Redes Neuronales con Keras

Rafael Jordá Muñoz

18 de enero de 2019

Contents

1 Introducción

Las redes neuronales pueden utilizarse en un gran número y variedad de aplicaciones. En particular, en este caso se van a usar para el reconocimiento de imágenes. El objetivo es aplicar las redes neuronales a cifar10, una base de datos que contiene imágenes de aviones, automóviles, pájaros, etc; en total 10 grupos. Dada una imagen, la red deberá clasificarla en uno de esos grupos.

El objetivo de este proyecto es evaluar las ventajas de una red de convolución con respecto a una red MLP y también erradicar el posible overfitting que aparezca utilizando alguna regularización.

Se va a llevar, en general, un enfoque incremental en complejidad; esto es, empezamos con modelos "sencillos" aumentando poco a poco la complejidad; conforme aumente dicha complejidad irá haciendo aparición el sobreentrenamiento y ahí, aplicaremos los mecanismos de regularización.

Se va a trabajar con un conjunto de train de 10000 ejemplares para que los entrenamientos lleven menos tiempo. Posteriormente, cuando hayamos seleccionado ya un modelo final; se entrenará con toda la base de datos de cifar10 y se evaluará con todo el test proporcionado también por cifar10.

2 Multilayer Perceptron

Comenzamos viendo el Multilayer Perceptron. Primero explicaremos el preprocesado de los datos de cifar10: como hemos mencionado en la introducción, hemos optado por tener un conjunto de 10000 ejemplares para entrenamiento. Después veremos el ajuste de hiperparámetros, para culminar con nuestro modelo final.

2.1 Preparación de los datos

Empezamos tomando el conjunto de *training*:

```
library(tensorflow)
library(keras)
set.seed(12345)

dswsize = 10000
cifar10 <- dataset_cifar10()
dssize = 50000
mask = sample (1:dssize,dswsize)

cifar10$train$x = cifar10$train$x[mask,,,]
cifar10$train$y = cifar10$train$y[mask,]
```

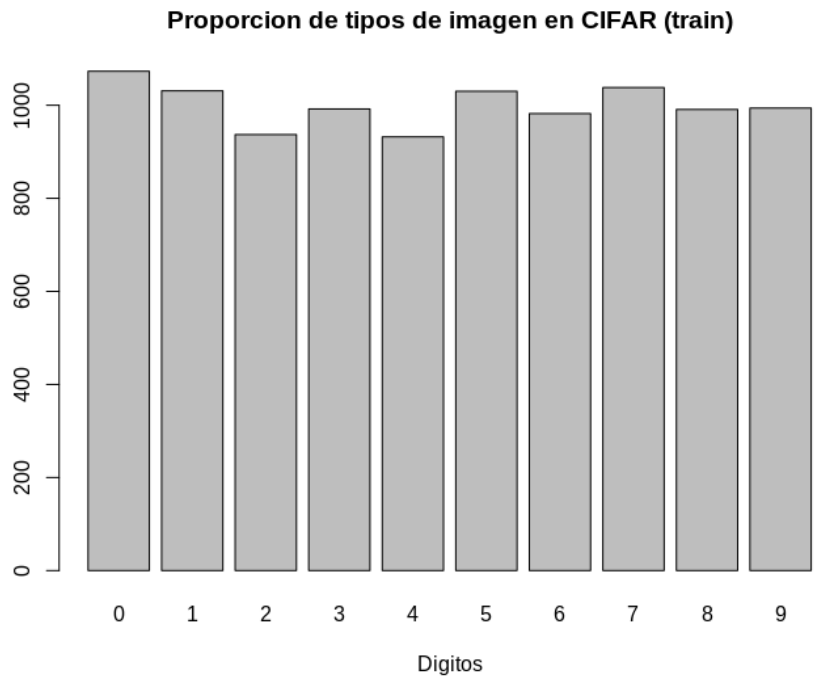
A continuación, convertimos los datos de cifar10 de la representación (muestra, fila_pixel, columna_pixel, color) a la representación (muestra, indice_pixel) con *array_reshape()*. Después normalizamos los datos, llevándolos al intervalo $[0, 1]$. Por último, usamos la codificación *one-hot* a la salida.

```
x_train <- cifar10$train$x
y_train <- cifar10$train$y

x_train <- array_reshape(x_train, c(nrow(x_train), 3072))
x_train <- x_train / 255
y_train <- to_categorical(y_train, 10)
```

Como se podía prever, podemos observar en los siguientes gráficos que los datos de train están muy balanceados, gracias al uso de *barplot*:

```
barplot(table(cifar10$train$y),main="Proporción de tipos de imágenes en CIFAR (train)",
                               xlab="Digits")
```



2.2 Mecanismo de validación cruzada

En nuestro proyecto usamos la función **createFolds** para dividir nuestro conjunto de entrenamiento en k pliegues. El método consiste en hacer un total de k iteraciones, asignando en cada una k-1 pliegues a los ejemplares de entrenamiento, y 1 pliegue a los de validación. De esta manera, conseguimos garantizar que los resultados son indepientes de la partición elegida. Podemos ver a continuación la estructura general que se seguirá de aquí en adelante.

```
myfolds = createFolds(y=cifar10$train$y,k=5)
k=5
for(i in 1:k){
    (...)
    model %>%
    (...)

    model %>% compile
    (...)

    history.It[[i]] =
              model %>% fit(
              x_train[-myfolds[[i]],],
              y_train[-myfolds[[i]],],
              (...)
              validation_data = list(x_train[myfolds[[i]],],
                                     y_train[myfolds[[i]],]),
              verbose = 2)
}
```

2.3 Ajuste de hiperparámetros

El proceso seguido a la hora de elegir el mejor modelo ha sido el siguiente:

- En primer lugar, empezamos considerando múltiples pruebas, con una y dos capas, en busca de una que destaque sobre las demás. En este paso, todavía no lidiamos con el *overfitting*.
- Una vez vistos los resultados, hemos elegido los tres mejores y les hemos aplicado el mecanismo de regularización *dropout*, para eliminar el sobreentrenamiento. De entre ellos, vemos cuál nos da el mejor *accuracy*.
- El resultado obtenido, superaba el 40% de accuracy. Sin embargo, creímos conveniente probar añadiendo una capa a este mejor modelo.
- Una vez obtenidos los resultados de los modelos con tres capas, el que daba mejor accuracy era uno de ellos. Por tanto el modelo final elegido es ese.

Veamos ahora en detalle cada uno de los pasos anteriores. Por un lado, la creación de nuestro primer modelo con Keras, el cual tiene una sola capa oculta de 64 nodos, se hace de la siguiente forma:

```
model = keras_model_sequential()
          model %>%
          layer_dense(units = unitsHidden, activation = 'relu', input_shape = c(3072)) %>%
          layer_dense(units = 10) %>%
          layer_activation('softmax')


          model %>% compile(
```

```
          loss = 'categorical_crossentropy',
          optimizer = optimizer_rmsprop(),
          metrics = c('accuracy'))

          history.It[[i]] =
            model %>% fit(
          x_train[-myfolds[[i]],],
          y_train[-myfolds[[i]],],
          epochs = 100,
          batch_size = 128,
          validation_data = list(x_train[myfolds[[i]],],
                                 y_train[myfolds[[i]],]),
          verbose = 2)
```

Veamos la descripción de este modelo:

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                196672
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 197,322
Trainable params: 197,322
Non-trainable params: 0
```

El número total de parámetros se obtiene fácilmente haciendo el siguiente cálculo:

$$totalPesos = nOculta * 3072 + nOculta + nSalida * nOculta + nSalida$$

Como podemos ver, este es un ejemplo muy sencillo. Cuando lleguemos al modelo final, con muchos más parámetros, estudiaremos también el resultado para explicar de dónde proviene la suma. Veamos primero los experimtentos realizados con una sola capa. Hemos decidido probar con 64, 128, 256, 512 y 1024 nodos, siendo el penúltimo de ellos el mejor, con un accuracy medio de 0.419.

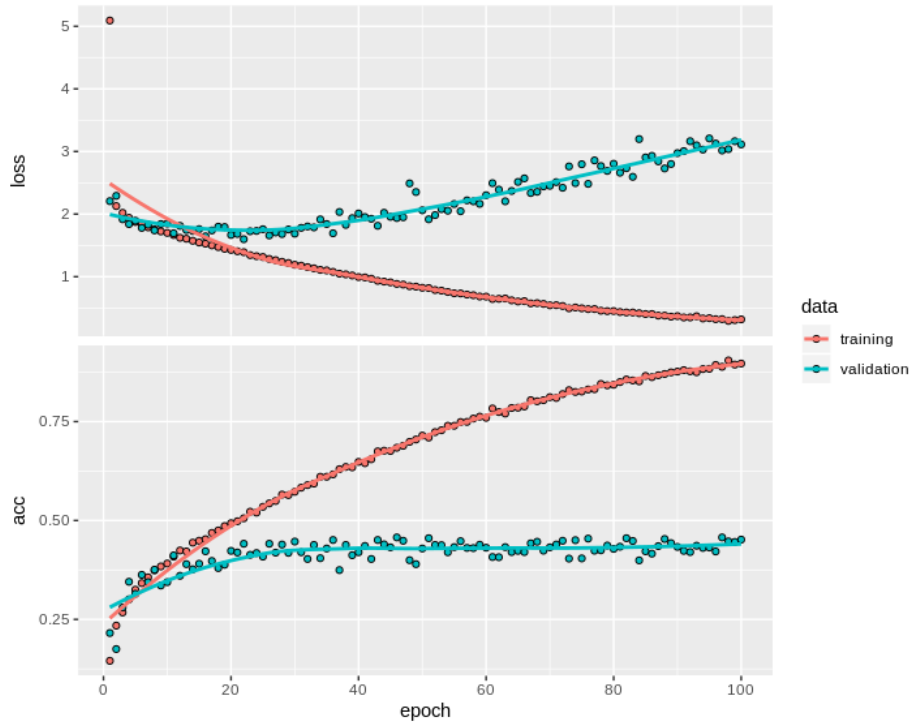
	Nodos capa oculta	val_acc medio
1	64	0.373
2	128	0.392
3	256	0.395
4	512	0.419
5	1024	0.4

Por otro lado, los resultados obtenidos para los experimentos con dos capas son:

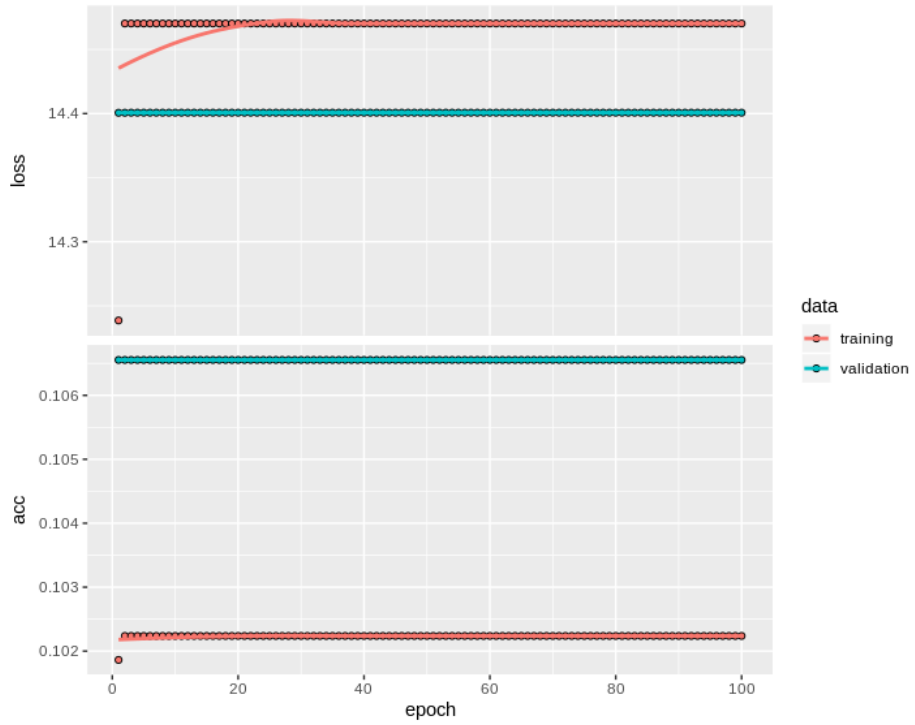
	Nodos primera capa oculta	Nodos segunda capa oculta	val_acc medio
1	256	256	0.416
2	256	512	0.406
3	256	1024	0.396
4	512	256	0.409
5	512	512	0.411
6	512	1024	0.421

7	1024	256	0.419
8	1024	512	0.359
9	1024	1024	0.29

Nos queda ahora observar estos resultados, y en función de ellos, decidir cuáles son los que elegiremos para el siguiente paso. Lo primero es darnos cuenta de que en todos los experimentos anteriores obtenemos overfitting, especialmente acentuado cuando trabajamos con dos capas. Veamos en la siguiente gráfica el loss y el accuracy de training y validación para el primer fold del experimento que tiene 512 nodos en la primera capa y 256 en la segunda.



Como hemos dicho, hay un gran sobreentramiento, el error de validación crece mientras que el de entrenamiento disminuye rápidamente. A la hora de elegir, además de mirar los valores de las tablas anteriores, nos percatamos de que en los dos últimos casos (1024-512 y 1024-1024), obteníamos en algunos folds resultados muy buenos, mientras que en otros, tanto el accuracy como el loss se mantenían constantes, como se puede observar en la siguiente imagen. Esto se atribuye al gran *overfitting* que se genera en este caso, ya que tenemos un gran número de nodos en ambas capas.



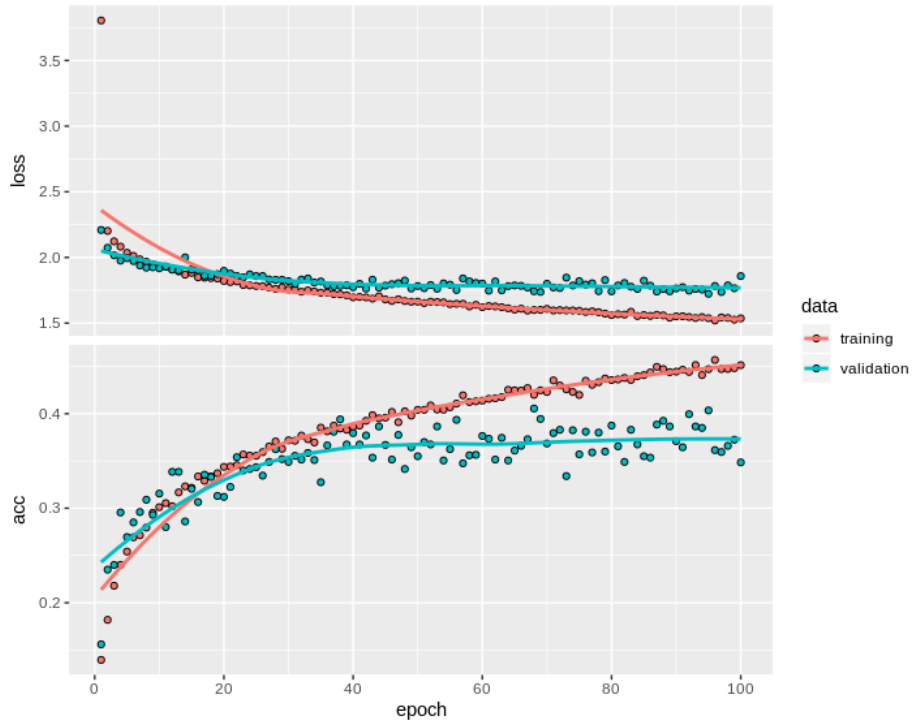
Tras ver el comportamiento que obtenemos en cada uno de los casos, los modelos que hemos elegido para eliminar *overfitting* y examinarlos en más detalle son:

- Modelo con dos capas compuesto de 512 nodos en la primera y 1024 en la segunda, ya que tiene un *accuracy* de 0.421, mayor que todos los demás.
- Modelo de dos capas compuesto de 1024 nodos en la primera y 512 en la segunda. Este es uno de los modelos que tenía tanto overfitting, que en algunos fold se mantenía constante, pero en otros daba buenos resultados, el cual sospechamos que puede dar buenos resultados.
- Modelo de dos capas compuesto de 1024 nodos en la primera y 1024 en la segunda. Este es el otro modelo que tenía mucho overfitting, luego también lo elegimos para examinarlo más en profundidad.

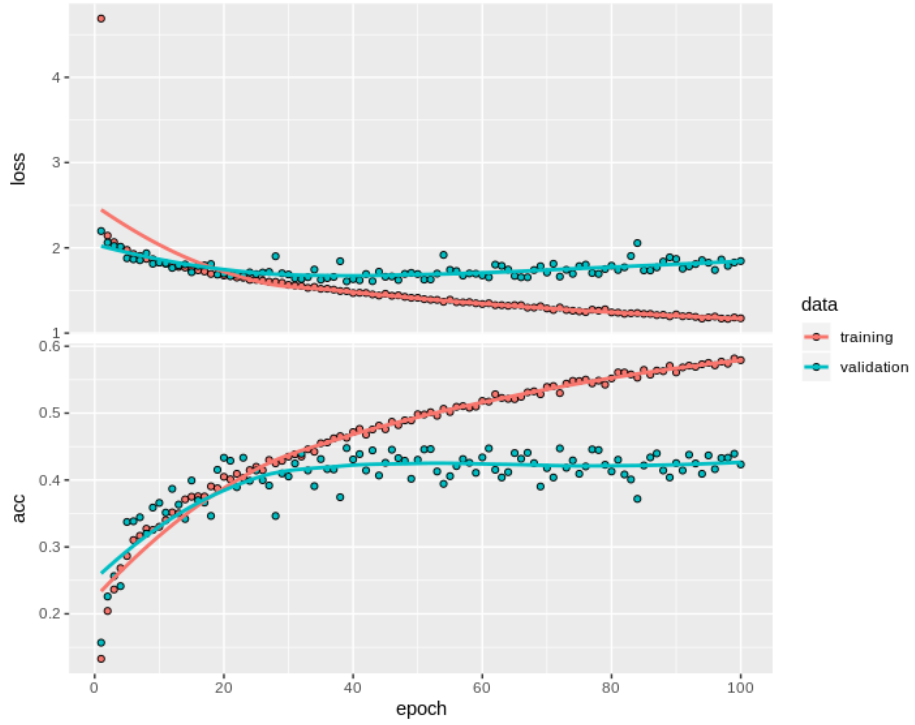
Una vez obtenemos esos tres modelos, hemos decidido usar la técnica de regularización *dropout* para eliminar el *overfitting* que tenían. Hemos probado dos opciones: añadir una capa de *dropout* de 0.3 tras ambas capas ocultas, o añadir una de 0.5 tras ambas.

	Nodos primera capa oculta	Nodos segunda capa oculta	Dropout	val_acc medio
1	512	1024	0.3	0.413
2	512	1024	0.5	0.355
3	1024	512	0.3	0.426
4	1024	512	0.5	0.38
5	1024	1024	0.3	0.345
6	1024	1024	0.5	0.375

En los casos en los que hemos añadido una capa de *dropout* de 0.5, hemos conseguido arreglar el overfitting, pero como el rate es tan grande estamos evitando que el modelo aprenda convenientemente obteniendo un *accuracy* pobre. Podemos ver en el siguiente gráfico el loss y el accuracy correspondiente al experimento que tiene 1024 nodos en la primera capa, 512 en la segunda, y un *dropout* de 0.5, para visualizar que el sobreentrenamiento ha sido eliminado.



Sin embargo, en los casos en los que hemos puesto un *rate* de 0.3, obtenemos mejores resultados, a excepción del último, en el que vuelve a haber tanto overfitting en uno de los folds, que el accuracy se mantiene constante a un valor muy bajo y la media disminuye. El que mejor *accuracy* nos da es el que tiene 1024 nodos en la primera capa y 512 en la segunda. Sin embargo, podemos observar que sigue teniendo un poco de *overfitting*:



Con lo cual, hemos decidido elegir este modelo, pero con un dropout de 0.4, el cual ha eliminado el sobreentrenamiento, y nos ha dado un *accuracy* final de 0.411.

Como no nos quedamos satisfechos con este valor, hemos probado a añadirle una tercera capa oculta a estas dos, lo que nos ha dado mejores resultados, como podemos contemplar en la siguiente tabla:

	Nodos 1ª capa	Nodos 2ª capa	Nodos 3ª capa	Dropout	val_acc medio
1	1024	512	128	0.3	0.435
2	1024	512	256	0.3	0.425
3	1024	512	512	0.3	0.422

El dropout usado en cada una de las capas ha sido 0.3. El mejor de ellos se corresponde con: 1024 nodos en la primera capa, 512 en la segunda y 128 en la tercera. Este será nuestro modelo final.

2.4 Evaluación del rendimiento futuro

En resumen nuestro modelo final escogido es el siguiente:

```
unitsHidden = 1024
unitsHidden2 = 512
unitsHidden3 = 128
dout = 0.3

model = keras_model_sequential()
model %>%
  layer_dense(units = unitsHidden, activation = 'relu', input_shape = c(3072)) %>%
  layer_dense(units = unitsHidden2, activation = 'relu') %>%
  layer_dropout(rate=dout) %>%
  layer_dense(units = unitsHidden3, activation = 'relu') %>%
  layer_dropout(rate=dout) %>%
  layer_dense(units = 10) %>%
  layer_activation('softmax')
```

Como hemos mencionado anteriormente, vamos a calcular también en este caso el número total de parámetros de nuestro modelo. Este se obtiene haciendo el siguiente cálculo:

$$totalPesos = uH * 3072 + uH + uH * uH2 + uH2 + uH3 * uH2 + uH3 + uH3 * 10 + 10$$

Por tanto tenemos un total de 3738506 parámetros que entrenar, como se puede observar ejecutando el comando *summary* sobre el modelo:

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 1024)              3146752
_________________________________________________________________
dense_2 (Dense)              (None, 512)               524800
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense               (None, 128)               65664
_________________________________________________________________
dropout_2                    (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
```

```
Total params: 3,738,506
Trainable params: 3,738,506
Non-trainable params: 0
```

Una vez fijados los hiperparámetros, se procede a entrenar en modelo con todo el conjunto de entrenamiento. Hemos optado por utilizar todos los ejemplares de cifar10.

```
cifar10 <- dataset_cifar10()
x_train <- cifar10$train$x
y_train <- cifar10$train$y
x_test <- cifar10$test$x
y_test <- cifar10$test$y
```

Una vez entrenado el modelo, es hora de emplear el test:

```
score <- model %>% evaluate(
  x_test, y_test,
  verbose = 0
)
```

Obtenemos los siguientes resultados:

```
> history$metrics$acc[100]
[1] 0.52604
> score
$`loss`
[1] 1.578348

$acc
[1] 0.496
```

En el epoch 100 obtenemos un accuracy de 0.52604, y en el test obtenemos un accuracy de 0.496. Lo cual no es mucho pero al menos llegamos al mínimo. Vamos a ver si podemos mejorar algo utlizando convolución.

3 Red de convolución

Ahora vamos a pasar a las redes de convolución. Seguiremos un enfoque parecido al que llevamos en MLP; primero empezaremos con la preparación de los datos, después veremos el ajuste de hiperparámetros, culminando con nuestro modelo final.

3.1 Preparación de los datos

Hacemos exactamente lo mismo que en MLP. Lo único que variamos son los parámetros del *array_reshape()*:

```
img_rows <- 32
img_cols <- 32
x_train <- array_reshape(x_train, c(nrow(x_train), img_rows,img_cols,3))
input_shape <- c(img_rows,img_cols,3)
```

En convolución vamos a construir filtros a partir de desplazar una ventana del filtro a lo largo y ancho de la imagen. De ahí que tengamos que aplicar el *array_reshape()* a cuatro dimensiones: la primera para indexar las imágenes y las otras tres para indexar cada píxel y canal (tres canales, RGB). Además en *input_shape* definimos la entrada con dos dimensiones para los píxels y, para cada píxel, tres valores que se corresponden con RGB.

3.2 Ajuste de hiperparámetros

Debido a la gran cantidad de hiperpámetros presentes nos hemos visto obligados a fijar los siguientes:

- Número de *epochs* a 60. Esto lo hemos creído conveniente pues las redes de convolución necesitan mucho tiempo para entrenarse y, en nuestro caso, el tiempo no es un recurso que nos sobre.
- Consideraremos una única capa oculta de MLP por la misma razón. Además, como posteriormente veremos, vamos a trabajar con 2 y 4 capas de convolución lo que dota de una complejidad significante a la red y, por consiguiente, creemos que el añadir más capas a la MLP posterior no nos aportará beneficios notables.

Así pues consideraremos, en principio, el siguiente modelo de convolución:

```
model %>%
  layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu',
                input_shape = input_shape) %>%
  layer_dropout(rate = 0.4) %>%
  layer_conv_2d(filters = filter2, kernel_size = c(ks1,ks1), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_flatten() %>%
  layer_dense(units = unitsHidden, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')
```

Como se puede observar, consta de dos capas de convolución seguidas de una capa de *pooling* y finalmente la parte de MLP con una capa oculta. Vamos a variar el número de filtros de cada capa de convolución, las dimensiones de los filtros y el número de nodos ocultos como sigue:

```
for(filter in c(16,32)){
  for(filter2 in c(32,64)){
    for(ks1 in c(3,4)){
      for(unitsHidden in c(128,256)){
      (...)
        myfolds = createFolds(y=cifar10$train$y,k=5)
        k=5
          for(i in 1:k){
          (...)
}
```

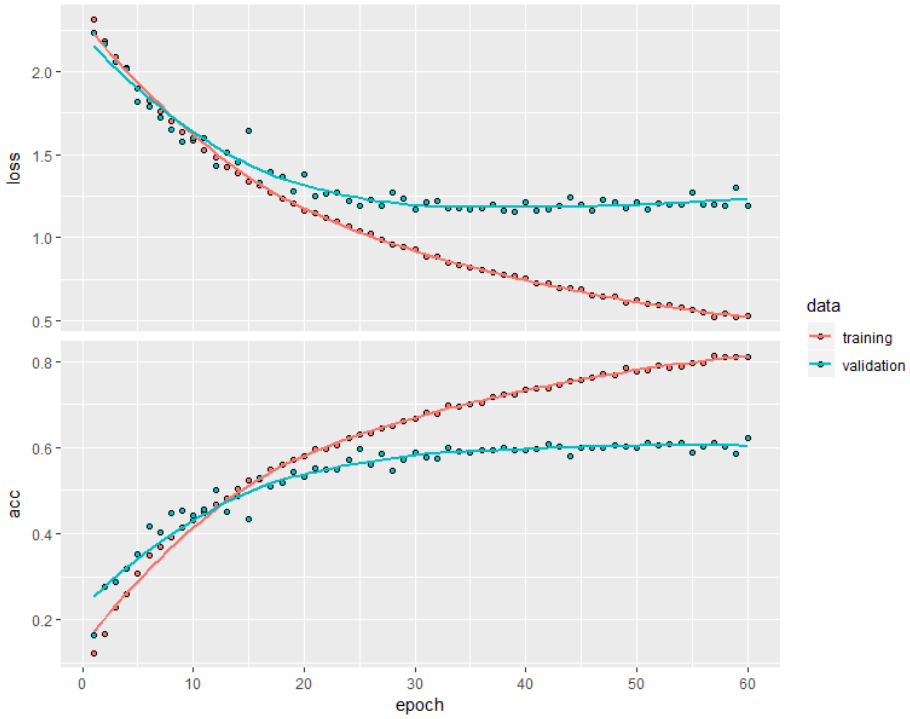
Obtenemos los siguientes resultados:

	Filtros	Nodos	Kernel Size	Accuracy medio
1	16-32	128	3	0.589
2	16-32	256	3	0.587
3	16-32	128	4	0.582
4	16-32	256	4	0.589
5	16-64	128	3	0.605
6	16-64	256	3	0.596
7	16-64	128	4	0.595
8	16-64	256	4	0.591
9	32-32	128	3	0.589
10	32-32	256	3	0.601
11	32-32	128	4	0.595
12	32-32	256	4	0.594

13	32-64	128	3	0.602
14	32-64	256	3	0.598
15	32-64	128	4	0.595
16	32-64	256	4	0.601

Como se puede ver, el mejor resultado se obtiene en:

	Filtros	Nodos	Kernel Size	Accuracy medio
5	16-64	128	3	0.605



Tiene un poco de *overfitting*. En vez de intentar solucionarlo (aumentando el *dropout*, como antes en MLP), hemos optado por subir el número de filtros a 4 y hacer otro tune para ver si podemos mejorar el *accuracy* medio. Consideramos pues, el siguiente modelo:

```
model %>%
 layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu',
               input_shape = input_shape) %>%
 layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu') %>%
 layer_max_pooling_2d(pool_size = c(2, 2)) %>%
 layer_dropout(rate = 0.35) %>%

 layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2), activation = 'relu') %>%
 layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2), activation = 'relu') %>%
             layer_max_pooling_2d(pool_size = c(2, 2)) %>%
 layer_dropout(rate = 0.35) %>%

 layer_flatten() %>%
 layer_dense(units = unitsHidden, activation = 'relu') %>%
 layer_dropout(rate = 0.5) %>%
 layer_dense(units = 10, activation = 'softmax')
```

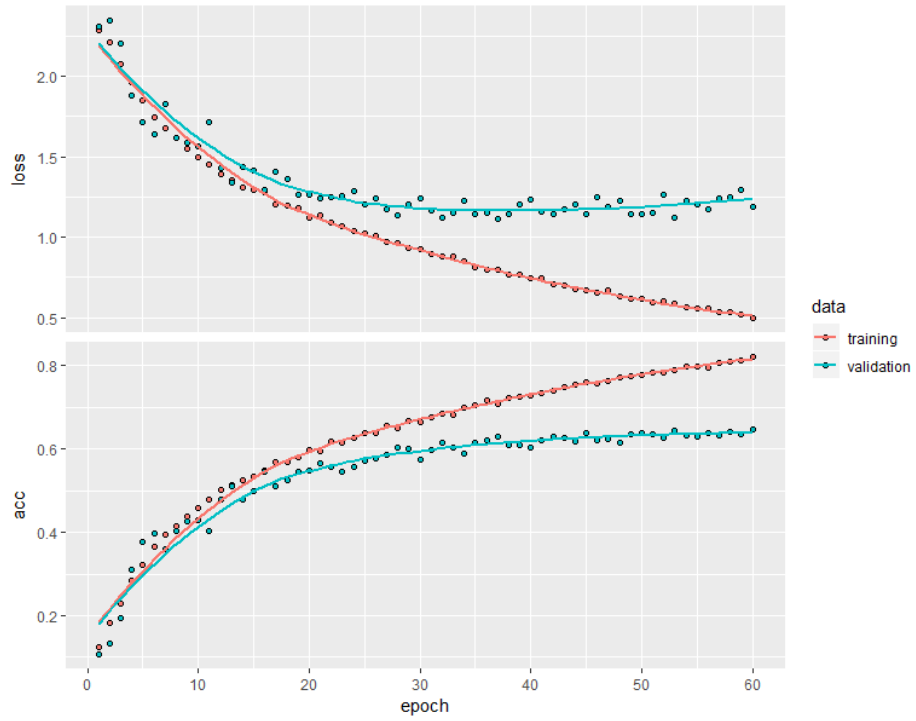
La dos primeras y las dos segundas capas de convolución van a tener las mismas características (filtros y *kernels*), las capas de *pooling* se mantienen fijas y el *dropout* se baja salvo en MLP que se mantiene en 0.5. Además, vamos a fijar los nodos ocultos de la capa MLP a 128, pues es la que mejores resultados nos ha dado en general con dos capas de convolución. Realizamos el siguiente bucle:

```
for(filter in c(32)){
  for(filter2 in c(64)){
    for(ks1 in c(3,4)){
      for(ks2 in c(3,4)){
        for(unitsHidden in c(128)){
        (...)
}
```

Vamos a hacer una primera toma de contacto con esta arquitectura; para ello, empezaremos fijando el numero de filtros de las capas de convolución a 32 y 64 respectivamente. Variaremos el *kernel size* de cada par de filtros. Obtenemos los siguientes resultados:

	Filtros	Nodos	Kernel Size	Accuracy medio
1	32-32-64-64	128	3-3	0.639
2	32-32-64-64	128	3-4	0.646
3	32-32-64-64	128	4-3	0.619
4	32-32-64-64	128	4-4	0.613

El mejor accuracy medio se obtiene para un *kernel size* de 3-4, es decir, los dos filtros de las primeras dos capas tendrán tamaño 3 mientras que los filtros de las últimas tendrán tamaño 4. A continuación podemos observar las curvas de *loss* y *accuracy* de entrenamiento y validación para este último caso:



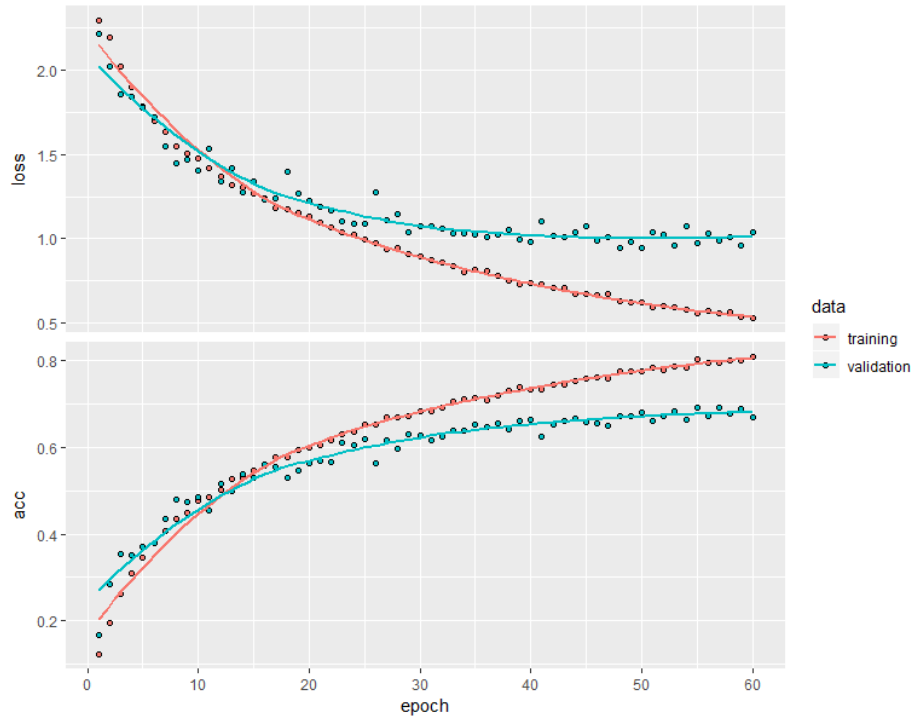
Una vez fijado el *kernel size*, pasaríamos a aumentar el número de filtros y, por consiguiente, la complejidad de la red. Como la facilidad para sobreentrenar suele tener una dependencia positiva con la complejidad del modelo, a raíz de la última gráfica (donde se aprecia un ligero *overfitting*) hemos optado por aumentar el rate de *dropout*, añadir una capa más de *dropout* en convolución y estudiar el comportamiento del modelo para los siguientes hiperparámetros:

```
for(filter in c(48)){
  for(filter2 in c(96)){
    for(ks1 in c(3)){
      for(ks2 in c(4)){
      (...)
         model %>%
           layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu',
                         input_shape = input_shape) %>%
           layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1),
                         activation = 'relu') %>%
           layer_max_pooling_2d(pool_size = c(2, 2)) %>%
           layer_dropout(rate = 0.4) %>%

           layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2),
                         activation = 'relu') %>%
           layer_dropout(rate = 0.4) %>%
           layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2),
                         activation = 'relu') %>%
           layer_max_pooling_2d(pool_size = c(2, 2)) %>%
           layer_dropout(rate = 0.4) %>%

           layer_flatten() %>%
           layer_dense(units = unitsHidden, activation = 'relu') %>%
           layer_dropout(rate = 0.5) %>%
           layer_dense(units = 10, activation = 'softmax')

           (...)
```



Como se puede observar no hay *overfitting*. Esto es lógico, pues hemos sido muy agresivos con el dropout. Aunque este último modelo tiene un *accuracy* mejor que los anteriores (0.653), hemos decidido ir un poco

más allá aumentado el número de filtros como sigue:

```
for(filter in c(64)){
  for(filter2 in c(96,128)){
    for(ks1 in c(3)){
      for(ks2 in c(4)){
      (...)
       model %>%
         layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu',
                       input_shape = input_shape) %>%
         layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1),
                       activation = 'relu') %>%
         layer_max_pooling_2d(pool_size = c(2, 2)) %>%
         layer_dropout(rate = 0.5) %>%

         layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2),
                       activation = 'relu') %>%
         layer_dropout(rate = 0.5) %>%
         layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2),
                       activation = 'relu') %>%
         layer_max_pooling_2d(pool_size = c(2, 2)) %>%
         layer_dropout(rate = 0.5) %>%

         layer_flatten() %>%
         layer_dense(units = unitsHidden, activation = 'relu') %>%
         layer_dropout(rate = 0.5) %>%
         layer_dense(units = 10, activation = 'softmax')

         (...)
```

Tras una larga espera, los resultados son los siguientes:

```
|   | Filtros       | Nodos | Kernel Size | Accuracy medio |
|---|---------------|-------|-------------|----------------|
| 1 | 64-64-96-96   | 128   | 3-4         | 0.668          |
| 2 | 64-64-128-128 | 128   | 3-4         | 0.669          |
```

Como la diferencia entre ambos accuracy medio es minúscula, optamos por el que tiene menos filtros (más simple), es decir, el primero. En suma el modelo final escogido sería:

```
filter <- 64
filter2 <- 96
ks1 <- 3
ks2 <- 4


model %>%
  layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu',
                input_shape = input_shape) %>%
  layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.5) %>%

  layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2), activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2), activation = 'relu') %>%
```

```
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.5) %>%

  layer_flatten() %>%
  layer_dense(units = unitsHidden, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')
```

Ahora solo resta entrenar el modelo con todos los datos de entrenamiento y ver su comportamiento en test.

3.3 Evaluación del rendimiento futuro

En resumen, nuestro modelo escogido es el siguiente:

```
filter = 64
filter2 = 96
ks1 = 3
ks2 = 4
unitsHidden = 128
epochs = 60
batch_size = 128
drate = 0.5

model %>%
  layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu',
                input_shape = input_shape) %>%
  layer_conv_2d(filters = filter, kernel_size = c(ks1,ks1), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = drate) %>%

  layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2), activation = 'relu') %>%
  layer_dropout(rate = drate) %>%
  layer_conv_2d(filters = filter2, kernel_size = c(ks2,ks2), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = drate) %>%

  layer_flatten() %>%
  layer_dense(units = unitsHidden, activation = 'relu') %>%
  layer_dropout(rate = drate) %>%
  layer_dense(units = drate, activation = 'softmax')
```

El número total de parámetros se obtiene fácilmente haciendo el siguiente cálculo:

$$totalPesos1Capa = 9 * 3 * 64 + 64$$

$$totalPesos2Capa = 64 * 9 * 64 + 64$$

$$totalPesos3Capa = 64 * 16 * 96 + 96$$

$$totalPesos4Capa = 96 * 16 * 96 + 96$$

$$totalPesosMlp1 = 96 * 16 * 128 + 128$$

$$totalPesosNodosSalida = 128 * 10 + 10$$

Por tanto tenemos un total de 482698 parámetros que entrenar, lo que se puede confirmar con el siguiente *summary*:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 30, 30, 64)        1792
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 28, 28, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2D)  (None, 14, 14, 64)     0
_________________________________________________________________
dropout_7 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 11, 11, 96)        98400
_________________________________________________________________
dropout_8 (Dropout)          (None, 11, 11, 96)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 8, 8, 96)          147552
_________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 4, 4, 96)       0
_________________________________________________________________
dropout_9 (Dropout)          (None, 4, 4, 96)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 1536)              0
_________________________________________________________________
dense_7 (Dense)              (None, 128)               196736
_________________________________________________________________
dropout_10 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_8 (Dense)              (None, 10)                1290
=================================================================
Total params: 482,698
Trainable params: 482,698
Non-trainable params: 0
```

Una vez fijados los hiperparámetros, se procede a entrenar en modelo con todo el conjunto de entrenamiento. Hemos optado por utilizar todos los ejemplares de cifar10.

```
cifar10 <- dataset_cifar10()
x_train <- cifar10$train$x
y_train <- cifar10$train$y
x_test <- cifar10$test$x
y_test <- cifar10$test$y
```

Una vez entrenado el modelo, es hora de emplear el test:

```
score <- model %>% evaluate(
  x_test, y_test,
  verbose = 0
)
```

Obtenemos los siguientes resultados:

```
> historyConv$metrics$acc[60]
[1] 0.78232
> score
$`loss`
```

```
[1] 0.6381743

$acc
[1] 0.8058
```

Obtenemos un accuracy de 0.78232 en el último epoch, mientras que en test obtenemos un 0.8058, lo cual no está nada mal.

4 Conclusiones

Atendiendo a los dos objetivos principales del proyecto (Convolución vs MLP, regularización), dejamos constancia de lo siguiente:

- El *accuracy* en el test que hemos obtenido utilizando convolución supera en gran medida al obtenido en MLP, lo cual refleja la superioridad de convolución sobre MLP. Por otro lado, debemos destacar que el tiempo empleado por la red de convolución para entrenar es también muy superior al de MLP (en los modelos con muchos filtros, convolución nos ha llegado a tardar 5 min/epoch); sin embargo, se podría decir que el precio a pagar es justo, pues llegamos a obtener un gran *accuracy*.
- El método de regularización que hemos empleado para evitar el *overfitting* ha sido el *dropout*. En MLP no hemos sido muy agresivos con el mismo; pero en convolución nos podía permitir ser agresivos con el *dropout* desde el principio ya que había gran cantidad de sobreajuste, esto podría ser debido a la complejidad de la red de convolución.