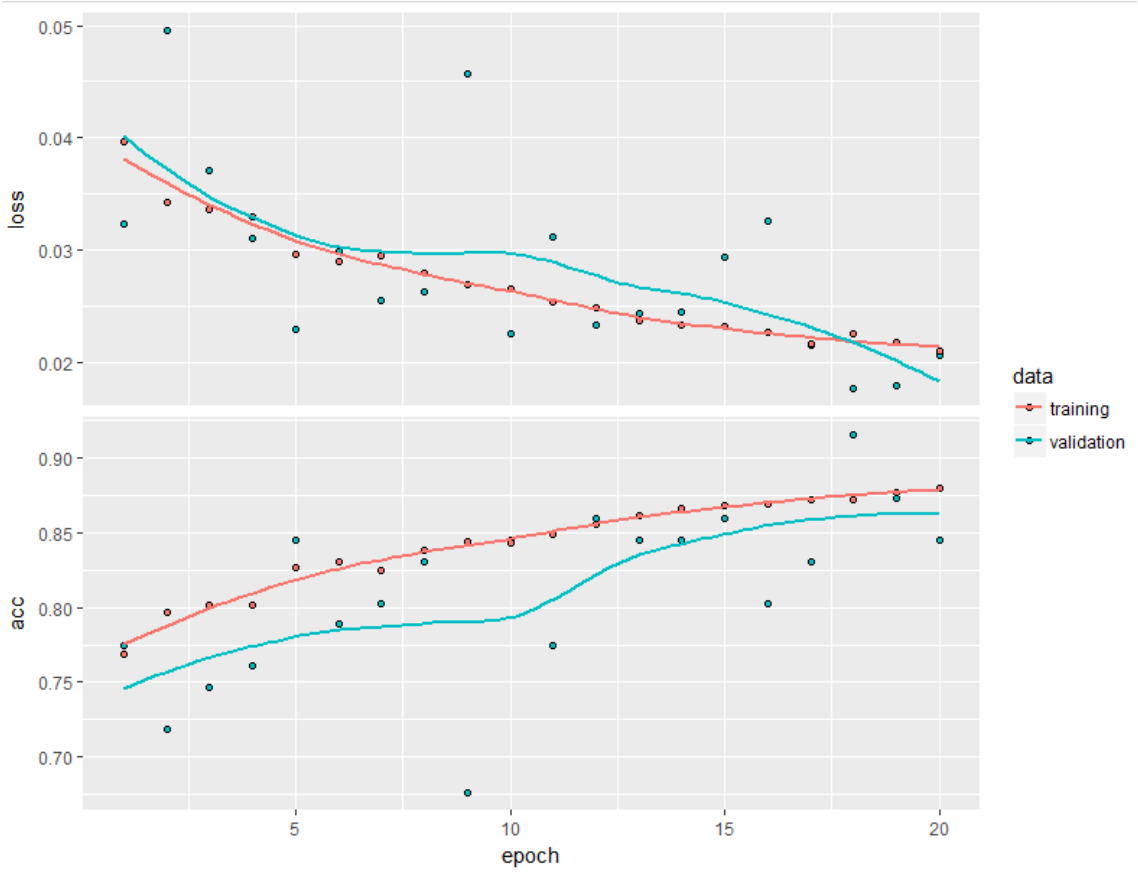

Figura 19: Plot Test Adadelta 20 èpocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.01808415

$acc
[1] 0.8775
```

Figura 20: Resultados test RMSProp 20 épocas

- **30 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.03872136

$acc
[1] 0.8029167
```

Figura 21: Entrenamiento Adadelta 30 épocas

  - **Resultados test:**

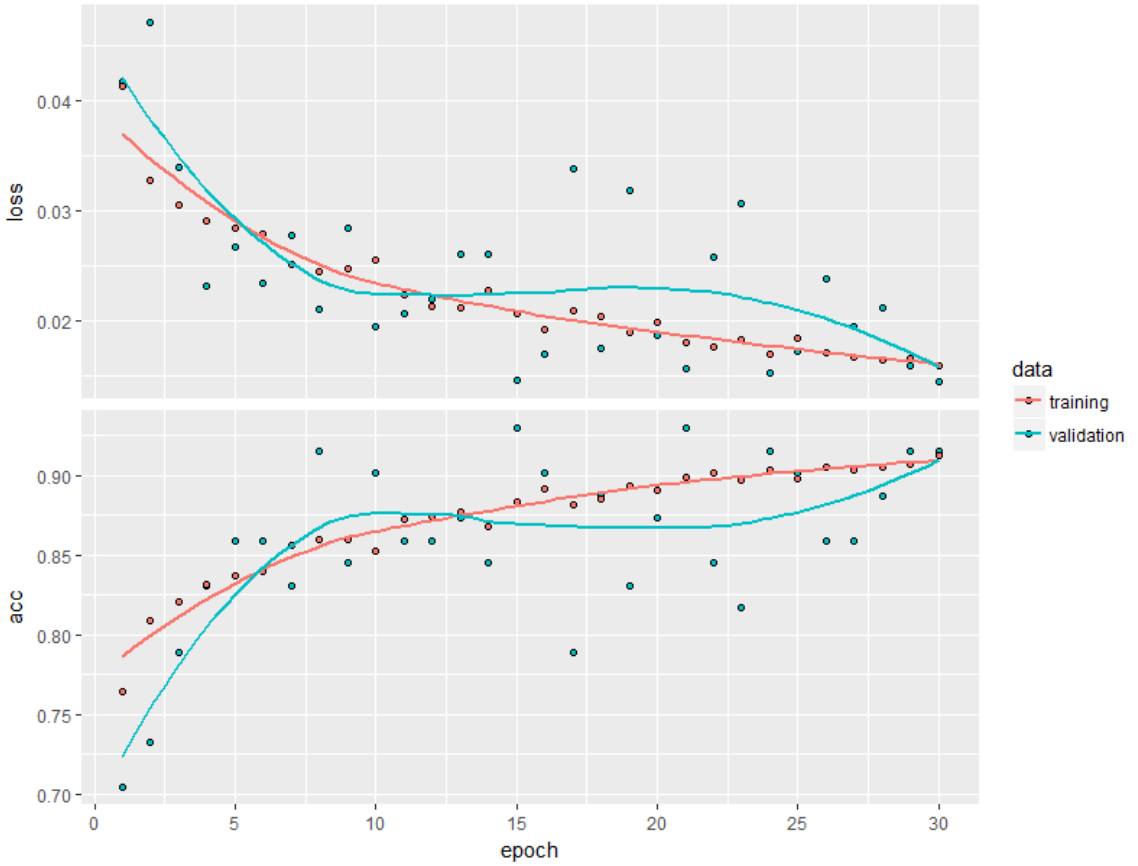

Figura 22: Plot Test Adadelta 30 épocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.01432471

$acc
[1] 0.9154167
```

Figura 23: Resultados Test Adadelta 30 épocas

4.3. Optimizador Adam

- **20 épocas:**

  - **Resultados entrenamiento:**

```
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.06946567

$acc
[1] 0.6475
```

Figura 24: Entrenamiento Adam 20 épocas

  - **Resultados test:**

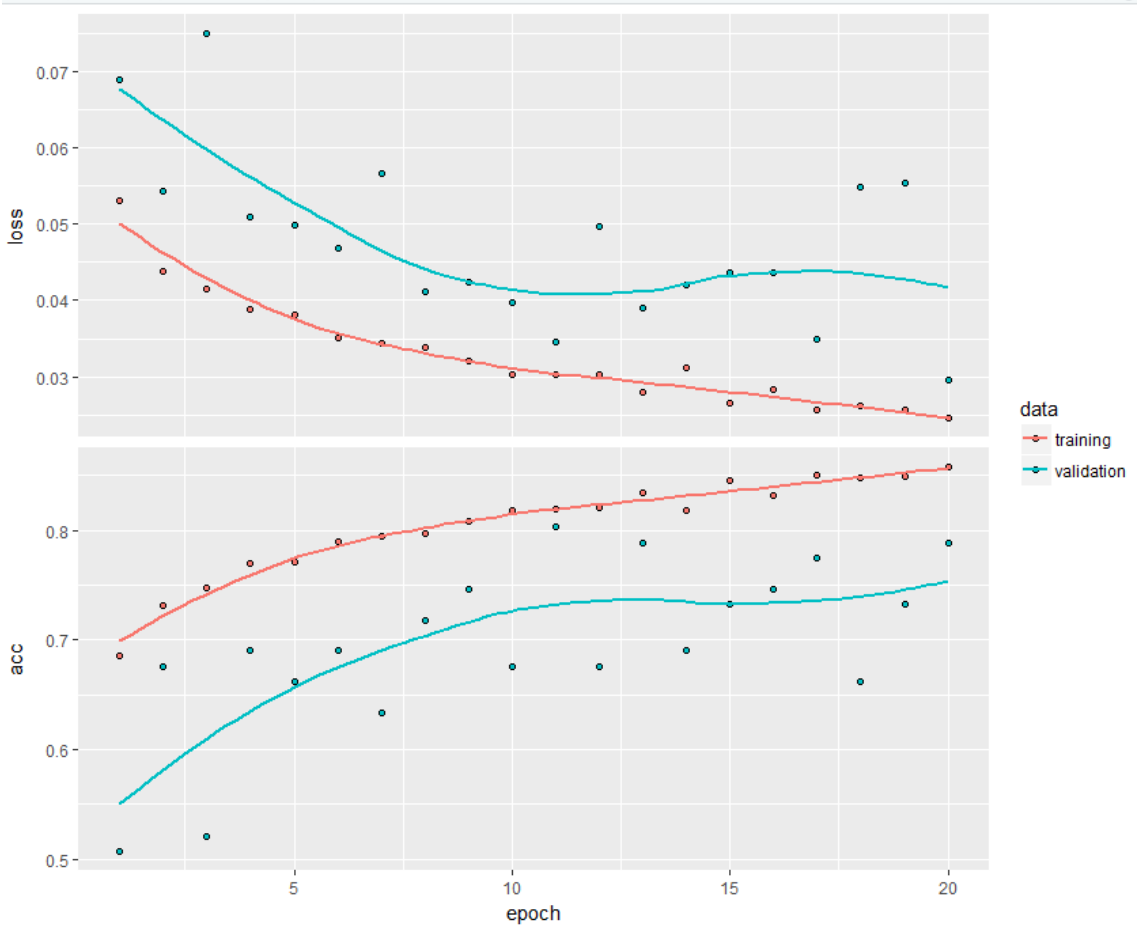

Figura 25: Plot Test Adam 20 épocas

```
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.02595512

$acc
[1] 0.8433333
```

Figura 26: Resultados Test Adam 20 épocas

- **30 épocas:**

  - **Resultados entrenamiento:**

```
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.06583292

$acc
[1] 0.6529167
```

Figura 27: Entrenamiento Adam 30 épocas

  - **Resultados test:**

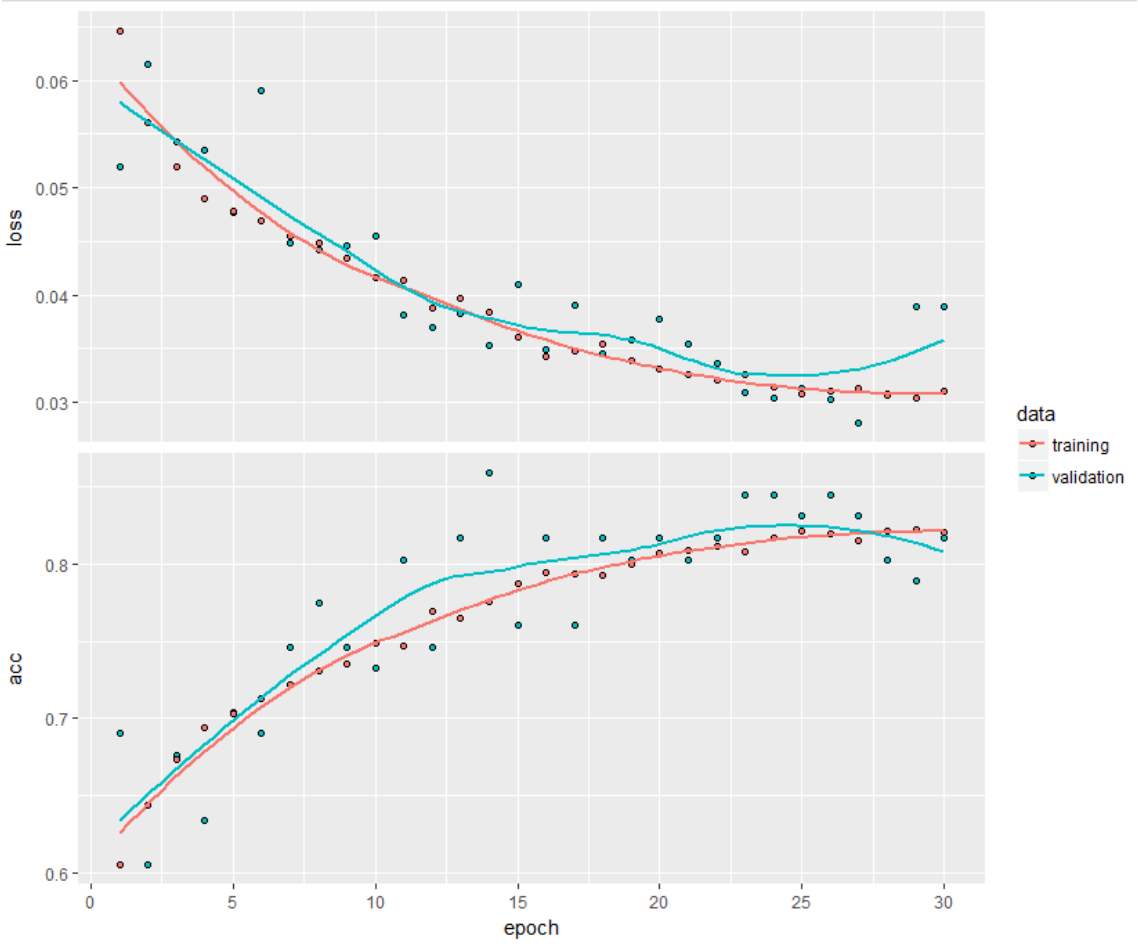

Figura 28: Plot Test Adam 30 épocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.03328499

$acc
[1] 0.81625
```

Figura 29: Resultados Test Adam 30 épocas

4.4. Optimizador SGD

- **20 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.09215223

$acc
[1] 0.29
```

Figura 30. Entrenamiento SGD 20 épocas

  - **Resultados test:**

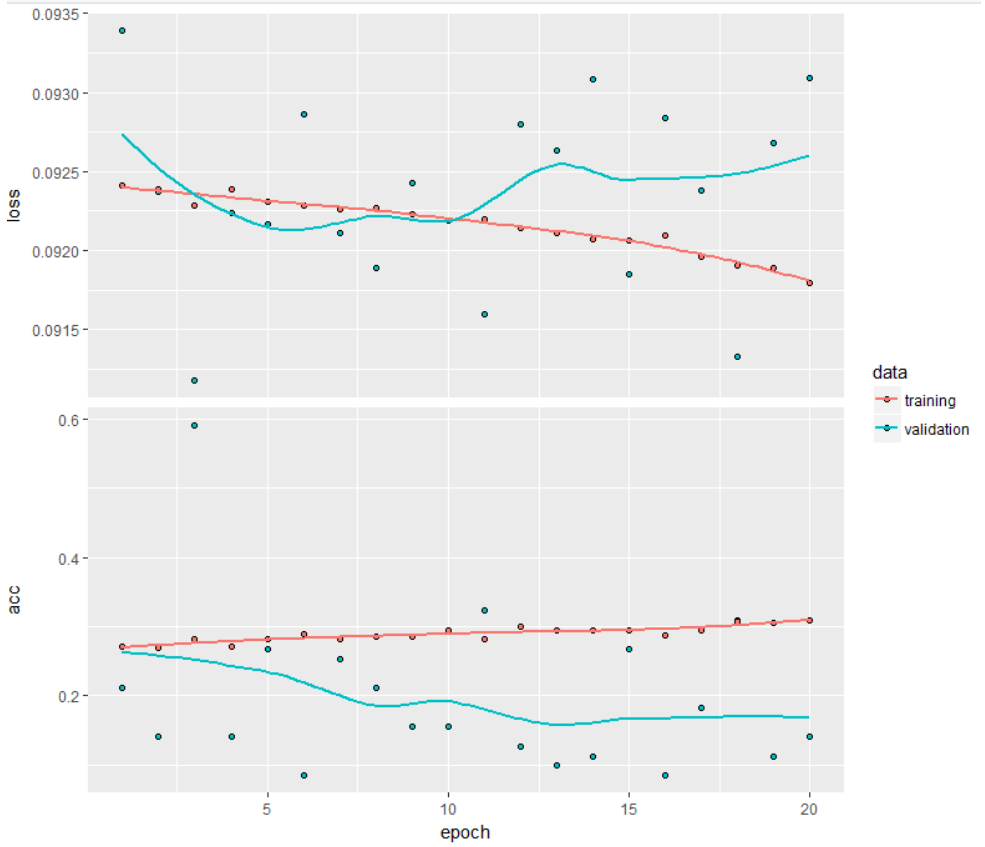

Figura 31: Plot Test SGD 20 épocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.09102154

$acc
[1] 0.3391667
```

Figura 32: Resultados Test SGD 20 épocas

- **30 épocas:**

  - **Resultados entrenamiento:**

```
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.09163028

$acc
[1] 0.3025
```

Figura 33: Entrenamiento SGD 30 épocas

  - **Resultados test:**

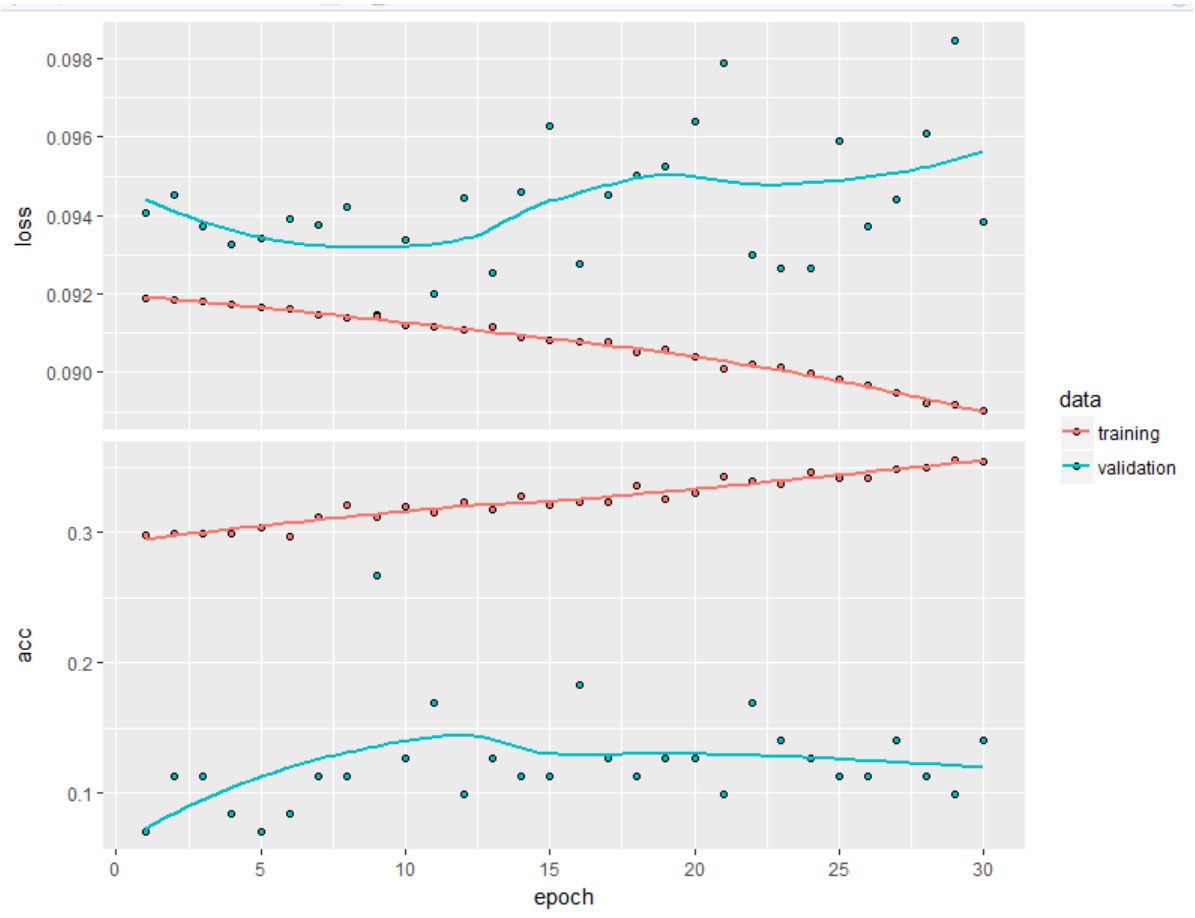

Figura 34: Plot Test SGD 30 épocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.08651127

$acc
[1] 0.3970833
```

Figura 35: Resultados Test SGD 30 épocas

4.5. Optimizador Adagrad

- **20 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.18037

$acc
[1] 0.2491667
```

Figura 36: Entrenamiento Adagrad 20 épocas

  - **Resultados test:**

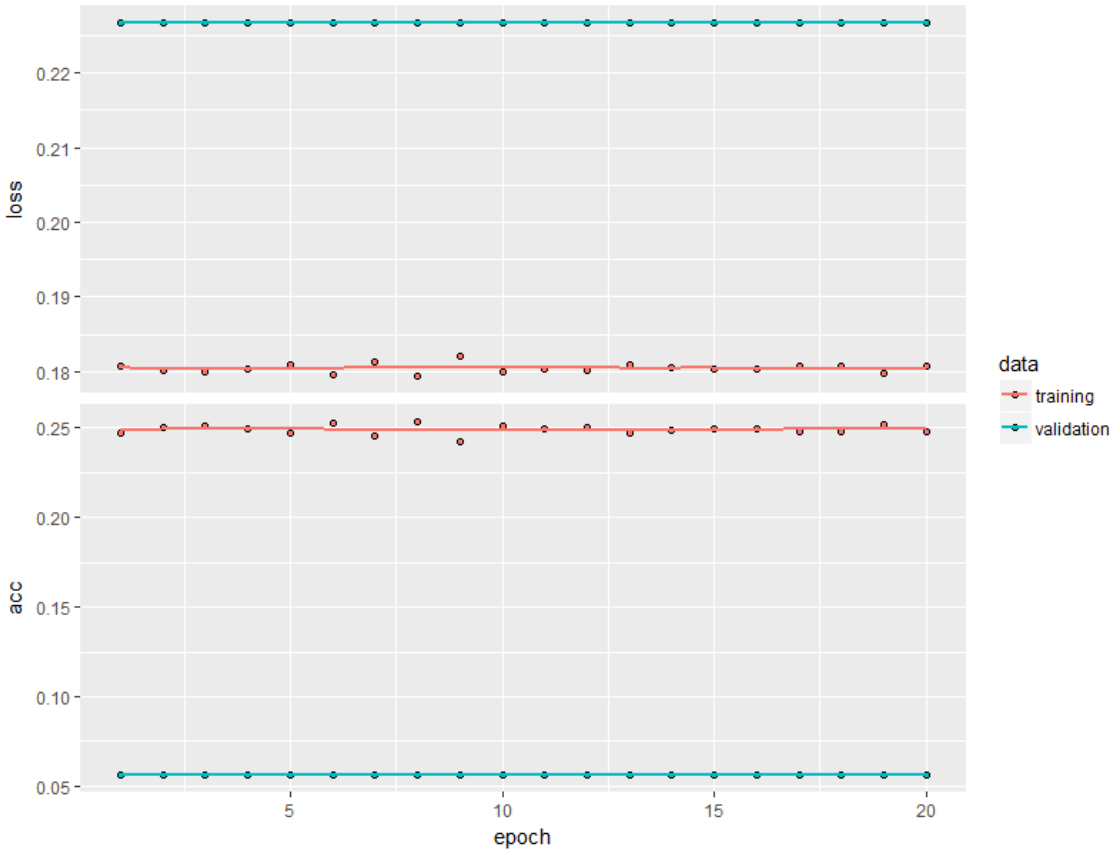

Figura 37: Plot Test Adagrad 20 épocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.18037

$acc
[1] 0.2491667
```

Figura 38: Resultados Test Adagrad 20 épocas

- **30 épocas:**

  - **Resultados entrenamiento:**

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.08985093

$acc
[1] 0.3891667
```

Figura 39: Entrenamiento Adagrad 30 épocas

  - **Resultados test:**

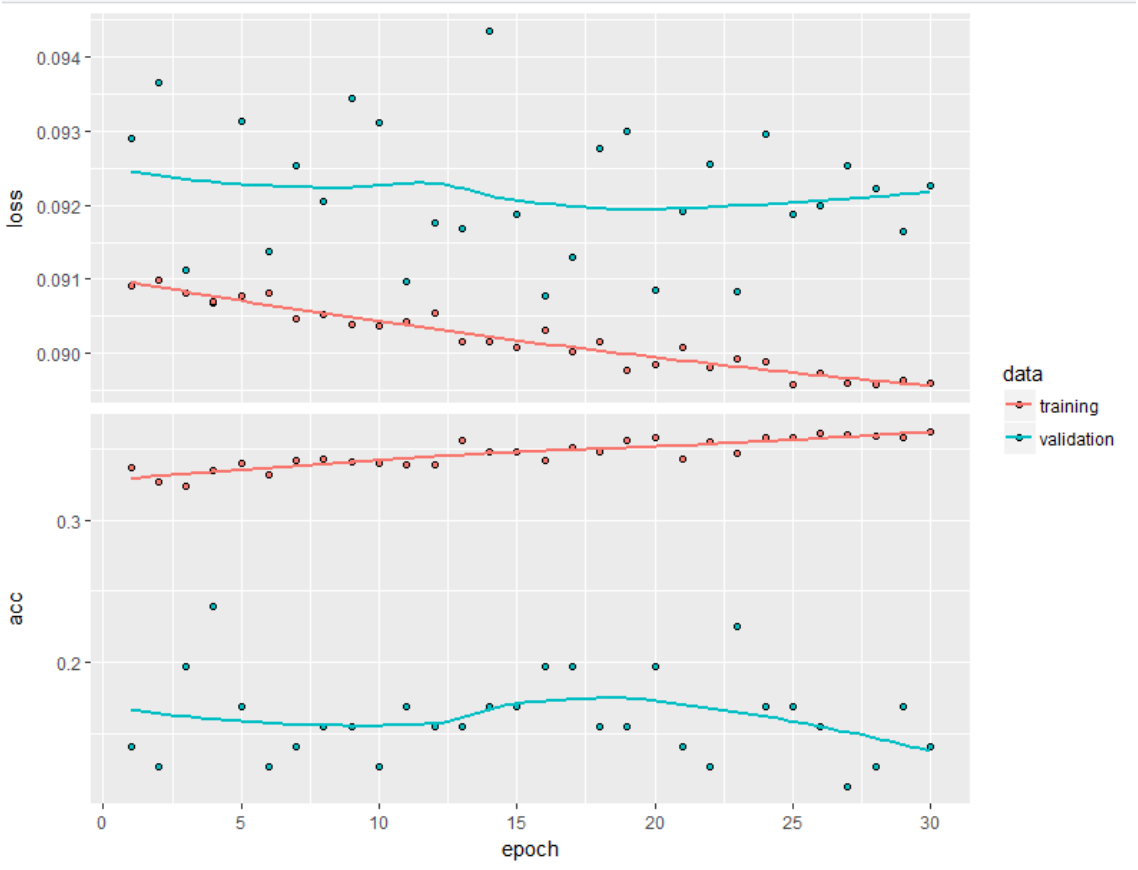

Figura 40: Plot Test Adagrad 30 épocas

```
> # Evaluar modelo
> model %>% evaluate_generator(test_data, steps = 75)
$`loss`
[1] 0.08737797

$acc
[1] 0.41125
```

Figura 41: Resultados Test Adagrad 30 épocas

4.6. Resultados obtenidos

Una vez mostrados todos los resultados se comentarán los detalles obtenidos de los mismos. A continuación, se evaluarán los resultados obtenidos comparando los algoritmos de optimización para 20 y 30 épocas en entrenamiento y test:

- **Resultados de entrenamiento con 20 épocas:**

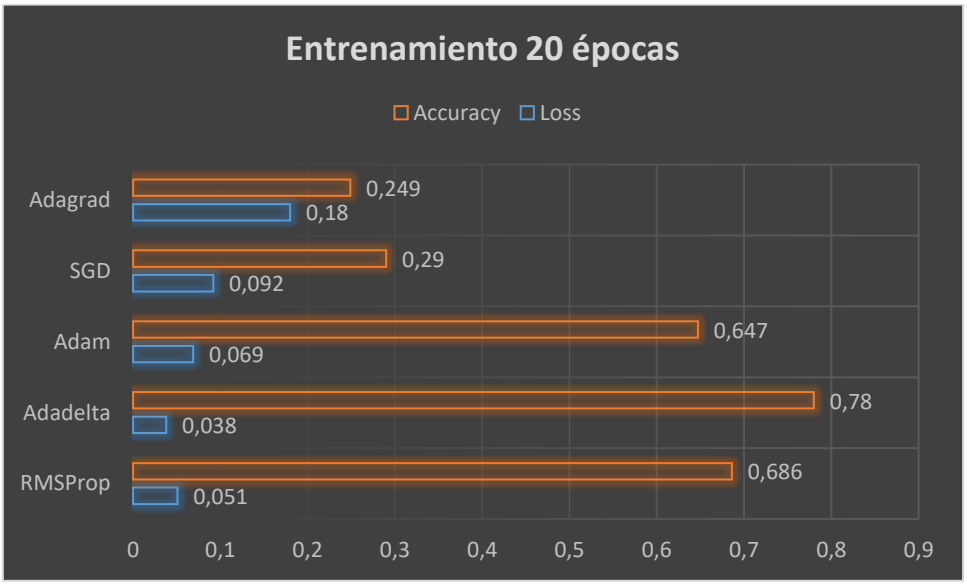

Figura 42: Resultados entrenamiento 20 épocas

- **Resultados de entrenamiento con 30 épocas:**

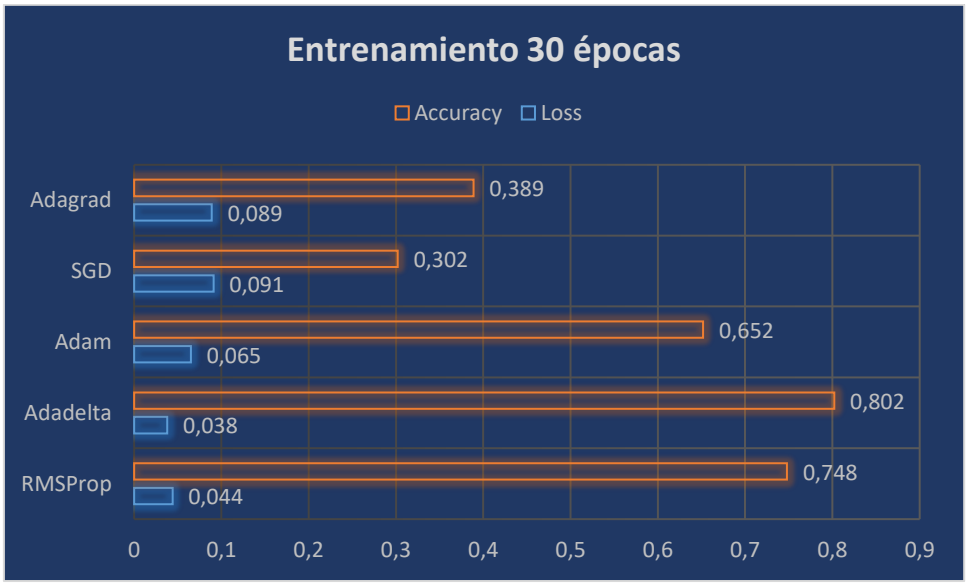

Figura 43: Resultados entrenamiento 30 épocas

En primer lugar, concretando en el entrenamiento, se pueden observar unos mejores resultados con la ejecución en 30 épocas. Consiguiendo en general, unos valores mayores de accuracy y una tasa de pérdida inferior.

Tal y como refleja el gráfico, durante el entrenamiento el algoritmo de optimización que ha obtenido mejores resultados ha sido el Adadelta. Además como se puede ver en el siguiente gráfico realiza un ascenso progresivo de los resultados en accuracy y decreciente en el error, proporcionando buenos resultados:

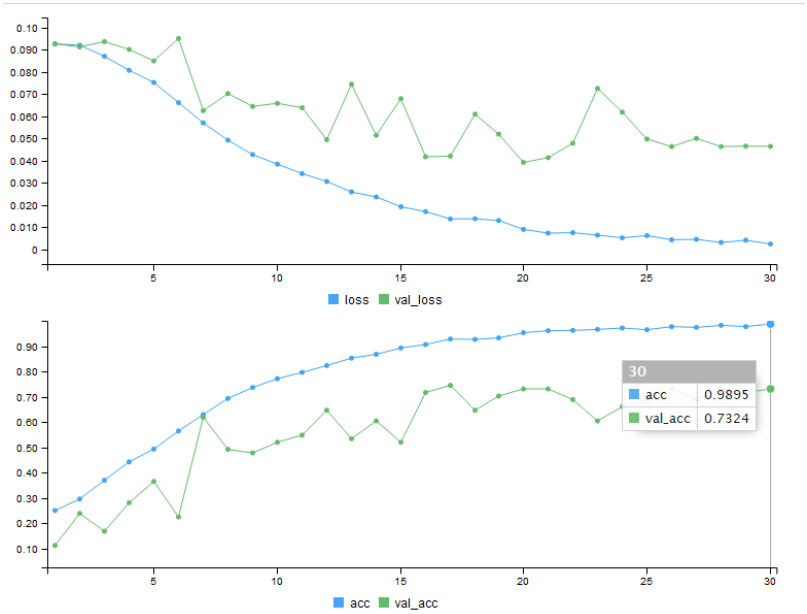

Figura 44: Plot de entrenamiento Adadelta con 30 épocas

Siendo estos los mejores resultados del entrenamiento cabe esperar un rendimiento similar en los test. Por ello a continuación se muestran los gráficos obtenidos sobre las pruebas de test:

- **<u>Resultados de test 20 épocas:</u>**

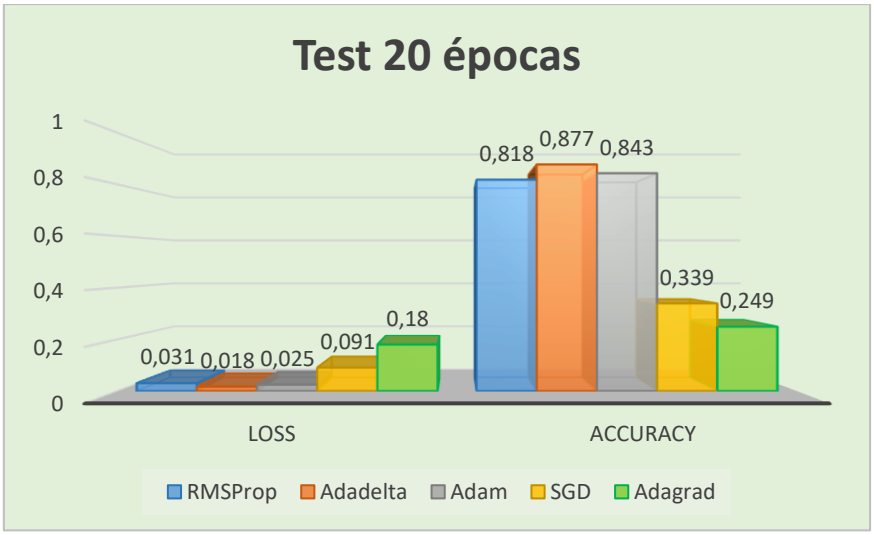

Figura 45: Resultados test 20 épocas

- **Resultados de test 30 épocas:**

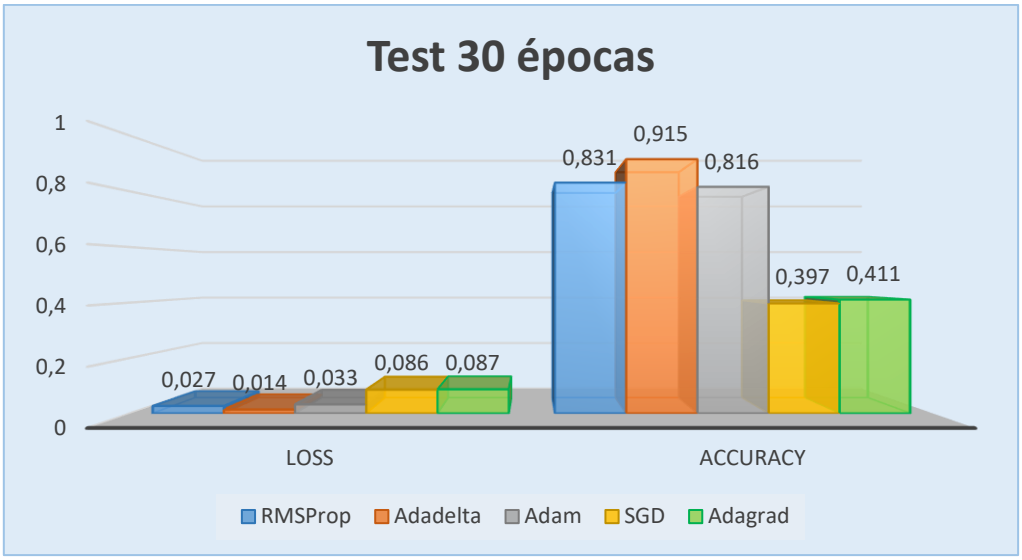

Figura 46: Resultados test 30 épocas

Como se puede observar en general se han obtenido mejores resultados ejecutando 30 épocas. Además en cuanto a los resultados se puede extraer como conclusión que el algoritmo Adadelta ha obtenido los mejores resultados llegando a 0.915 en accuracy y ofreciendo una tasa de pérdida muy baja de tan sólo 0.014, algo que ya se presuponía contrastando los datos del entrenamiento.

Por otro lado, se puede observar también el buen rendimiento de otros dos optimizadores como Adam y RMSProp los cuales han obtenido resultados inferiores pero muy buenos también.

Sin embargo, no se puede afirmar lo mismo de los optimizadores SGD y Adagrad, los cuales han obtenido unos resultados de "accuracy" bajos con respecto a los demás. Así mismo, también muestran unas tasas de pérdida altas en comparación con el resto de optimizadores pero que en general son muy bajas con respecto al uso de otras funciones de pérdida como "categorical_crossentropy". También se puede destacar la mejora del algoritmo Adagrad, que ha superado al SGD al contrario que en el entrenamiento.

5. CONCLUSIONES

Una vez discutidos los resultados se puede concluir que se ha realizado una práctica completa y muy interesante.

El uso de distintos algoritmos de optimización ha dado lugar a una visión global del ámbito de estos optimizadores y su gran importancia en el mundo de las redes neuronales.

Cabe destacar que se han realizado muchas más pruebas de las que aquí se muestran, así como la ejecución también con 40 épocas y la captura de gráficos tanto de entrenamiento como de test, sin embargo, debido a los peores resultados y a la

extensión con las imágenes se ha decidido mostrar una visión reducida de las pruebas que facilite su rápida comprensión y aportando los datos más concisos.

Concluye de esta manera una asignatura que ha conseguido profundizar en algunos conceptos iniciados en otras materias y permite tener una base sólida sobre la que se podrán abordar temas relacionados con el tratamiento de datos en una empresa si un puesto futuro lo requiriese.

Así mismo, cabe destacar la ayuda aportada por el material de la asignatura y las clases que han conseguido que no fuese un temario que abordar solamente desde los fundamentos teóricos, sino que ha conseguido que se aplicase de una manera práctica y entretenida por parte de los alumnos.

6. BIBLIOGRAFÍA

[1]- https://www.ibm.com/support/knowledgecenter/es/SSLVMB_24.0.0/spss/neural_network/nnet_whatis.html **- Último acceso: 01/06/2018**

[2]- https://www.kaggle.com/paultimothymooney/blood-cells **- Último acceso: 01/06/2018**

[3]- https://relopezbriega.github.io/blog/2017/06/13/introduccion-al-deep-learning/ **- Último acceso: 01/06/2018**

[4]- http://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent **- Último acceso: 01/06/2018**

[5]- https://keras.io/optimizers/ **- Último acceso: 01/06/2018**

[6]- https://www.javiercancela.com/2017/04/09/python-machine-learning-iv/ **- Último acceso: 01/06/2018**

[7]-

https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ **- Último acceso: 01/06/2018**

[8]- https://github.com/jgromero/sige2018/blob/master/pr%C3%A1cticas/4.%20Modelos%20avanzados%20de%20anal%C3%ADtica/dogsVScats/dogs_cats.R **- Último acceso: 01/06/2018**

[9]- http://elvex.ugr.es/decsai/deep-learning/slides/NN5%20Regularization.pdf - **Fernando Berzal [2018]. Último acceso: 01/06/2018**

[10]- Apuntes de la asignatura Sistemas Inteligentes para la Gestión en la Empresa, impartida en el Máster de Ingeniería Informática en la **Universidad de Granada.**