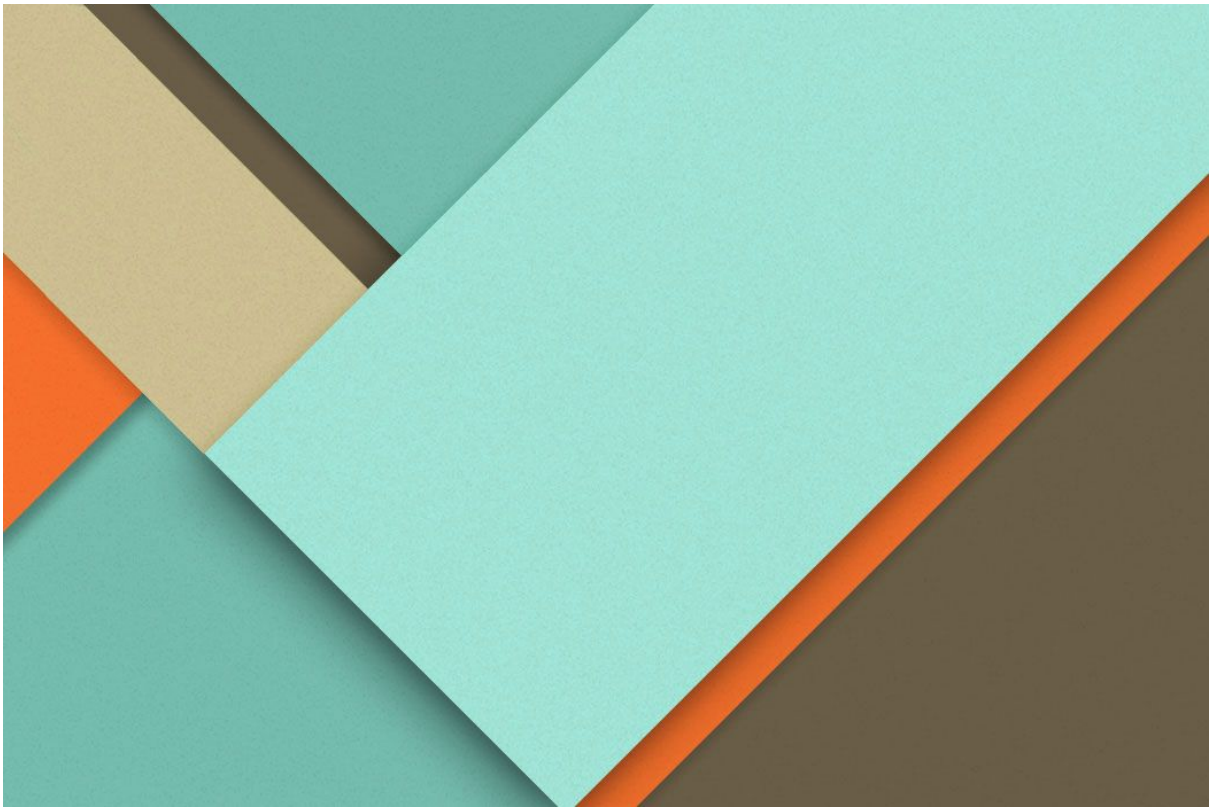


Deep Learning para multi-clasificación

Práctica 2

Maria Victoria Santiago Alcalá - Dayana Aguirre Iñiguez
Sistemas Inteligentes para la Gestión en la Empresa
Universidad de Granada
Granada, Junio 2018

Introducción

En esta práctica se ha realizado el estudio de cómo crear un modelo de clasificación de imágenes basado en redes neuronales profundas, sobre el conjunto de imágenes de células sanguíneas del dataset de Kaggle Blood Cell Images (el cual se puede encontrar en el enlace https://www.kaggle.com/paultimothymooney/blood-cells), en el que se ha llevado a cabo la tarea de predecir cual es el tipo de las celulas en una imagen, las mismas que pueden pertenecer a las cuatro categorias que se citan a continuación:

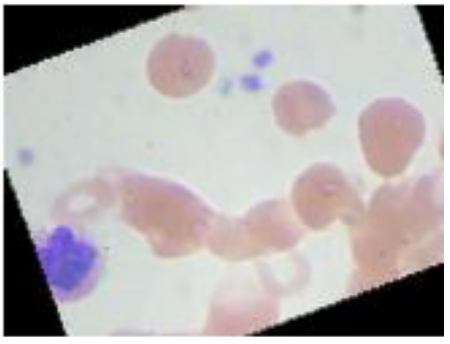
- Eosinophil
- Lymphocyte
- Monocyte
- Neutrophil

Con el fin de resolver de una forma más oportuna y a su vez utilizando los recursos proporcionados por nuestros equipos usaremos Keras que es una librería de Python que nos proporciona de manera limpia y sencilla la creación de un modelo de Deep Learning encima de otras librerías en este caso TensorFlow con soporte de GPU; para instalarlo requerimos: CUDA toolkit, NVIDIA drivers, última versión cuDNN entre otros.

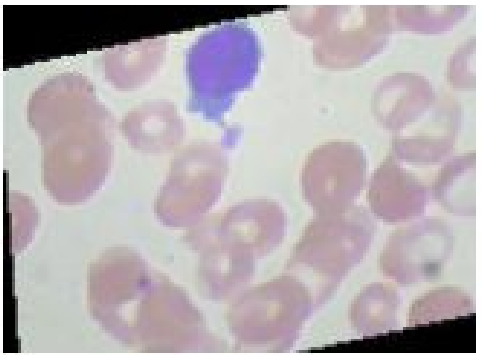
Dataset

El dataset a analizar contiene un total de 12500 imágenes de blood cells en formato JPEG, las cuales están divididas en 4 diferentes grupos.

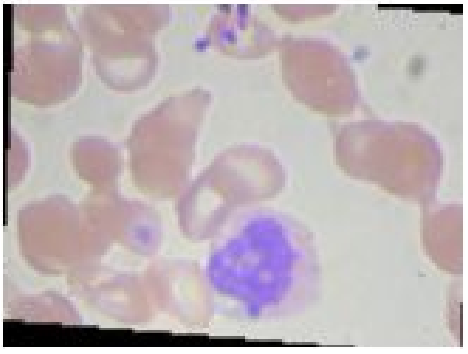
1. EOSINOPHIL



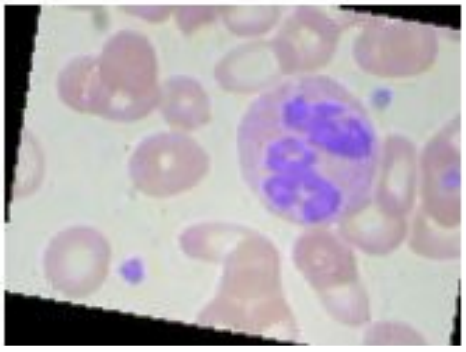
2. LYMPHOCYTE



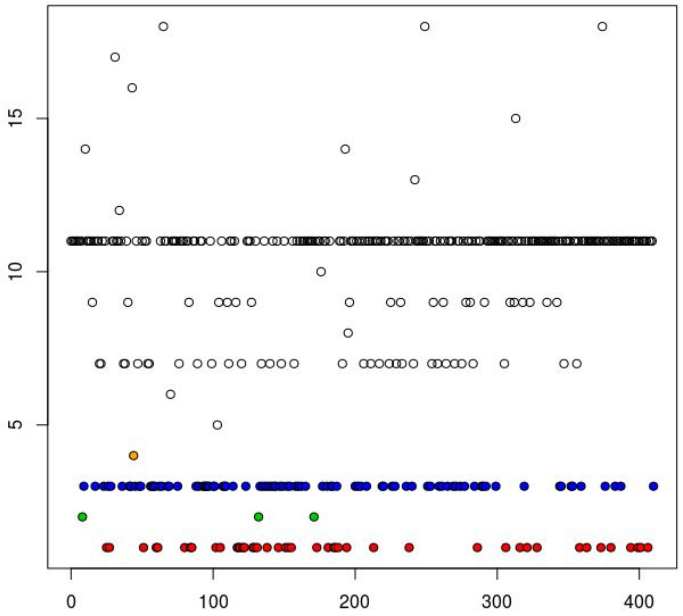
3. MONOCYTE



4. NEUTRIPHIL



Inicialmente, tras cargar nuestro dataset podemos ver la distribución de las células en las categorías anteriormente mencionadas como muestra en la imagen que se muestra a continuación.




```
> summary(cell_images)
    X                Image          Category
 Mode:logical   Min.   :  0.0   NEUTROPHIL:207
 NA's:411       1st Qu.:102.5   EOSINOPHIL: 88
                Median :205.0              : 44
                Mean   :205.0   LYMPHOCYTE: 33
                3rd Qu.:307.5   MONOCYTE  : 21
                Max.   :410.0   BASOPHIL  :  3
                                (Other)   : 15
```

Ilustración 1. Células distribuidas en categorías.

Seguidamente, tras el preprocesamiento de las imágenes, llevando a cabo tareas como bien son la rotación, el escalado, zoom y demás operaciones, se nos aumenta el tamaño de nuestro set de entrenamiento quedando en 9957 imágenes correspondientes a nuestras 4 clases equilibrando con ello el conjunto de datos a tratar.

A continuación damos paso al apartado fundamentos teóricos el cual detalla los procedimientos realizados.

Fundamentos teóricos

No existe una fórmula exacta para determinar cuál es el número correcto de capas, o cuál es el tamaño correcto para cada capa. Por eso evaluaremos evaluar una serie de arquitecturas diferentes con el conjunto de validación y no con el conjunto de pruebas para así encontrar el tamaño de modelo correcto para sus datos.

El flujo de trabajo general para encontrar un tamaño de modelo apropiado es comenzar con relativamente pocas capas y parámetros, y comenzar a aumentar el tamaño de las capas o agregar nuevas capas hasta que vea rendimientos decrecientes con respecto a la pérdida de validación.

Data Preprocessing

Como primer paso hay que formatear o preprocesar los datos antes de alimentarlos a nuestra red. Actualmente, nuestros datos se encuentran en disco como archivos JPEG, por lo que los pasos para acceder a nuestra red son aproximadamente:

- Leer los archivos de imagen.
- Decodificar el contenido JPEG en cuadrículas RGB de píxeles.
- Convertir floating point tensors
- Cambiar la escala de los valores de píxel (entre 0 y 255) al intervalo [0, 1]

Argumentos flow_images_from directory

1. Directory = train_dir -> ruta al directorio de destino, el que contiene un subdirectorio por clase.
2. generator = train_datagen -> Generador de datos de imagen con una rescale = 1/255
3. target_size = c(150, 150) -> Dimensiones de las imágenes
4. batch_size = 50
5. class_mode = "categorical"

Creación del modelo

Se creará una pila de etapas alternas entre de layer_conv_2d () (con relu activation) y layer_max_pooling_2d (); adicional tendrá más de una de esas etapas, así podremos aumentar la capacidad de la red y así reducir el tamaño de los mapas de características para que no sean demasiado grandes cuando llegue a la capa layer_flatten (). Comenzamos con entradas de tamaño 150 × 150 y terminaremos con mapas de funciones de tamaño 7 × 7. La profundidad de los mapas de características aumenta progresivamente en la red (de 32 a 128), mientras que el tamaño de los mapas de características disminuye (de 148 × 148 a 7 × 7). Debido a que está atacando un problema de clasificación categorical, terminará la red con un layer_dense (4,512) y una activación softmax. Con el fin de evitar para evitar el sobreajuste usamos Dropout de entorno al 0.4 - 0.2.

Esta unidad codificará la probabilidad de que la red esté mirando una clase u otra.

Layer (type)	Output Shape	Param #
conv2d_107 (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d_99 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_108 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_100 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_109 (Conv2D)	(None, 34, 34, 64)	36928
max_pooling2d_101 (MaxPooling2D)	(None, 17, 17, 64)	0
conv2d_110 (Conv2D)	(None, 15, 15, 128)	73856
max_pooling2d_102 (MaxPooling2D)	(None, 7, 7, 128)	0
conv2d_111 (Conv2D)	(None, 5, 5, 128)	147584
max_pooling2d_103 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten_15 (Flatten)	(None, 512)	0
dropout_19 (Dropout)	(None, 512)	0
dense_40 (Dense)	(None, 512)	262656
dense_41 (Dense)	(None, 128)	65664
dense_42 (Dense)	(None, 4)	516

Ilustración 2- Modelo creado como primera prueba.

Para crear el modelo simplemente tenemos que introducir en nuestro script de R capas como las que se muestran en las siguientes imágenes la cuales corresponden a los dos modelos finalmente tratados:

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation = "relu", input_shape = c(150, 150, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32,  kernel_size = c(3, 3), activation = "relu", input_shape = c(150, 150, 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_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 4, activation = "softmax")
```

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(150, 150, 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 = 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_dropout(rate=0.4) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 4, activation = "softmax")
```

Ilustración 3 - Códigos de los modelos empleados.

Compilación

Utilizaremos diferentes optimizadores como bien son RMSprop, SGD, Y ADAM se va a terminar nuestra red con una sola unidad softmax. Usaremos la crossentropía categorical como nuestra pérdida.

```
# Compilar modelo
# https://tensorflow.rstudio.com/keras/reference/compile.html
model %>% compile(
  loss= 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(), #optimizer_adam( lr= 0.0001 , decay = 1e-6 ),
  metrics = c("accuracy")
)
```

```
# Compilar modelo
# https://tensorflow.rstudio.com/keras/reference/compile.html
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_adam(lr=0.0001,decay = 1e-6),
  metrics = c('accuracy')

)
```

```
# Compilar modelo
# https://tensorflow.rstudio.com/keras/reference/compile.html
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_sgd(lr=0.0001,decay = 1e-6),
  metrics = c('accuracy')
```

Ilustración 4. Ejemplos de compilación del modelo

Entrenamiento

Utilizamos la función fit_generator, como primer argumento le indicaremos los lotes de entradas, debido a que los datos se están generando infinitamente, el generador necesita saber cuántas muestras extraer del generador antes de declarar una época(steps_per_epoch=100), luego de ejecutar el steps de gradiente de descenso, el proceso de adaptación irá a la próxima época. Cabe indicar un validation_data y validation_steps, que le indica al proceso cuántos lotes extraer del generador de validación para su evaluación.

```
history <- model %>%
  fit_generator(
    train_data,
    steps_per_epoch = 200,
    epochs = 30,
    validation_data = validation_data,
    validation_steps = 50
```

Resultados

Epoch 1/30

200/200 [==============] - 456s 2s/step - loss: 1.3777 - acc: 0.2846 - val_loss: 1.3091 - val_acc: 0.6197

Epoch 2/30

200/200 [==============]- 478s 2s/step - loss: 1.1745 - acc: 0.4187 - val_loss: 1.5000 - val_acc: 0.1127

Epoch 3/30

200/200 [==============]- 486s 2s/step - loss: 1.0498 - acc: 0.4997 - val_loss: 1.0333 - val_acc: 0.4789

Epoch 4/30

200/200 [==============]- 477s 2s/step - loss: 0.8540 - acc: 0.6364 - val_loss: 0.8487 - val_acc: 0.5915

Epoch 5/30

200/200 [==============]- 491s 2s/step - loss: 0.6922 - acc: 0.7039 - val_loss: 0.6995 - val_acc: 0.6761

Epoch 6/30

200/200 [==============]- 486s 2s/step - loss: 0.6020 - acc: 0.7533 - val_loss: 0.7777 - val_acc: 0.6620

Epoch 7/30

200/200 [==============]- 487s 2s/step - loss: 0.5205 - acc: 0.7755 - val_loss: 0.5619 - val_acc: 0.7324

Epoch 8/30

200/200 [==============]- 480s 2s/step - loss: 0.4807 - acc: 0.7984 - val_loss: 0.6307 - val_acc: 0.7042

Epoch 9/30

200/200 [==============]- 450s 2s/step - loss: 0.4459 - acc: 0.8176 - val_loss: 0.5951 - val_acc: 0.7042

Epoch 10/30

200/200 [==============]- 450s 2s/step - loss: 0.4019 - acc: 0.8338 - val_loss: 0.5740 - val_acc: 0.7746

Epoch 11/30

200/200 [==============]- 447s 2s/step - loss: 0.3826 - acc: 0.8472 - val_loss: 0.9038 - val_acc: 0.5634

Epoch 12/30

200/200 [==============]- 447s 2s/step - loss: 0.3477 - acc: 0.8573 - val_loss: 0.5325 - val_acc: 0.7606

Epoch 13/30

200/200 [==============]- 448s 2s/step - loss: 0.3340 - acc: 0.8671 - val_loss: 0.4564 - val_acc: 0.7887

Epoch 14/30

200/200 [==============]- 449s 2s/step - loss: 0.3283 - acc: 0.8807 - val_loss: 0.9385 - val_acc: 0.6620

Epoch 15/30

200/200 [==============]- 445s 2s/step - loss: 0.2998 - acc: 0.8834 - val_loss: 0.5370 - val_acc: 0.7887

Epoch 16/30

200/200 [==============]- 444s 2s/step - loss: 0.2808 - acc: 0.8948 - val_loss: 1.1332 - val_acc: 0.5915

Epoch 17/30

200/200 [==============]- 448s 2s/step - loss: 0.2689 - acc: 0.9014 - val_loss: 2.1224 - val_acc: 0.5070

Epoch 18/30

200/200 [==============]- 445s 2s/step - loss: 0.2588 - acc: 0.9081 - val_loss: 1.0998 - val_acc: 0.6197

Epoch 19/30

200/200 [==============]- 454s 2s/step - loss: 0.2363 - acc: 0.9139 - val_loss: 0.6982 - val_acc: 0.7746

Epoch 20/30

200/200 [==============]- 457s 2s/step - loss: 0.2368 - acc: 0.9138 - val_loss: 0.8438 - val_acc: 0.7465

Epoch 21/30

200/200 [==============]- 456s 2s/step - loss: 0.2545 - acc: 0.9134 - val_loss: 0.8177 - val_acc: 0.7042

Epoch 22/30

200/200 [==============]- 455s 2s/step - loss: 0.2174 - acc: 0.9248 - val_loss: 1.4065 - val_acc: 0.6479

Epoch 23/30

200/200 [==============]- 457s 2s/step - loss: 0.2050 - acc: 0.9277 - val_loss: 0.6665 - val_acc: 0.7183

Epoch 24/30

200/200 [==============]- 456s 2s/step - loss: 0.2432 - acc: 0.9216 - val_loss: 0.9594 - val_acc: 0.7465

Epoch 25/30

200/200 [==============]- 455s 2s/step - loss: 0.2377 - acc: 0.9231 - val_loss: 0.7668 - val_acc: 0.7324

Epoch 26/30

200/200 [==============]- 444s 2s/step - loss: 0.2333 - acc: 0.9294 - val_loss: 1.0908 - val_acc: 0.6901

Epoch 27/30

200/200 [==============]- 446s 2s/step - loss: 0.2101 - acc: 0.9289 - val_loss: 1.0009 - val_acc: 0.6338

Epoch 28/30

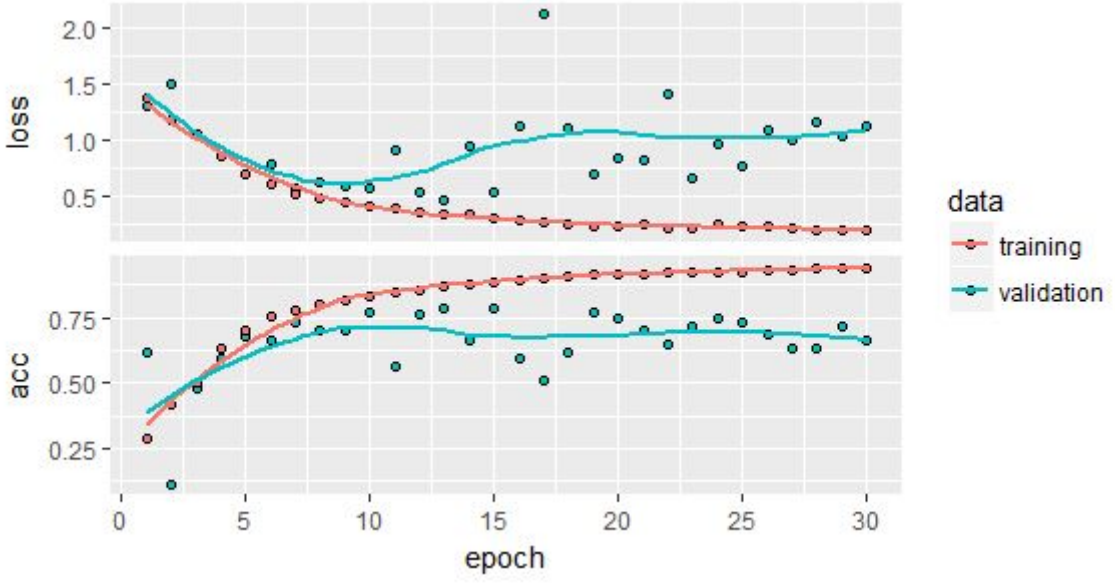
200/200 [==============]- 446s 2s/step - loss: 0.1858 - acc: 0.9388 - val_loss: 1.1692 - val_acc: 0.6338

Epoch 29/30

200/200 [==============]- 441s 2s/step - loss: 0.2009 - acc: 0.9370 - val_loss: 1.0391 - val_acc: 0.7183

Epoch 30/30

200/200 [==============]- 441s 2s/step - loss: 0.1870 - acc: 0.9426 - val_loss: 1.1207 - val_acc: 0.6620



Estos plots son características del sobreajuste. Nuestra precisión de entrenamiento aumenta linealmente con el tiempo, hasta que alcanza casi el 95%, mientras que nuestra precisión de validación se detiene en un 60-70%. Nuestra pérdida de validación varía constantemente con el transcurso de cada época, mientras que la pérdida de entrenamiento sigue disminuyendo linealmente hasta alcanzar casi 0.

Descripción de las redes empleadas

Data Augmentation

Resumidamente, la aumentación de datos consiste en agregar valor a los datos básicos que se tienen con la finalidad de conseguir un modelo mejor ya que los datos de un dataset no son infinitos por lo que el número de casos de uso a usar es finito.
Este aumento se hace agregando información derivada de fuentes internas y externas dentro de lo que sería el entorno del dataset. El aumento de datos puede ser muy beneficioso ya que agrega calidad a los datos mejorandolos significativamente.

Numerosos son los métodos que podemos encontrar para aumentar los datos, por ejemplo se pueden rotar las imágenes, recortarla o regular su luminosidad.

Su funcionamiento se puede resumir en los siguientes pasos:
En cada época de aprendizaje, las trasformaciones con parámetros seleccionados en un intervalo especificado al azar son aplicados a cada una de las imágenes originales del conjunto de entrenamiento.

El siguiente paso consiste en realizar una época, es decir, después de exponer nuestro algoritmo de aprendizaje a todo el conjunto de datos de entrenamiento, procedemos a iniciar la siguiente época de aprendizaje. Tenemos entonces que los datos de entrenamiento se

aumentan de nuevo mediante la aplicación de las correspondientes transformaciones que se especificaron en los datos de entrenamiento originales.

Por lo tanto, como estamos trabajando con redes neuronales y modelos de aprendizaje, se requiere que los datos estén preparados para trabajar con ellos lo cual viene siendo cada vez más complejo.
Para ello, aprovechando Keras, realizamos esta aumentación de forma simple y bastante potente. Keras nos proporciona la clase "image_data_generator" la cual define la configuración para la preparación y el aumento de nuestros datos de imágenes.
Se caracteriza por incluir capacidades como vienen a ser los siguientes apartados entre otros:

- Normalización de las características.
- Normalización de las muestras.
- Reordenamiento de las dimensiones.
- Rotación, volteos y demás manipulaciones.
- ZCA Blanqueamiento.
- Autoguardado de las imágenes que se han aumentado.

Una ventaja que tiene usar Keras es que en lugar de realizar el conjunto de operaciones en el dataset de imágenes completo en la memoria, lo que hace su API es iterar por el proceso de ajuste del modelo de aprendizaje creando con ello los datos de las nuevas aumentaciones de las imágenes.

Esta característica hace que se reduzca la sobrecarga de memoria pero agrega un ligero costo de tiempo adicional durante el entrenamiento del modelo.

En nuestro caso, hemos aplicado data augmentation creando y configurando el image_data_generator de forma ajustada a nuestros datos. En la siguiente imagen se puede ver un trozo del script que describe este proceso:

Ilustración X. Script Data Augmentation

Con ello lo que buscamos es calcular las estadísticas que sean necesarias para realizar las transformaciones a nuestros datos. Para ello hemos usado la función "fit_generador", en el generador de datos al cual le hemos pasado el conjunto de datos que teníamos como se muestra en el siguiente volcado de pantalla.

```
history_augmentation <- model %>%
  fit_generator(
    train_augmented_data,
    steps_per_epoch = 180,
    epochs = 70,
    validation_data = validation_data,
    validation_steps = 50
  )
```

Ilustración 6. Fit generator

Podemos ver también que se pueden devolver las muestras de imágenes por lotes. Para ello, se puede configurar el tamaño de cada lote y preparar nuestro generador de datos para obtener estos lotes de imágenes por medio de la función "flow_images_from_directory".

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

Ilustración 7. Flow images from directory

Con fit_generator podremos pasar la duración de época que queremos, el número de épocas a entrenar o por ejemplo el generador de datos entre otros.

A continuación podemos ver un ejemplo de código en R de como aplicar data augmentation con los posibles parámetros que podría tener. Se ha de comentar que estos parámetros se han de ir modificando para obtener el mejor valor posible en los campos accuracy y loss.

```
data_augmentation_datagen <- image_data_generator(
  rescale = 1/255,
  rotation_range = 10,
  width_shift_range = 0.3,
  height_shift_range = 0.3,
  shear_range = 0.3,
  zoom_range = 0.2,
  horizontal_flip = TRUE,
  fill_mode = "nearest"
)
```

```
data_augmentation_datagen <- image_data_generator(
  rescale = 1/255
)
```

Ilustración 8. Ejemplos de posibles parámetros en data augmentation

Y a continuación se muestra otro ejemplo de como se ha ejecutado augmentation data en RStudio:

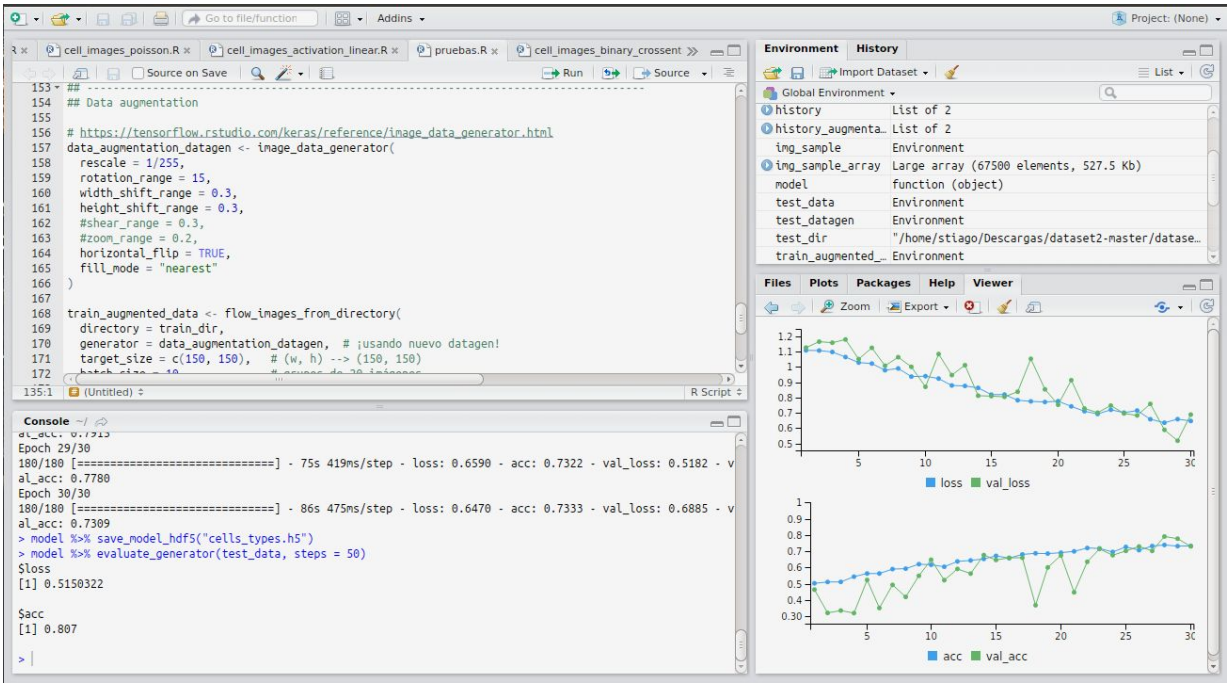


Ilustración 9. Aumentación de datos

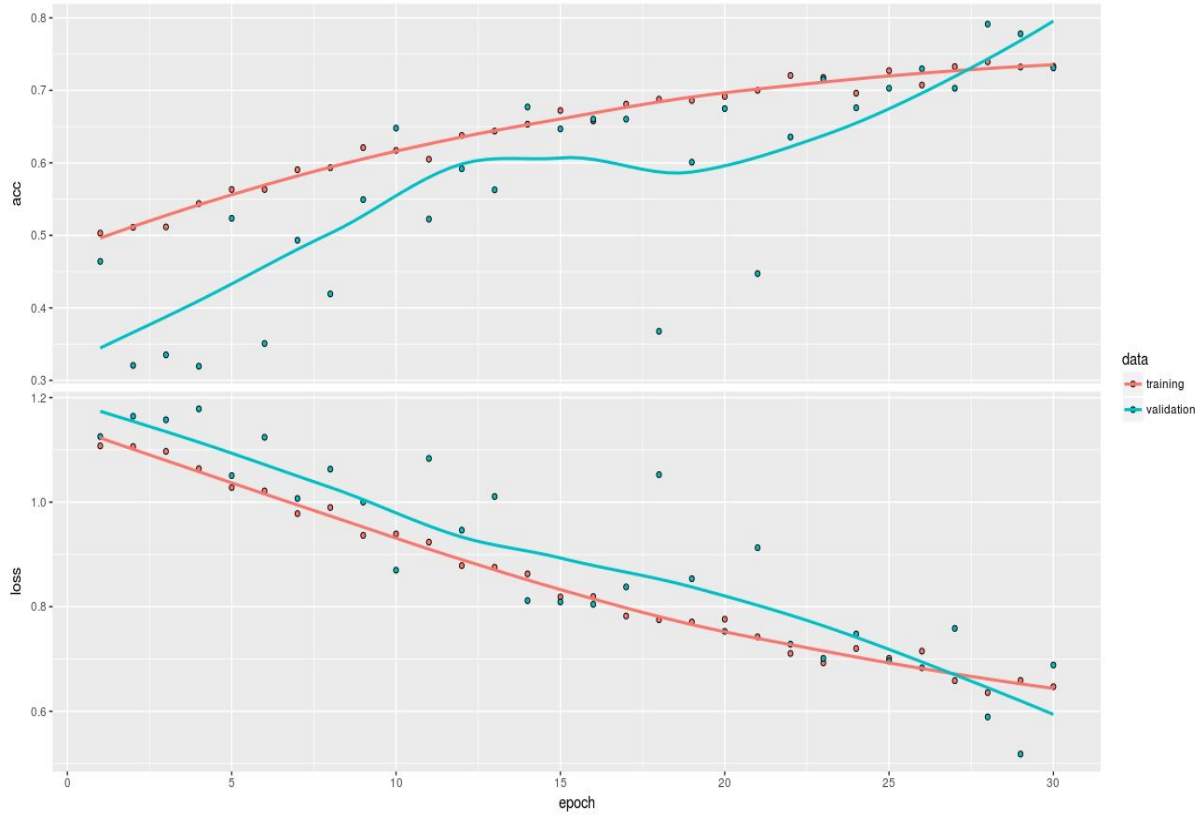


Ilustración 10. Plot de aumentación de datos

Transfer Learning

La idea principal del Transfer Learning consiste en usar el conocimiento que se ha aprendido de las distintas tareas para las que se tiene una inmensa cantidad de datos los cuales se encuentran etiquetados en entornos los cuales se caracterizan por tener pocos datos con etiquetas.

Esta creación de datos con etiquetas/características resulta bastante costosa por lo que se ha de aprovechar si o si los conjuntos de datos existentes de este tipo.
El aprendizaje de transferencia hace uso de los conocimientos que se han adquirido al dar una solución a un problema y aplicarlos a otro problema distinto pero que tenga relación con este.
En nuestro caso, cuando entrenamos nuestra red entrenamos todos los parámetros de la red neuronal y, por lo tanto, el modelo se aprende. Este proceso ha tomado incluso horas en la GPU.

Con el nuevo conjunto de datos podemos llevar a cabo el ajuste de la CNN preentrenada pero tenemos que tener en cuenta que el nuevo conjunto de datos es muy similar al conjunto de datos original que usamos en el preentrenamiento.
Como este es muy parecido, podemos usar los mismos pesos con el fin de extraer las características de nuestro nuevo conjunto de datos.

En esta parte es donde entra en juego el llamado Fine Tuning el cual explicaremos en el siguiente apartado.

Fine Tuning

El llama Fine Tuning es no más que otra técnica utilizada para la reutilización de modelos, complementaria a la extracción de características. Fine Tuning descongela algunas de las capas superiores de una base del modelo para la extracción de características y en el entrenamiento conjunto de la parte del modelo recién agregada y estas capas superiores.
Ajusta ligeramente las representaciones más abstractas del modelo que se está reutilizando, a fin de hacerlas más relevantes para el problema en cuestión.

Para mejorar aún más nuestro resultado anterior, podemos tratar de "ajustar" el último bloque convolucional del modelo VGG16 junto con el clasificador de nivel superior.
Los argumentos del modelo son:

- Weight. Se usa para especificar qué punto de control de peso inicializará el modelo → **imagenet.**
- include_top. Este argumento se refiere a incluir o no el clasificador densamente conectado en la parte superior de la red. → **FALSE**.
- input_shape, la forma de los tensores de imagen que alimentaremos a la red. → **(150,150,3)**

A continuación en la siguiente imagen podemos ver lo explicado anteriormente conjunto a uno de los modelos que se han creado para realizar las ejecuciones:

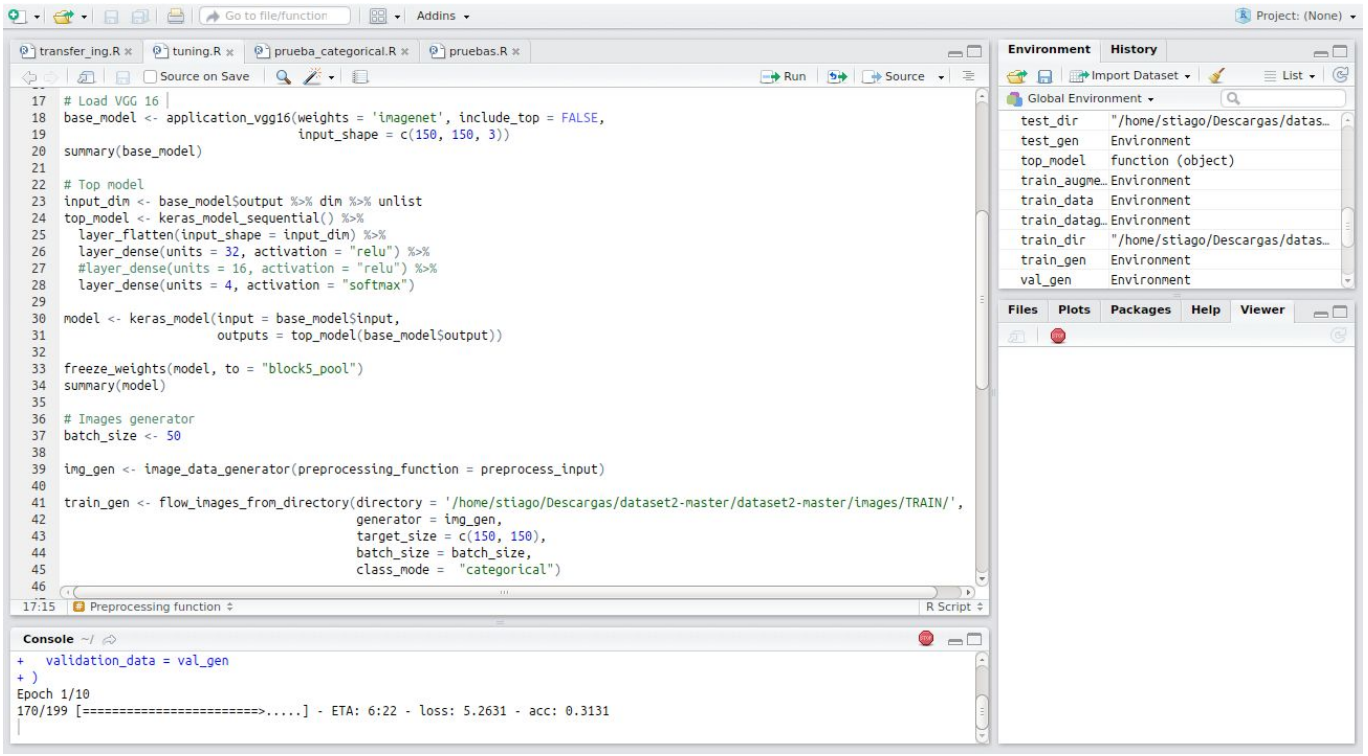


Ilustración 11. Ejemplo de Fine tuning

Aquí está el detalle de la arquitectura de la base convolucional VGG16

Fine tuning consiste en comenzar desde una red entrenada, luego volver a entrenarlo en un nuevo conjunto de datos usando actualizaciones de peso muy pequeñas. Por lo tanto, los pasos para ajustar una red son los siguientes:

- Agregar a la red personalizada sobre una red base ya entrenada; instancia de la base convolucional de VGG16.
- Frozen la red base.
- Entrena la parte que agregaste.
- Unfreeze algunas capas en la red base.
- Entrena conjuntamente estas capas y la parte que agregaste.

Debemos considerar lo siguiente para su afinamiento:

- Las capas anteriores en la base convolucional codifican características más genéricas y reutilizables, mientras que las capas superiores codifican características más especializadas.
- Es más útil afinar las características más especializadas, ya que estas son las que deben reutilizarse en nuestro nuevo problema. Habría rendimientos decrecientes en la puesta a punto de las capas inferiores.
- Elegimos ajustar solo el último bloque convolucional en lugar de toda la red para evitar el sobreajuste, ya que toda la red tendría una gran capacidad entrópica y, por lo tanto, una fuerte tendencia a sobreajustar. Las características aprendidas por los bloques convolucionales de bajo nivel son más generales, menos abstractas que las que se encuentran más arriba, por lo que es sensato mantener fijos los primeros bloques (características más generales) y solo afinar el último (características más especializadas).)

Empezamos ajustar nuestra red con el optimizador RMSprop, usando una tasa de aprendizaje baja, con el fin de limitar la magnitud de las modificaciones que hacemos a las representaciones de las capas que estamos afinando. También hemos implementado Fine Tunning con el optimizador SGD así para asegurarse de que la magnitud de las actualizaciones sea muy pequeña, para no arruinar las características aprendidas previamente.

A continuación se muestra con la siguiente captura un ejemplo del ajuste realizado para el optimizador RMSprop:

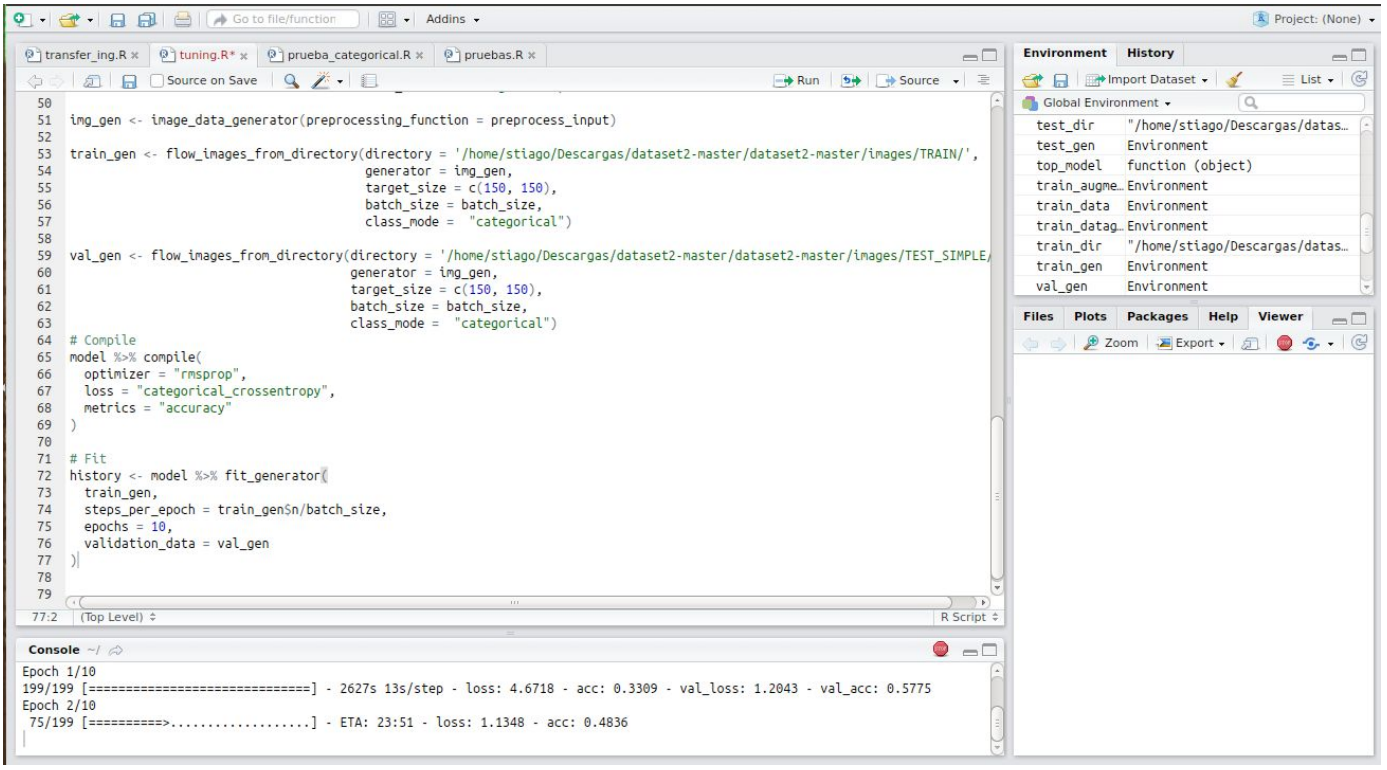


Ilustración 12. Optimizador RMSprop

Se ha de mencionar también que debido a la poca capacidad que tiene nuestro computador, no se ha podido ejecutar tuning con mayor a 6 épocas ya que se cerraba el programa inesperadamente. En la siguiente captura podemos apreciar como los resultados por cada época mejoraban considerablemente con respecto a los anteriores.

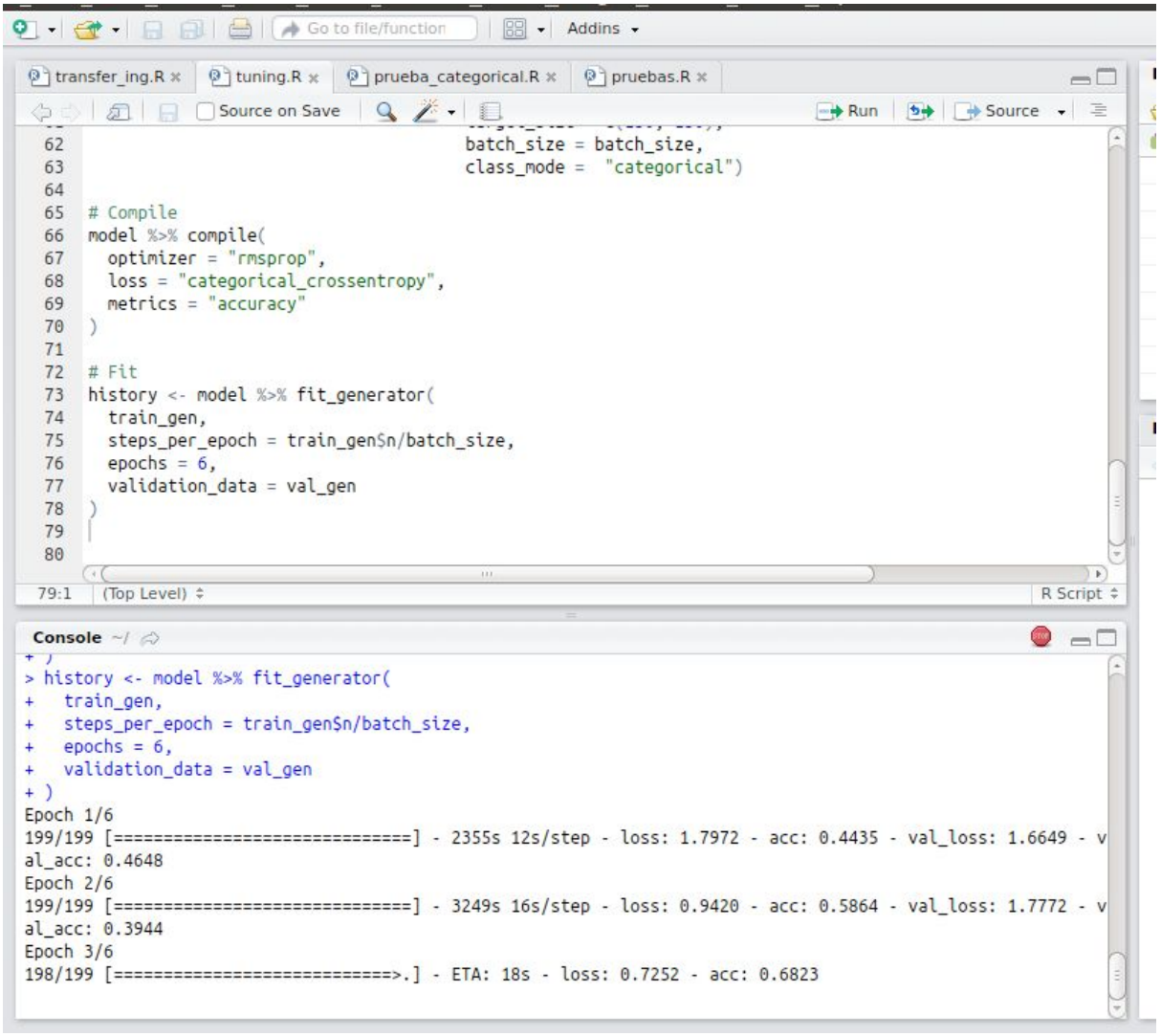


Ilustración 13. Fine Tuning

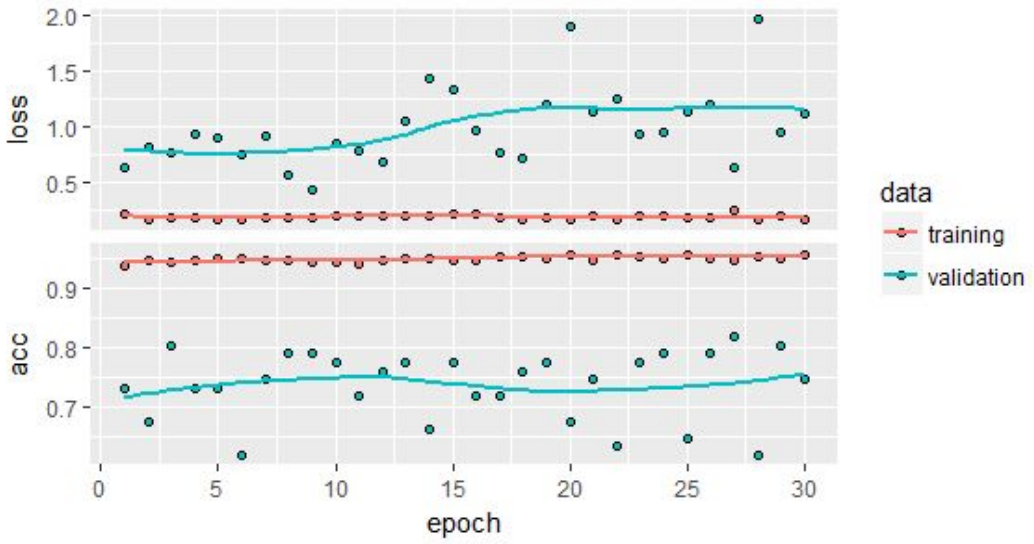
Discusión de resultados

Inicialmente se ha trabajado partiendo de un modelo dummy a partir del cual se han ido introduciendo capas y realizando mejoras.

Al ser dos personas los miembros del equipo se han dividido las distintas posibilidades a probar e investigar por lo que finalmente se han tenido dos modelos los cuales han sido compilados y entrenados separadamente para al final escoger el que mejor resultado nos ha proporcionado.

Los modelos creados desde el inicio, como se ha comentado anteriormente, han sido numerosos para ello se han ido variando sus capas, las formas de activación y demás parámetros, también se han realizado numerosos entrenamientos de los cuales finalmente se ha obtenido el mejor resultado de accuracy compilando el modelo con el optimizador rmsprop.

Se ha visto también que los mejores resultados debido a que es un problema multiclase, han sido dados por la clasificación usando categorical_crossentropy y softmax, con una tasa de aprendizaje especificada en el modelo del 0.2. El mejor resultado se ha obtenido en un entrenamiento donde se han fijado los parámetros steps_per_epoch, epochs y validation steps en los siguientes valores:




```
# Entrenamiento
history <- model %>%
 fit_generator(
        train_data,
        steps_per_epoch = 150,
        epochs = 80,
        validation_data = validation_data,
        validation_steps = 50
 )
```

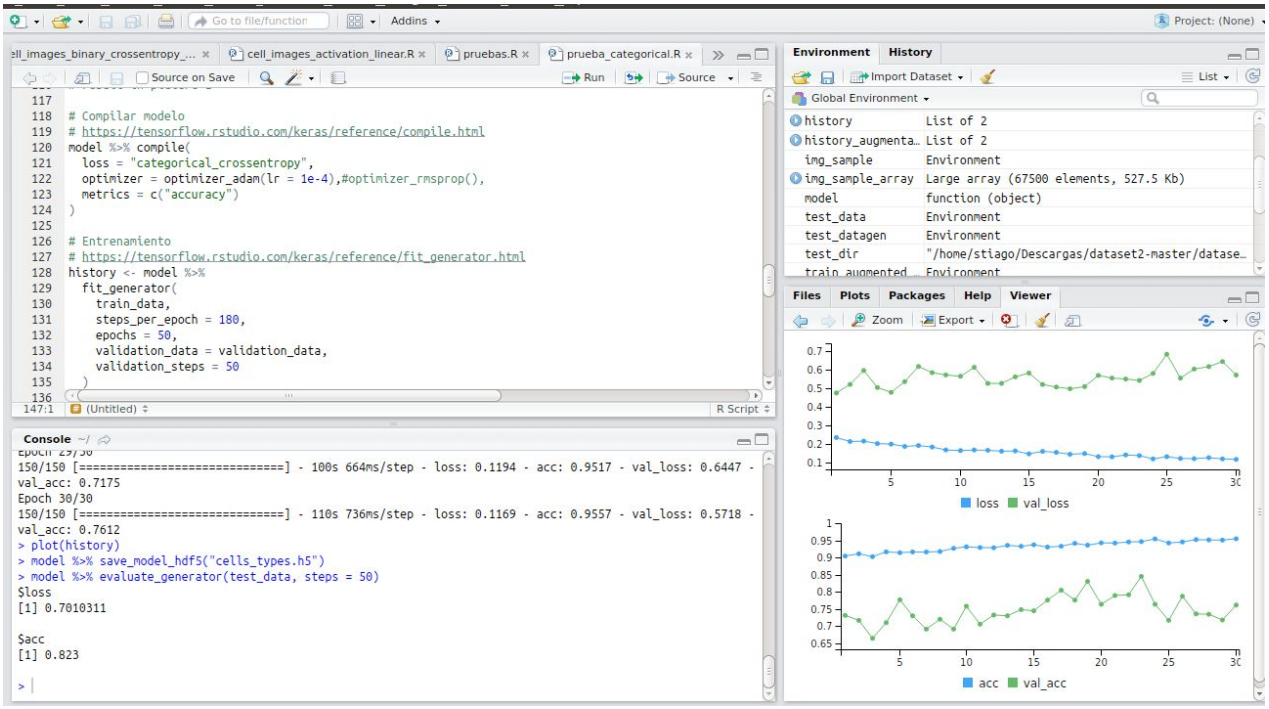
Se podría decir que a mayor número de épocas se ha obtenido un mejor resultado y que además visualizando con plot se puede ver en la gráfica que posiblemente el valor de accuracy seria mejor con más casos de uso disponibles en el set.

Los resultados han sido distintos dependiendo por ejemplo de los siguientes argumentos especificados en los siguientes apartados:

- Optimizadores

- Optimizador Adam:

Con este optimizador como se puede apreciar en la imagen, no se han obtenido muy buenos resultados ya que se ha obtenido un 0.82 de accuracy y un 0.78 de loss.



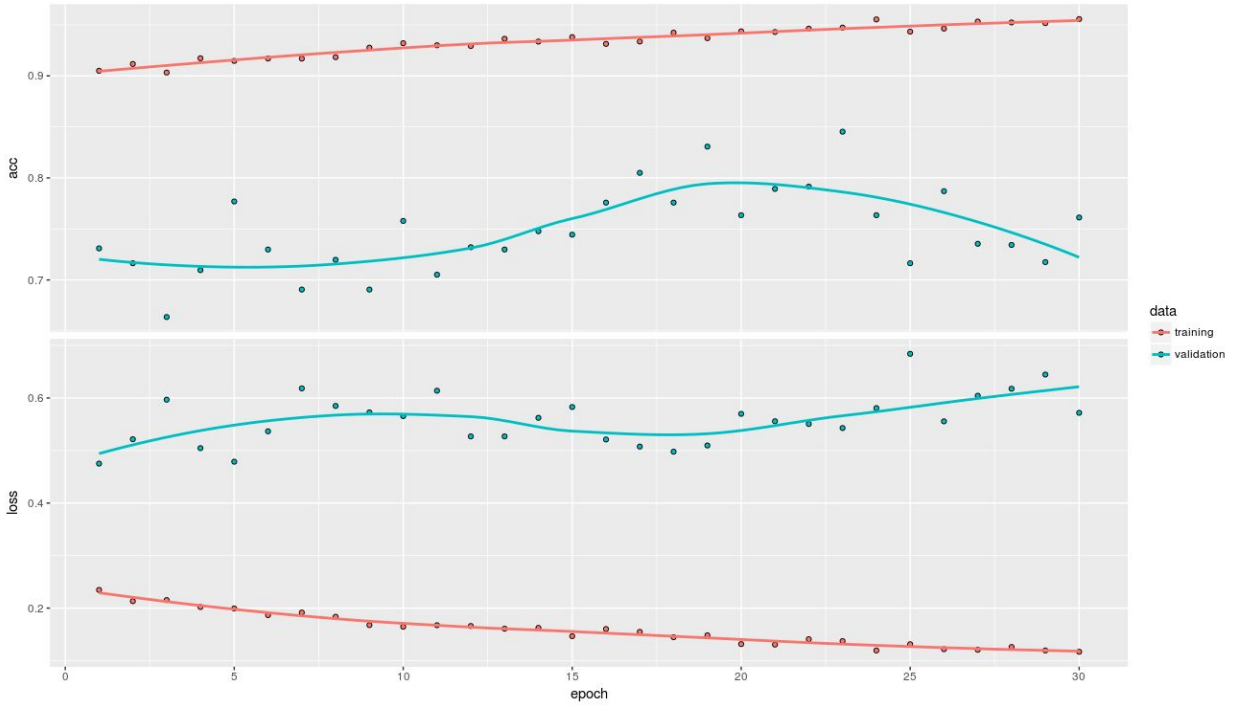


Ilustración 15. Resultados y Plot Adam

-Optimizador rmsprop:

Con dicho optimizador los resultados obtenidos han sido los siguientes:

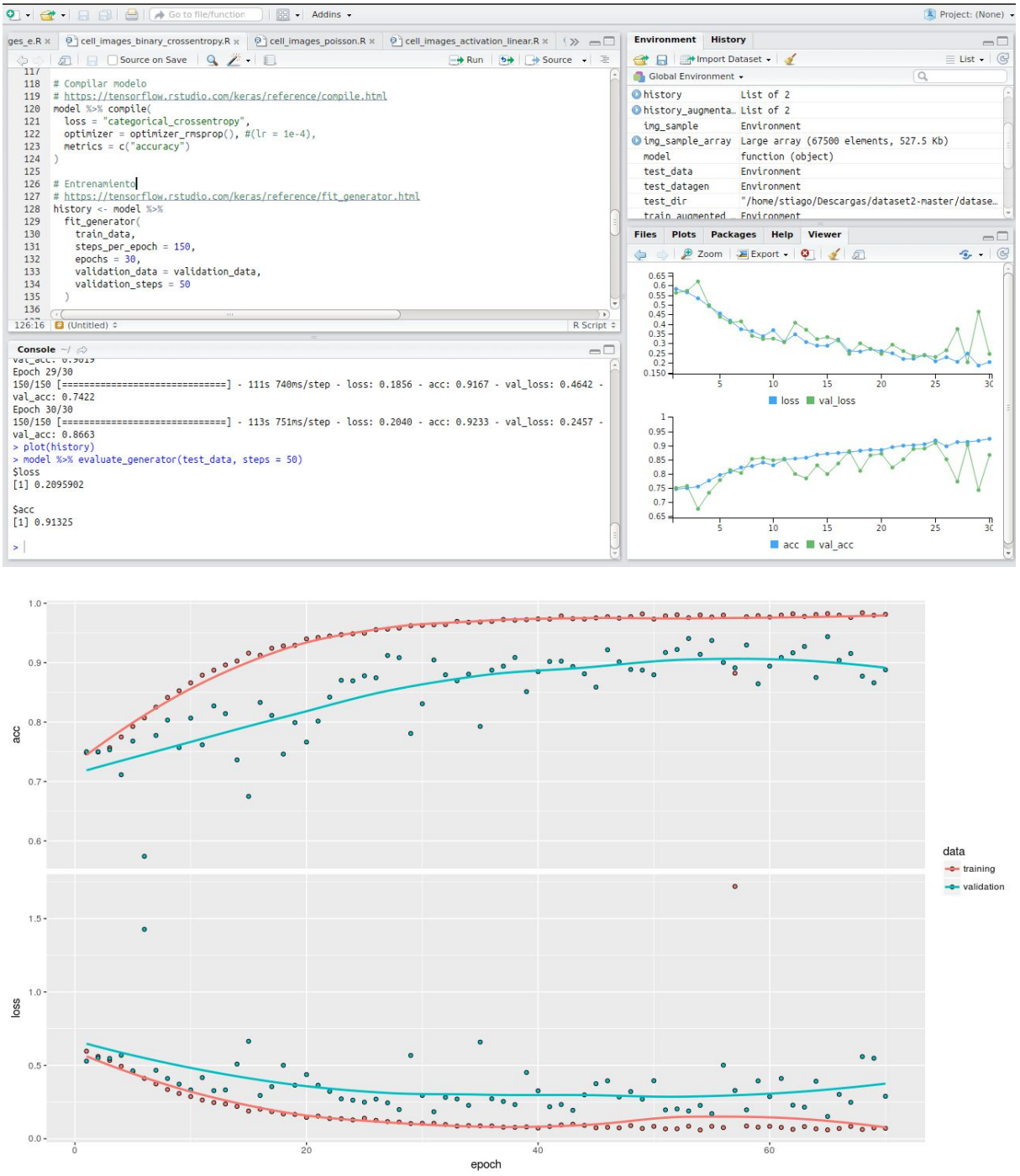
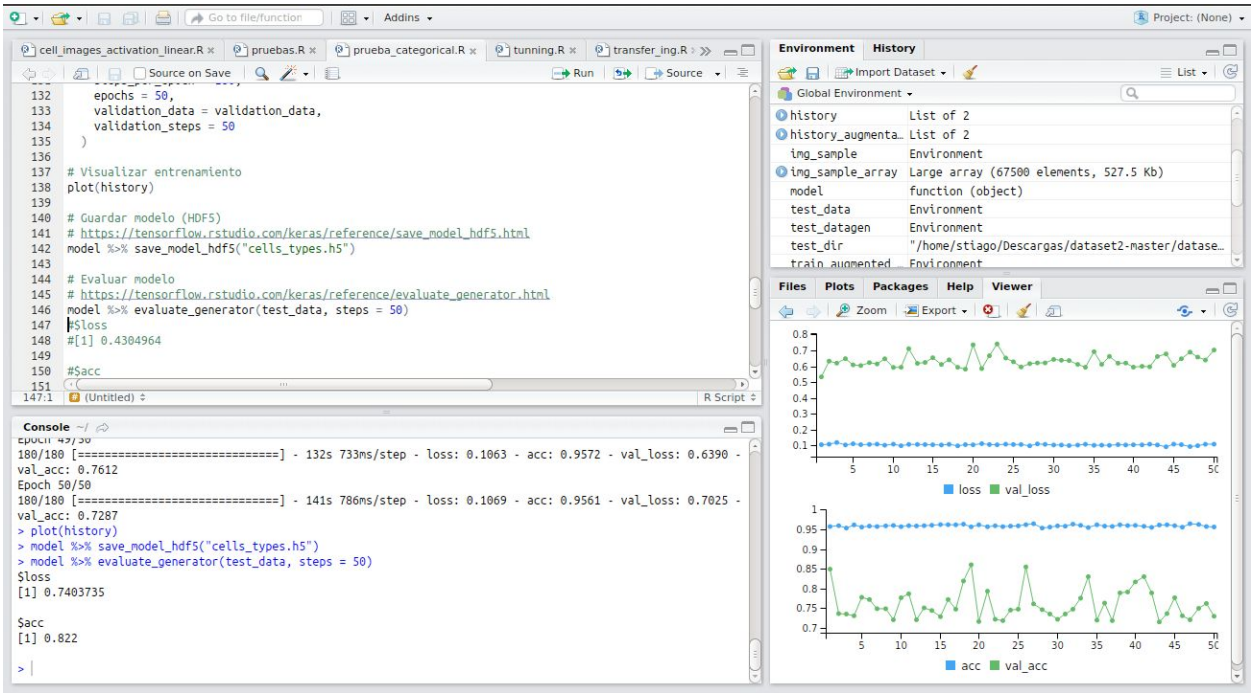


Ilustración 16. Rsultados y Plot de RMSprop

Se ha de comentar que este ha sido el optimizador que mejores resultados nos ha dado ya que se ha obtenido un 0.91 de accuracy y un 0.2 de loss.

-Optimizador SGD:

Ejemplo de resultados con el optimizador sgd donde se aprecia un accuracy del 0.82:



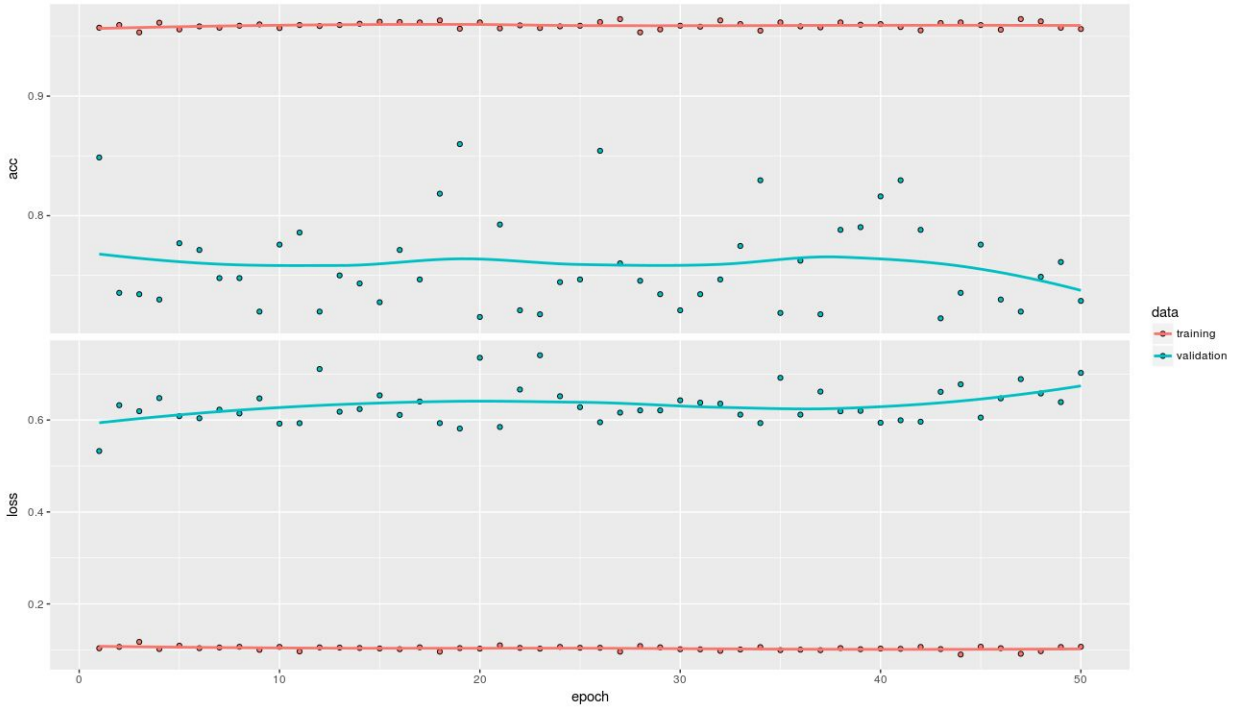


Ilustración 17. SGD

- Data Augmentation:

  Cuando se ha aplicado la aumentación los resultados han sido los siguientes:

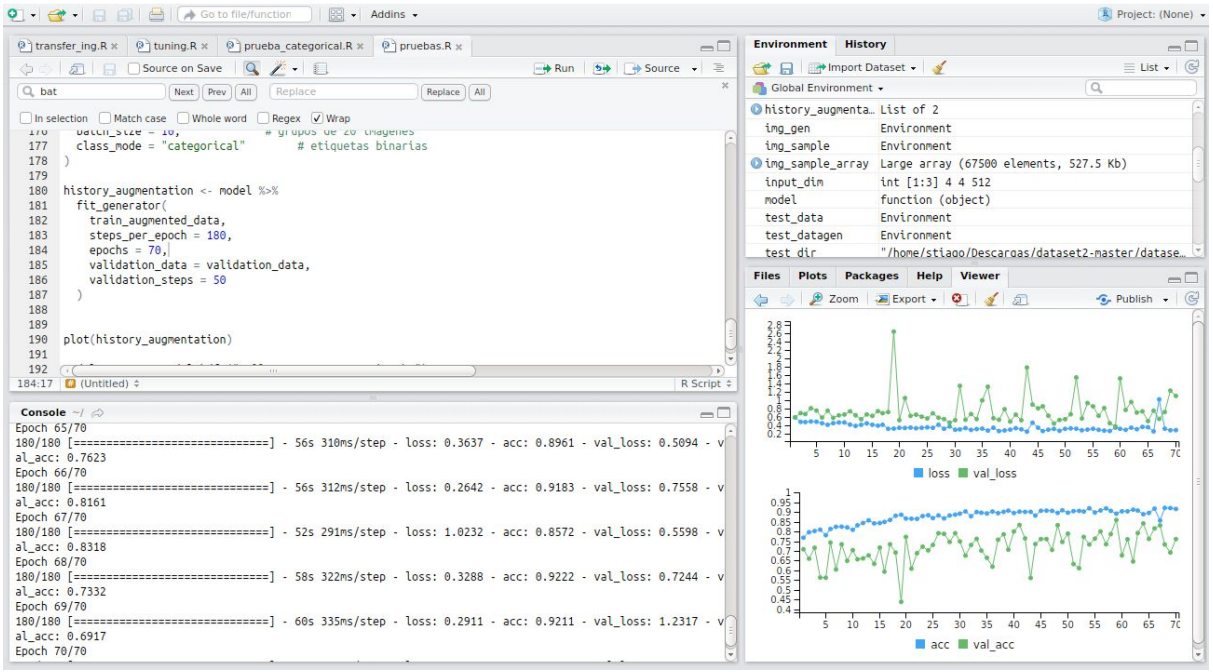


Ilustración 18. Aumentación de datos

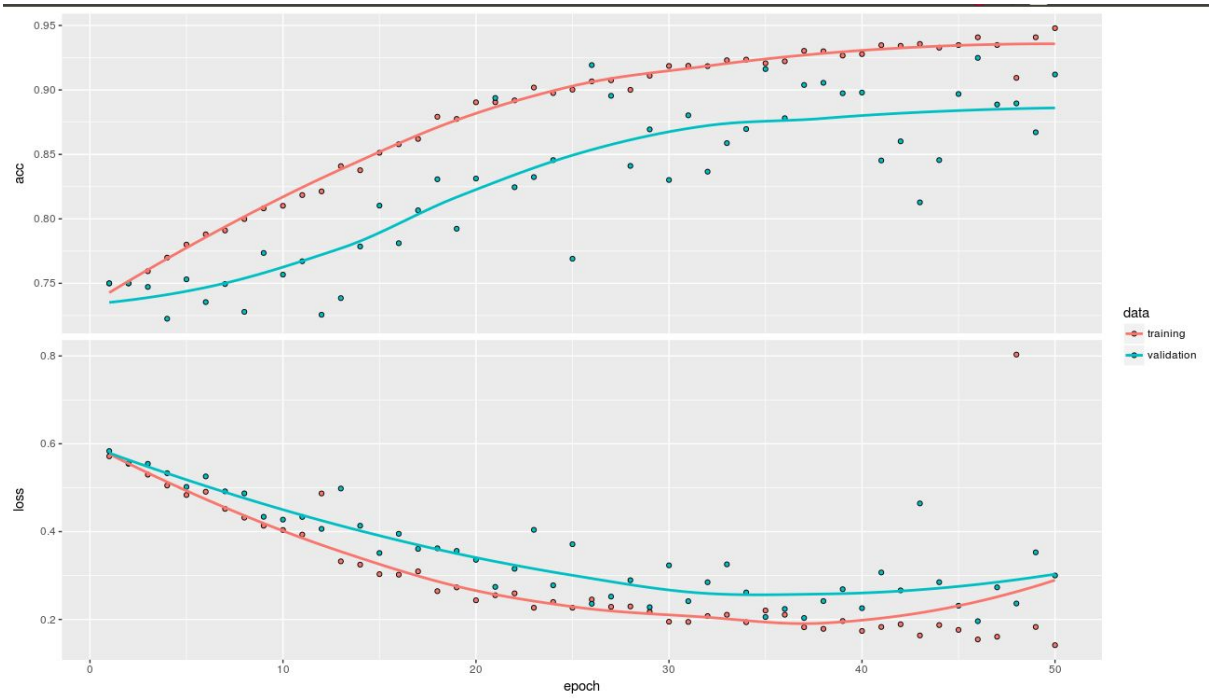


Ilustración 19. Plot de aumentación de datos

Donde los mejores resultados que se han obtenido han sido de 0.921 de accuracy y 0.21 de loss.

- Transfer Learning & Fine Tuning

En esta apartado se han cargado el modelo VGG16 y se han ido ajustando los parámetros de tal forma que finalmente se han obtenido resultados incompletos ya que debido a los equipos en los que se ha ejecutado, el programa se ha cerrado en todas las pruebas de forma inesperada llegando únicamente a la etapa 4 en el caso mejor. Los resultados interrumpidos se muestran las siguientes capturas de pantalla donde se puede apreciar que en cada etapa el accuracy experimentaba una mejora muy considerable con respecto a la etapa anterior.

El modelo ha sido el siguiente:

```
> summary(base_model)
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 150, 150, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 150, 150, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 150, 150, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 75, 75, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 75, 75, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 75, 75, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 37, 37, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 37, 37, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 37, 37, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 18, 18, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 18, 18, 512)       1180160
```

block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

==========

Total params: 14,714,688

Trainable params: 14,714,688

Non-trainable params: 0

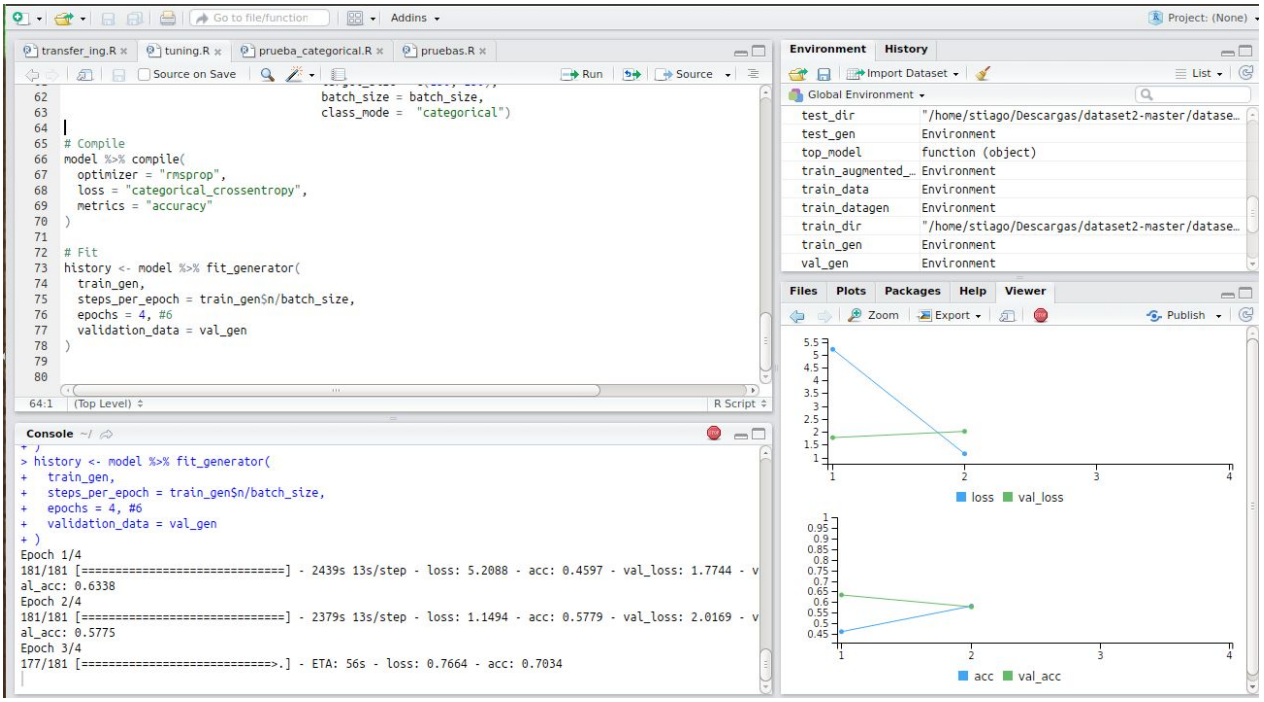


Ilustración 20. Fine Tuning

Justo minutos después al empezar la cuarta etapa en todos los experimentos se bloqueo el programa por lo que no se pudo llevar más lejos la investigación con Fine tuning.

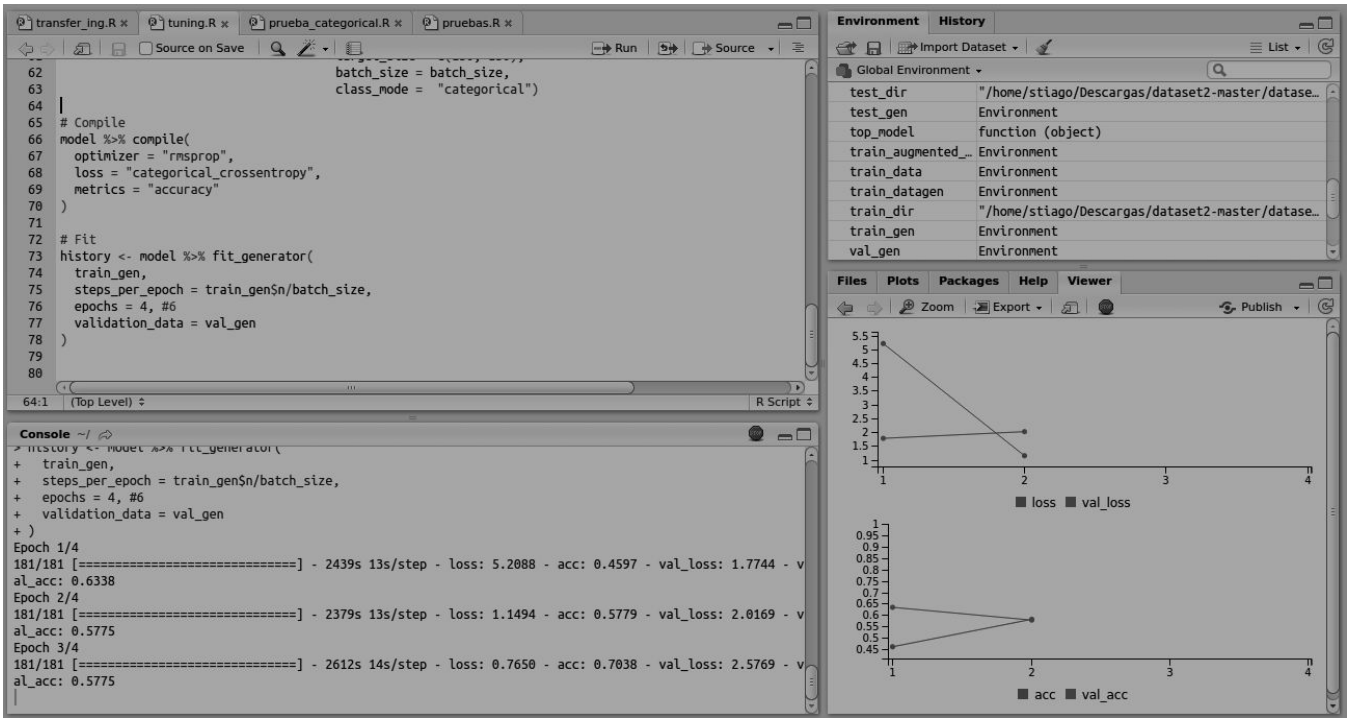


Ilustración 21. Fine Tuning

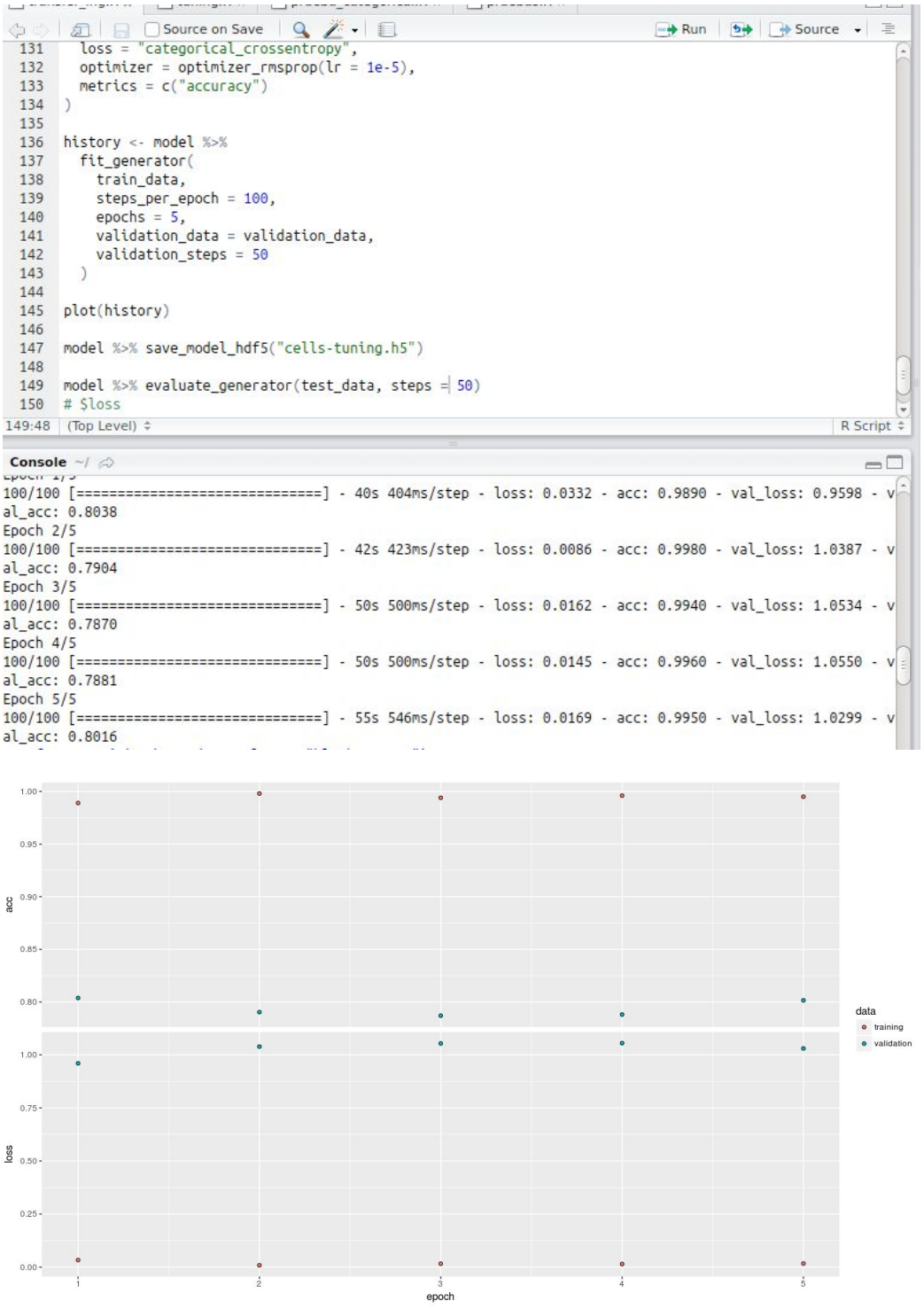


Ilustración 22. Fine Tuning

Se ha de notar que en esta última prueba en otro computador si termino el fine tuning con 5 épocas.

Conclusiones

Se ha de concluir comentando que el trabajo realizado ha sido bastante laborioso debido a que las máquinas en las que se han realizado las ejecuciones no contaban de una buena GPU por lo que esto ha hecho que el tiempo de ejecución tomase hasta 4 horas en algunos casos. También se ha de mencionar que han sido numerosos los problemas que se han encontrado en el camino debido al mismo programa RStudio el cual en numerosas ocasiones dejó de funcionar llevando desde su reinicio por bloqueo hasta su reinstalación completa varias veces.

Aun así, después de todos los problemas que han ido surgiendo a lo largo de la práctica, se ha de concluir que los resultados obtenidos han sido buenos, mejorables si se hubiese contado con más tiempo debido a la prolongada ejecución de cada una de las evaluaciones.

Se ha iniciado con modelos ejecutados finalmente con el optimizador rmsprop el cual ha sido el que mejor resultado nos ha dado al principio con respecto al accuracy y el loss, seguidamente se ha aplicado data augmentation debido a que visualizando los gráficos se ha percibido una tendencia a incrementar este accuracy por lo que aumentando los casos de usos podemos ver realmente el valor de este con nuestro modelo.

Seguidamente se ha aplicado Fine Tuning el cual se preveía que diese buenos resultado pero debido a los equipos como se viene comentando en el apartado anterior, no se ha podido obtener el resultado final.

Bibliografía

https://keras.rstudio.com/reference/flow_images_from_directory.html

https://towardsdatascience.com/decision-tree-ensembles-bagging-and-boosting-266a8ba60fd9

https://towardsdatascience.com/decision-tree-ensembles-bagging-and-boosting-266a8ba60fd9

https://towardsdatascience.com/how-to-implement-deep-learning-in-r-using-keras-and-tensorflow-82d135ae4889

https://towardsdatascience.com/ensembling-convnets-using-keras-237d429157eb

http://neuralnetworksanddeeplearning.com/chap6.html

https://www.learnopencv.com/

https://topepo.github.io/rsample/articles/Applications/Keras.html

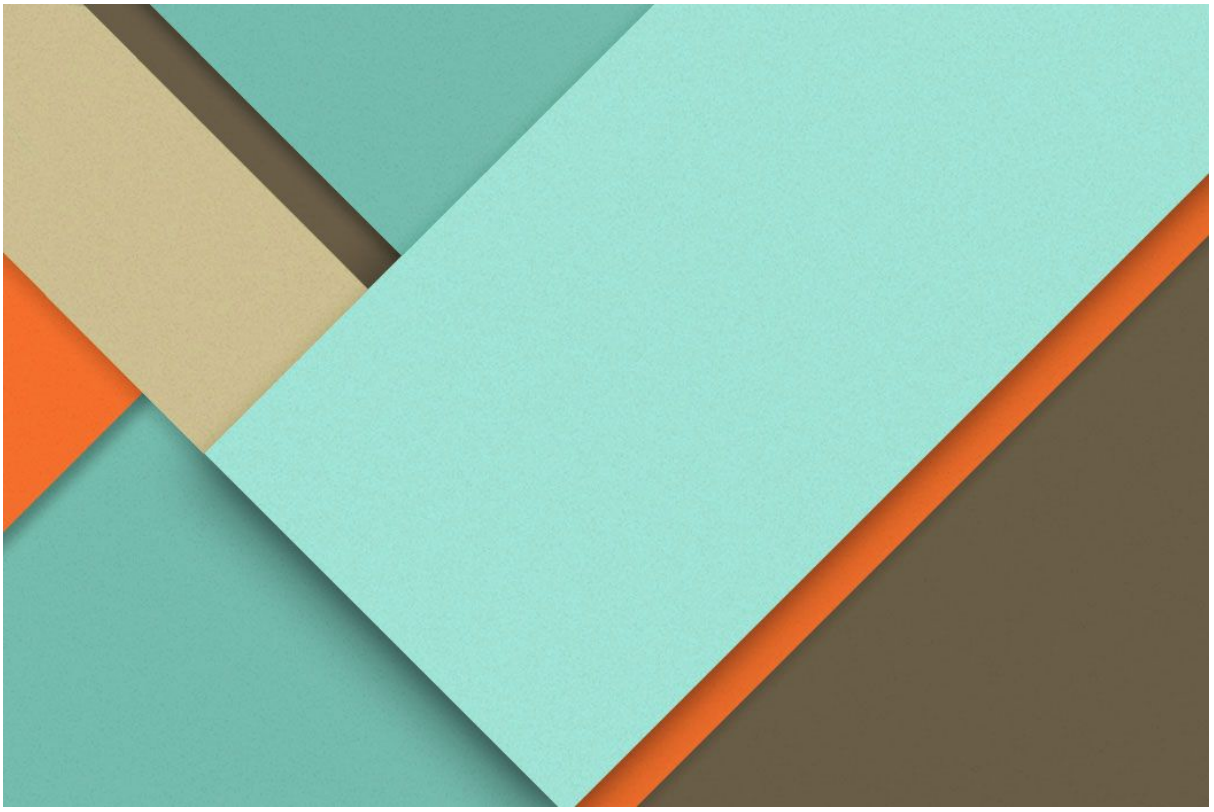
https://keras.io/getting-started/sequential-model-guide/

https://www.datacamp.com/community/tutorials/keras-r-deep-learning

https://machinelearningmastery.com/image-augmentation-deep-learning-keras/

https://towardsdatascience.com/what-is-transfer-learning-8b1a0fa42b4

https://tensorflow.rstudio.com/keras/



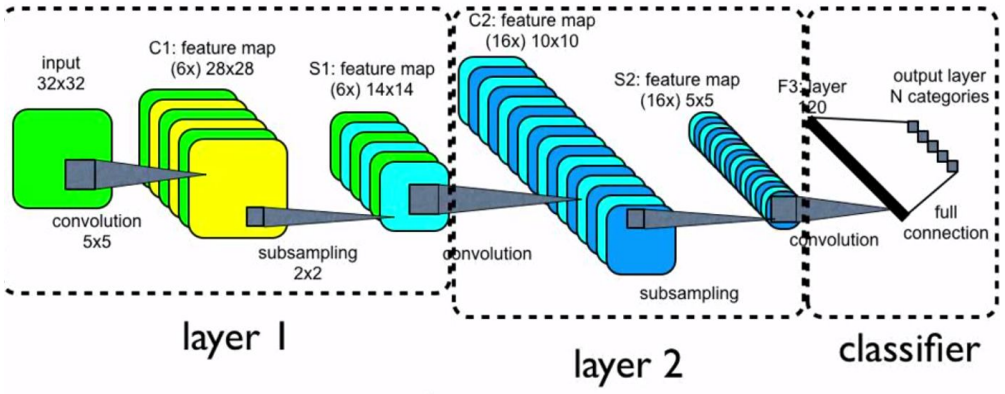
Trabajo de investigación: Deep Learning

Curso 2017-2018

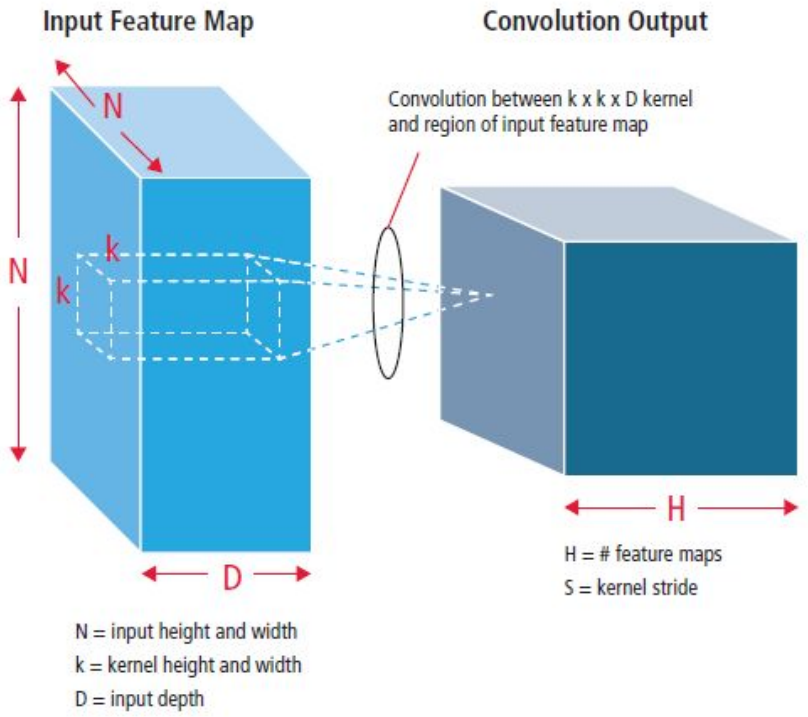
Maria Victoria Santiago Alcalá - Dayana Aguirre Iñiguez
Sistemas Inteligentes para la Gestión en la Empresa
Universidad de Granada
Granada, Junio 2018

Feature map

El mapa de características es la salida de un filtro aplicado a la capa anterior. Se dibuja un filtro dado en toda la capa anterior, se mueve un píxel a la vez. Cada posición da como resultado una activación de la neurona y la salida se recoge en el mapa de características. Puede ver que si el campo receptivo se mueve un píxel desde la activación hasta la activación, entonces el campo se propondrá con la activación previa mediante (ancho de campo - 1) valores de entrada.



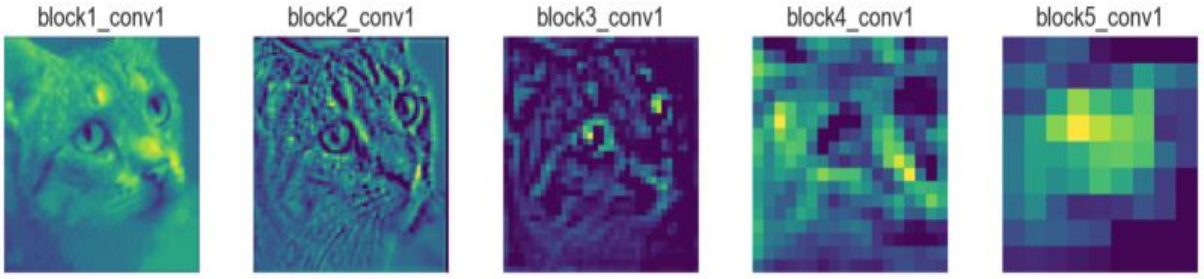
Por ejemplo, en una imagen de 32 × 32, arrastrando el campo receptivo de 5 × 5 a través de los datos de imagen de entrada con un ancho de paso de 1 se obtendrá un mapa de características de 28 × 28 (32-5 + 1 × 32-5 + 1) valores de salida o 784 activaciones distintas por imagen.



El tamaño del Feature map está controlado por tres parámetros que debemos decidir antes de realizar el paso de convolución:

- Profundidad: la profundidad corresponde a la cantidad de filtros que usamos para la operación de convolución.
- Stride: Stride es el número de píxeles con los que deslizamos nuestra matriz de filtro sobre la matriz de entrada. Mientras mayor sea el número se producirá mapas de características más pequeñas.
- Zero-padding: a veces, es conveniente rellenar la matriz de entrada con ceros alrededor del borde, de modo que podamos aplicar el filtro a los elementos que bordean nuestra matriz de imagen de entrada. Una buena característica del relleno cero es que nos permite controlar el tamaño de los mapas de características. Agregar zero-padding también se llama convolución amplia, y no usar zero-padding sería una convolución estrecha.

Los mapas de características también se llaman activaciones intermedias ya que la salida de una capa se llama activación. Hay algunas observaciones interesantes sobre los mapas de características a medida que avanzamos por las capas.



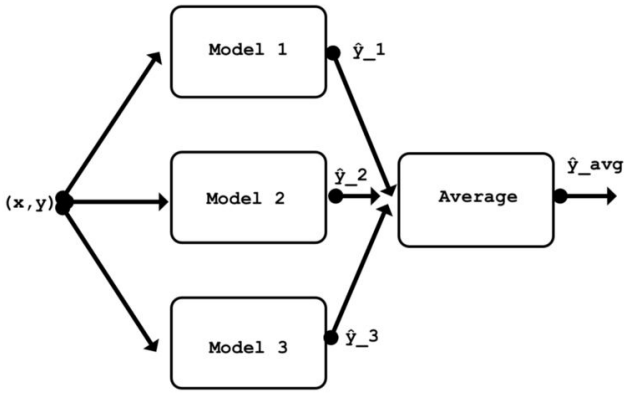
Los mapas de características de la primera capa (block1_conv1) retienen la mayor parte de la información presente en la imagen. En las arquitecturas CNN, las primeras capas suelen actuar como detectores de bordes.

A medida que profundizamos en la red, los mapas de características se parecen menos a la imagen original y más como una representación abstracta de la misma. Como puede ver en block3_conv1, el gato es algo visible, pero después de eso se vuelve irreconocible. La razón es que los mapas de características más profundos codifican conceptos de alto nivel como "nariz de gato" u "oreja de perro", mientras que los mapas de características de nivel inferior detectan bordes y formas simples. Es por eso que los mapas de funciones más profundos contienen menos información sobre la imagen y más sobre la clase de la imagen. Aún codifican características útiles, pero son menos interpretables visualmente por nosotros.

Los mapas de características se vuelven más escasos a medida que profundizamos, lo que significa que los filtros detectan menos características. Tiene sentido porque los filtros en las primeras capas detectan formas simples, y cada imagen contiene esas. Pero a medida que profundizamos, comenzamos a buscar cosas más complejas como "cola de perro" y no aparecen en todas las imágenes.

Ensembling ConvNets

Se lo llama así cuando se usa varios modelos diferentes en vez de solo uno, que combina varios árboles de decisión para producir un mejor rendimiento predictivo que utilizar un solo árbol de decisión. Ensemble es un procedimiento potente que mejora el rendimiento de una sola red, reduce la porción de varianza en la descomposición de polarización-varianza del error de predicción. Mejorando así el accuracy de precisión junto con una reducción en el tiempo de entrenamiento. Hay muchos tipos diferentes de ensembles; stacking es uno de ellos. Es uno de los tipos más generales y puede representar teóricamente cualquier otra técnica de ensembles. Stacking implica el entrenamiento de un algoritmo de aprendizaje para combinar las predicciones de varios otros algoritmos de aprendizaje.



En los últimos años, varios documentos han demostrado que ensembles puede ofrecer un rendimiento excepcional en la reducción del error de prueba. Más en particular, (Krizhevsky et al., 2012) mostró que en la clasificación de ImageNet 2012 punto de referencia, su modelo de conjunto con 5 convnets logró un top-1 tasa de error de 38.1%, en comparación con la tasa de error superior de 40.7% dada por el modelo individual. Breiman presentó el concepto de bagging, que nos ayudó a ganar una cierta comprensión de por qué el ensemble del árbol de clasificación y el árbol de regresión funcionan cuando fueron entrenados por muestras aleatorias de todo el conjunto de datos (Breiman, 1996).

Vamos a hablar de algunas técnicas para realizar árboles de decisión de ensambles:

- **Bagging**: se usa cuando nuestro objetivo es reducir la varianza de un árbol de decisiones. Aquí la idea es crear varios subconjuntos de datos de la muestra de entrenamiento elegida al azar con reemplazo. Ahora, cada colección de datos de subconjuntos se usa para entrenar sus árboles de decisión. Como resultado, terminamos con un conjunto de diferentes modelos. Se usa el promedio de todas las predicciones de diferentes árboles, que es más sólido que un solo árbol de decisión.

  Random Forest es una extensión sobre el bagging. Se necesita un paso adicional en el que, además de tomar el subconjunto aleatorio de datos, también se tome la selección aleatoria de características en lugar de utilizar todas las características para hacer crecer los árboles.

  Ventajas de usar la técnica Random Forest:

  - ❖ Maneja muy bien los datos de mayor dimensionalidad.
  - ❖ Maneja los valores perdidos y mantiene la precisión de los datos faltantes.

  Desventajas del uso de la técnica Random Forest:

  - ❖ Como la predicción final se basa en las predicciones medias de los árboles subconjuntos, no proporciona valores precisos para el modelo de regresión.

- **Boosting**: Ajustar árboles consecutivos (muestra aleatoria) y en cada paso, el objetivo es resolver el error neto del árbol anterior. Cuando una entrada se clasifica erróneamente mediante una hipótesis, su peso aumenta, por lo que es probable que la próxima hipótesis la clasifique correctamente. Al combinar todo el conjunto al final, convierte a los árboles débiles en modelos de mejor rendimiento. Gradient Boosting es una extensión sobre el método de refuerzo = Gradient Descent + Boosting; Utiliza un algoritmo de descenso de gradiente que puede optimizar cualquier función de pérdida diferenciable. Un conjunto de árboles se construye uno

por uno y los árboles individuales se suman de forma secuencial. El siguiente árbol intenta recuperar la pérdida (diferencia entre los valores reales y los pronosticados).

Ventajas de usar la técnica de Gradient Boosting:

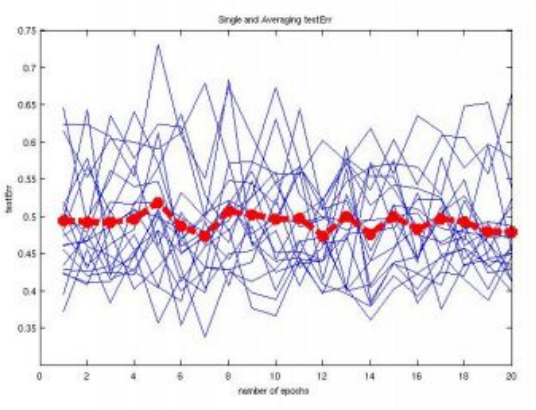
- ❖ Admite diferentes funciones de pérdida.
- ❖ Funciona bien con interacciones.

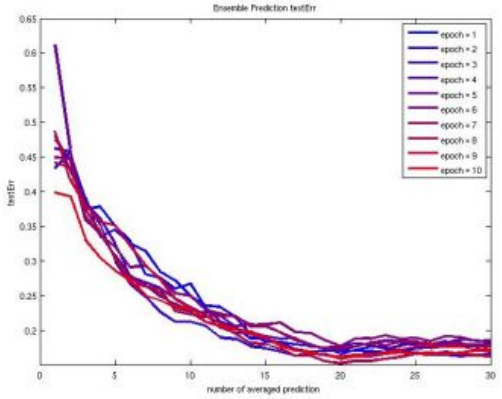
Desventajas de usar la técnica de Gradient Boosting:

- ❖ Propenso al exceso de ajuste.
- ❖ Requiere ajuste cuidadoso de diferentes hiperparámetros

Lo explicaremos mejor con una arquitectura que está compuesta por tres capas: dos capas convolucionales(H1, H2) y una capa completamente conectada(H3). El resultado de la capa completamente conectada se alimenta a una capa de Softmax que produce un vector de longitud 10. Ahora entrenamos la red con 20 epocas.

Con un resultado numérico y como un procedimiento adecuado de ensemble learning seria promediando la predicción sobre los predictores entrenados bajo el mismo número de épocas; mientras la medida de predictores promedio aumenta, la tasa de error de prueba se reduce, lo que indica que el conjunto promediando de la predicción contribuye a lograr un mejor rendimiento.

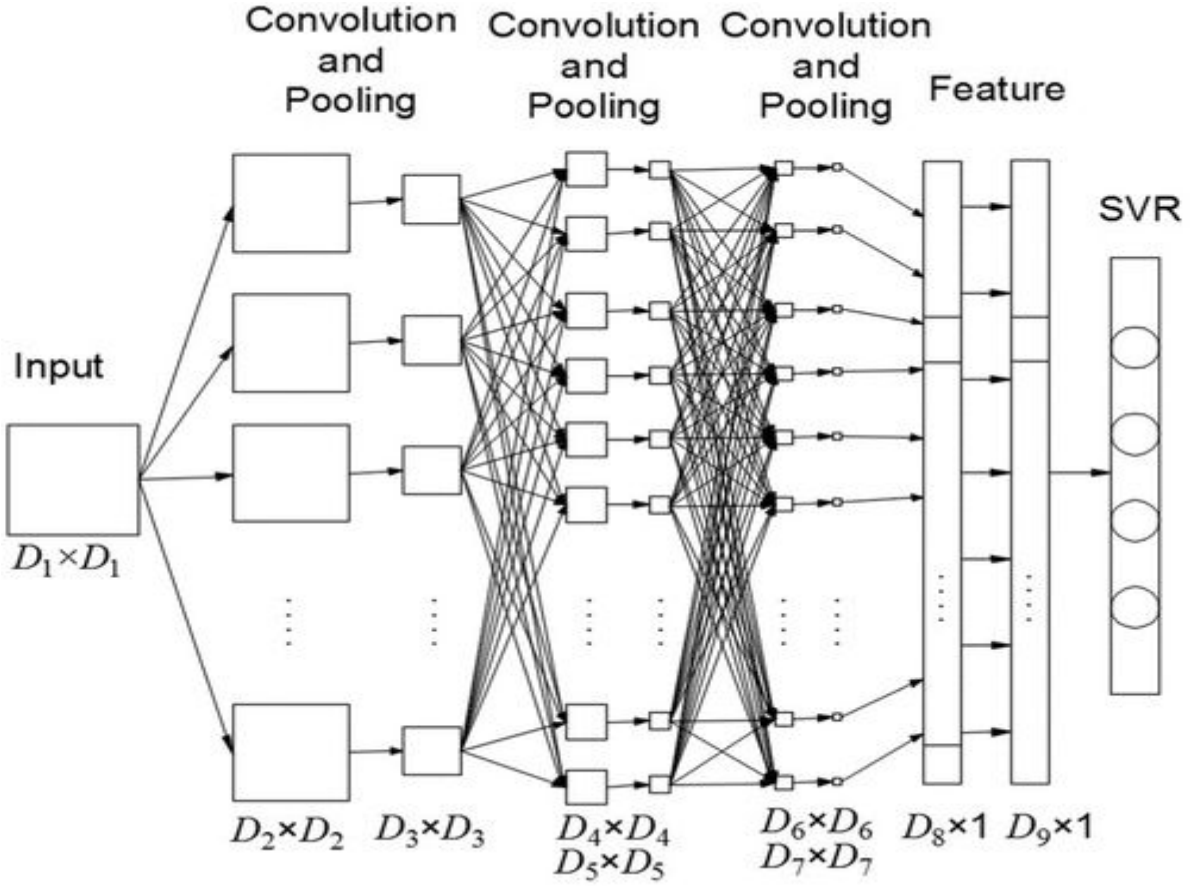




Hybrid CNN

Las redes HCNN combinan la idea original de las redes convolucionales de LeCun con los beneficios de las neuronas similares a RBF en todas las capas. El modelo híbrido propuesto es un algoritmo de combinación basado en las teorías CNN y SVR. La parte CNN se usa como un extractor de funciones y la SVR como un clasificador. En este modelo, durante la construcción, la capa superior de la CNN tradicional se reemplaza con un clasificador SVR.

En este modelo se apila capa por capa con capas convolucionales y capas de agrupamiento en el interior. Como se muestra en la Figura, la estructura se combinó de 10 capas en total, incluyendo la capa de entrada, tres capas convolucionales, tres capas de agrupación, dos capas con conexión completa y un clasificador regresivo de vector de soporte como la capa superior.



En el modelo particular CNN-SVR, los parámetros se establecen de la siguiente manera: el tamaño de capa de entrada D1 es 32, 5 núcleos de filtro con tamaño 5x5 para la primera capa convolucional, 10 núcleos de filtro con tamaño 5x5 para la segunda capa convolucional , 10 núcleos de filtro con tamaño 2 × 2 para la tercera capa convolucional, el tamaño de tres capas de agrupación son todos 2 × 2 y el tamaño de la segunda capa de conexión completa establecida como D9 = 100.

La señal sin procesar del conjunto de datos se usa como entrada. Al comienzo de la etapa de capacitación, los datos brutos del conjunto de datos original se transforman en una matriz con varias dimensiones, y luego la matriz se ingresa a la primera capa visible. La capa de agrupación junto a la capa convolucional usa la regla de agrupación máxima, es decir, la capa de agrupación selecciona los coeficientes más grandes sobre cada celda de muestreo para reducir las dimensiones a la mitad. Otras capas convolucionales y de agrupación se construyen a través de los mismos procedimientos. La salida de la última capa de agrupación se ingresa a una capa de conexión completa con una salida de 100 dimensiones. Además, esta salida se usa para extraer características de las señales originales. El proceso de aprendizaje de esta capa por red capa por capa se puede reduplicar muchas veces según los requisitos. Posteriormente, el mapa de características que se ingresará al clasificador SVR puede contener información más útil.

Con Fine tunning se lleva a cabo mediante el algoritmo BP. En detalle, el SGD supervisado se usa para ajustar los parámetros del modelo híbrido. Específicamente, primero calcula el error entre el vector de salida y el vector objetivo real, luego obtiene la función de pérdida y el error de los parámetros. Usando el algoritmo SGD para obtener los gradientes de la función de pérdida y actualizar los pesos en las capas correspondientes. Además, el SGD puede reducir aún más el error de entrenamiento y promover la precisión de la clasificación. En este estudio, se acepta el descenso de gradiente estocástico con minilotes (MSGD) para hacer que

el proceso de aprendizaje sea más eficiente.

Bibliografía

- http://theorycenter.cs.uchicago.edu/REU/2014/final-papers/chen.pdf
- https://towardsdatascience.com/ensembling-convnets-using-keras-237d429157eb
- https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2
- http://journals.sagepub.com/doi/full/10.1177/1687814017704146