Práctica 3: Redes neuronales convolucionales

Exequiel Alberto Castro Rivero

78308513E

Subgrupo 2

Contenido

[Introducción 3](#_Toc91200979)

[Ejercicio 1 3](#_Toc91200980)

[Batch size = 32 6](#_Toc91200981)

[Batch size = 64 7](#_Toc91200982)

[Batch size = 128 8](#_Toc91200983)

[Batch size = 256 9](#_Toc91200984)

[Comparación y análisis 10](#_Toc91200985)

[Ejercicio 2 11](#_Toc91200986)

[Mejora 1: Normalización de los datos de entrada 11](#_Toc91200987)

[Mejora 2: Early Stopping 13](#_Toc91200988)

[Mejora 3 15](#_Toc91200989)

[Mejora 4 16](#_Toc91200990)

[Mejora 5 17](#_Toc91200991)

[Mejora 6 18](#_Toc91200992)

[Ejercicio 3 19](#_Toc91200993)

[Apartado A 19](#_Toc91200994)

[Apartado B 21](#_Toc91200995)

[Apartado C 22](#_Toc91200996)

# Introducción

El trabajo de esta práctica está relacionado con un tema de gran actualidad: las redes neuronales. En el caso que nos ocupa, vamos a realizar la implementación de **redes neuronales convolucionales**. Este tipo de redes han demostrado ser capaces de obtener resultados sorprendentes en determinadas áreas de la Visión por Computador.

El objetivo que se sigue con esta práctica es el de que nos familiaricemos con el diseño de redes neuronales convolucionales que aporta la librería Keras. De este modo, la experimentación va a ser un aspecto clave.

# Ejercicio 1

En este ejercicio se nos pide la construcción de un modelo base, *BaseNet*, para la resolución de un subproblema de la base de datos de *CIFAR100*. En concreto, esta base de datos representa un problema de clasificación con 100 clases. Sin embargo, para el caso que nos ocupa, nos quedaremos con los datos pertenecientes a sólo **25** de las 100 clases que forman la base de datos. Esta base de datos es cargada en memoria con la función de ayuda dada por los profesores *cargarImagenes*.

La descripción del modelo de *BaseNet* se muestra en la posterior tabla:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Layer No.** | **Layer Type** | **Kernel size (for conv layers)** | **Input|Output dimension** | **Input|Output channels (for conv layers)** |
| 1 | Conv2D | 5 | 32|28 | 3|6 |
| 2 | Relu | - | 28|28 | - |
| 3 | MaxPooling2D | 2 | 28|14 | - |
| 4 | Conv2D | 5 | 14|10 | 6|16 |
| 5 | Relu | - | 10|10 | - |
| 6 | MaxPooling2D | 2 | 10|5 | - |
| 7 | Linear | - | 400|50 | - |
| 8 | Relu | - | 50|50 | - |
| 9 | Linear | - | 50|25 | - |

Tabla 1: Descripción de BaseNet

La función que define dicha red convolucional es *get\_basenet*. Dicha función recibe los siguientes parámetros:

* *input\_shape*: El tamaño de las imágenes de entrada a la red (*[32, 32, 3]* en el caso de las imágenes de nuestro problema).
* *num\_classes*: El número de clases que se desea clasificar por la red neuronal, *25* en el caso de nuestro problema.

La definición de la red se ha hecho de forma secuencial, ya que esta forma es la que me parece más intuitiva y directa. Esta forma de definir redes tiene ciertas limitaciones, pero debido a que en esta práctica no se van a diseñar modelos muy complejos, me parece la más apropiada. A continuación se explica con mayor detalle cómo se ha realizado la definición del modelo con Keras:

* En primer lugar se especifica que el modelo se va a realizar de forma secuencial con *model = Sequential()*.
* Se añade una capa convolucional con activación *Relu* para implementar las capas número 1 y 2 de la tabla. Esto se realiza de la siguiente manera: *model.add(Conv2D(6, kernel\_size=(5, 5), activation='relu', input\_shape=input\_shape))*. Esta capa de convolución tiene 6 filtros debido a que así está especificado en los canales de salida de la tabla. Del mismo modo se obtienen los tamaños del kernel (*kernel\_size*). El parámetro *input\_shape* es necesario debido a que esta es la capa inicial del modelo y se necesitan saber las dimensiones de la entrada. El parámetro *activation* se usa para que la activación tras esta capa sea implementada con la función *Relu*. Todos los demás parámetros son los establecidos por defecto por Keras, bien porque los parámetros por defectos son los correctos (es el caso del parámetro *padding*, el cual tiene que estar establecido a *valid* para que se produzca una reducción en anchura y en altura de los bloques convolucionales) o bien porque no tienen un gran impacto en el rendimiento (se han realizado pruebas con distintos parámetros y se han obtenido resultados similares).
* Se añade una capa *MaxPooling* para reducir las dimensiones del bloque convolucional (el número de canales no se reduce). Esto se realiza de la siguiente manera: *model.add(MaxPooling2D(pool\_size=(2, 2)))*. Para que la reducción de dimensiones se haga a la mitad (tal y como se especifica en la tabla del modelo) el *pool\_size* debe tener valor 2. Los demás parámetros son los establecidos por defecto por Keras, ya que estos son los parámetros correctos (por ejemplo, *padding* establecido a *valid* para que se produzca la reducción a la mitad).
* Se añade otra capa convolucional del mismo modo que anteriormente: *model.add(Conv2D(16, kernel\_size=(5, 5), activation='relu'))*.
* Se añade otra capa *MaxPooling* del mismo modo que anteriormente: *model.add(MaxPooling2D(pool\_size=(2, 2)))*.
* Se añade una capa *Flatten* de modo que todos los píxeles del último bloque convolucional se disponen en un único array 1D. Una forma intuitiva de ver esta capa es que ya se ha decidido que la extracción de características de las imágenes ha llegado a su fin y, por lo tanto, todos los píxeles del último bloque convolucional son características que van a ser la entrada del clasificador de la red.
* Se añade una capa Densa para implementar las capas número 7 y 8 de la tabla. Esto se realiza de la siguiente manera: *model.add(Dense(50, activation='relu'))*. El número de neuronas se establece a 50 ya que así está especificado en la tabla. Lo mismo pasa con la función de activación. Los demás parámetros son los establecidos por defecto por Keras porque no tienen un gran impacto en el rendimiento (se han realizado pruebas con distintos parámetros y se han obtenido resultados similares).
* Se añade la capa totalmente conectada final, la cual se corresponde con la clasificación propiamente dicha. Esto se realiza de la siguiente forma: *model.add(Dense(num\_classes, activation='softmax'))*. El número de neuronas de esta capa totalmente conectada se debe de corresponder con el número de clases del problema que se está tratando. La función de activación usada es *softmax* debido a que el problema que se está resolviendo es un problema de clasificación multiclase: cada neurona “se encarga” de clasificar una clase y la salida de cada neurona es la probabilidad de que la imagen de entrada pertenezca a dicha clase.

Una aproximación intuitiva de lo que hace esta red es la siguiente: mediante el bloque formado por *Conv2D, Relu y MaxPooling* lo que se está realizando son transformaciones de la información de los bloques convolucionales anteriores (*Conv2D*) para luego realizar una selección de las características más relevantes (aquellas con mayor respuesta) con *Relu* y *MaxPooling*. Esta selección es importante, ya que si sólo se realizaran transformaciones de la información inicial el aprendizaje sería imposible. En la selección de aquellas características más relevantes es donde está la potencia de este tipo de aprendizaje (esto es similar a la forma de clasificar imágenes de los humanos: en una fotografía no todo es relevante a la hora de clasificarla). Una vez ya se ha completado la extracción de características (transformación + selección) dichas características son pasadas como entrada a un modelo de aprendizaje automático supervisado (una red neuronal en el caso de este modelo) con el fin de obtener un modelo de clasificación.

Como apunte adicional, si se desea ver con mayor profundidad el modelo definido o comprobar la correctitud del mismo se puede visualizar el resumen ofrecido por Keras (con la función *summary*) que se encuentra disponible en la ejecución del código.

Una vez terminada la etapa de definición del modelo se procede a la declaración del optimizador. En mi caso el optimizador usado será Adam. La justificación de la elección de este optimizador se encuentra en el paper [Adam: A Method for Stochastic Optimization](https://arxiv.org/pdf/1412.6980.pdf), paper donde se concluye que Adam es un buen optimizador para problemas con datasets grandes (el cual es nuestro caso). Además, en el mismo paper se menciona que los parámetros usados por este optimizador necesitan de poco ajuste (los parámetros por defecto obtienen un buen rendimiento en la mayoría de los casos). En mi caso, probé con distintas configuraciones de parámetros de este optimizador y en todas las pruebas se obtuvieron resultados similares o peores. Es por esta razón por la que me decanté usar los parámetros por defecto del optimizador.

Seguidamente se procede con la compilación del modelo. La función de pérdida usada es la entropía cruzada categórica: como ya se ha visto en la asignatura de aprendizaje automático, esta función de pérdida es la más popular a la hora de enfrentarse a problemas de clasificación multiclase. La métrica usada para evaluar el rendimiento es la *accuracy*, métrica popular en los problemas de clasificación. Esta métrica es definida como el porcentaje de imágenes bien clasificadas por el modelo. Como métrica se podría haber usado la misma función de pérdida establecida anteriormente, pero la entropía cruzada es muy poco intuitiva; por este motivo se usa la *accuracy* al ser bastante más intuitiva.

Tanto la definición del optimizador como la compilación del modelo se realizan con la función *model\_compile*, la cual recibe el modelo como parámetro. Dicha compilación se realiza con la función *compile* de Keras. Para la llamada sólo se especifican los parámetros *loss*, *optimizer* y *metrics*. Los demás parámetros son los establecidos por Keras ya que estos son los correctos (por ejemplo, al no tener desbalanceo de clases no hace falta que se use el parámetro *loss\_weights*).

En cuanto a la etapa de entrenamiento, el entrenamiento del modelo se realiza con la función *train\_model*. Esta función recibe por parámetros las imágenes de entrenamiento, las etiquetas de las mismas, el tamaño del batch a usar y el número de épocas. Para realizar el entrenamiento propiamente dicho se usa la función *fit* de Keras, con los parámetros anteriormente comentados. Los demás parámetros son los que establece por defecto Keras ya que, como en ocasiones anteriores, son los correctos.

En cuanto al número de épocas usado para los entrenamientos, se establece a 100. Esta decisión ha sido tomada en cuenta el compromiso precisión-coste computacional de este parámetro. Para un mayor número de épocas en general se consigue un mejor rendimiento (salvo en los casos en los que hay sobreajuste) pero el coste computacional es mayor. A menor número de épocas se consigue un peor rendimiento, pero se gana en coste computacional. Para los propósitos de esta práctica ese valor para las épocas me parece adecuado ya que permite observar la evolución de la función de pérdida y la métrica correctamente sin “gastar” demasiado tiempo.

El tamaño del batch usado para los entrenamientos es un parámetro difícil de fijar. El valor óptimo de este parámetro depende del problema que se está intentando resolver. En el paper [*Systematic evaluation of CNN advances on the Imagenet*](https://arxiv.org/pdf/1606.02228.pdf) los autores realizan un estudio experimental y concluyen que los tamaños de batch 64 y 128 son buenos valores en la mayoría de los casos. Para la elección de este parámetro hay que tener en cuenta el compromiso convergencia-coste computacional: a menor número de batch, la convergencia de los optimizadores es alcanzada en menor número de épocas, pero el coste computacional es mayor; sin embargo, a mayor número de batch la convergencia es alcanzada en un mayor número de épocas, pero el coste computacional es menor. Como se puede observar, hay cierta dependencia entre el tamaño del batch y el número de épocas de entrenamiento. Debido a todas las razones anteriormente comentadas, se van a probar distintos tamaños de batch y se va a analizar el comportamiento de las gráficas. Los resultados serán presentados posteriormente.

Finalmente, para el cálculo de la función de pérdida y la *accuracy* en test se usa la función *get\_loss\_accuracy*. Esta función recibe los datos de test y las etiquetas de dichos datos y haciendo uso de la función *evaluate*, se calcula y devuelve los valores de la función de pérdida y la *accuracy* en test.

Una vez explicadas todas las fases de construcción del modelo, se procede a realizar las pruebas anteriormente descritas. Se van a probar los siguientes tamaños de batch: 32, 64, 128 y 256. Cabe mencionar que, como se van a realizar distintos entrenamientos sobre el mismo modelo, los pesos aleatorios inicializados se guardan (con la función *get\_weights*) para poder restablecerlos en posteriores entrenamientos (con la función *set\_weights*).

## Batch size = 32

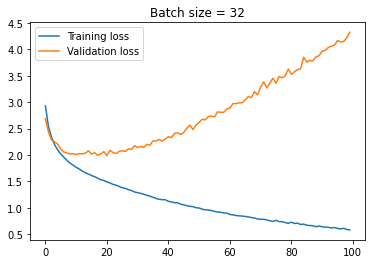


Ilustración 1: Evolución de la función de pérdida para tamaño de batch 32

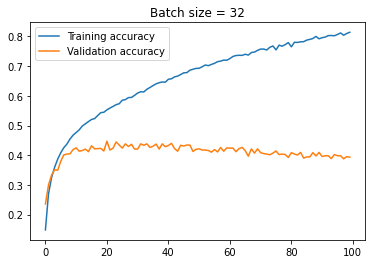


Ilustración 2: Evolución de la accuracy para tamaño de batch 32

Como se puede ver en las ilustraciones anteriores (*Ilustración 1* e *Ilustración 2*) la convergencia de la red se produce de manera bastante rápida (entorno a la época 20 ya se empieza a producir sobreajuste: la función de pérdida en validación empieza a crecer mientras que en los datos de entrenamiento sigue disminuyendo). A continuación, se presentan los resultados obtenidos en test y el tiempo que ha tardado la red en entrenar:

|  |  |  |
| --- | --- | --- |
| **Test loss** | **Accuracy test** | **Training time** |
| 4.16 | 38,88% | ~3 min 20s |

Tabla 2: Resultados en test y tiempo de entrenamiento para tamaño de batch 32

## Batch size = 64

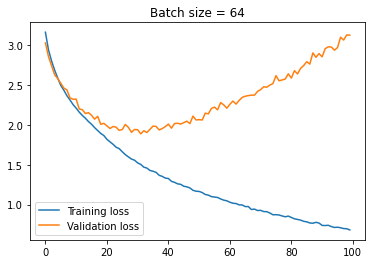


Ilustración 3: Evolución de la función de pérdida para tamaño de batch 64

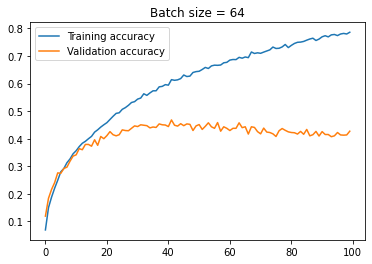


Ilustración 4: Evolución de la accuracy para tamaño de batch 64

Como se puede ver en las ilustraciones anteriores (*Ilustración 3* e *Ilustración 3*) la convergencia de la red se produce de manera rápida (entorno a la época 35 ya se empieza a producir sobreajuste: la función de pérdida en validación empieza a crecer mientras que en los datos de entrenamiento sigue disminuyendo). A continuación, se presentan los resultados obtenidos en test y el tiempo que ha tardado la red en entrenar:

|  |  |  |
| --- | --- | --- |
| **Test loss** | **Accuracy test** | **Training time** |
| 3.10 | 41,16% | ~2 min 10s |

Tabla 3: Resultados en test y tiempo de entrenamiento para tamaño de batch 64

## Batch size = 128

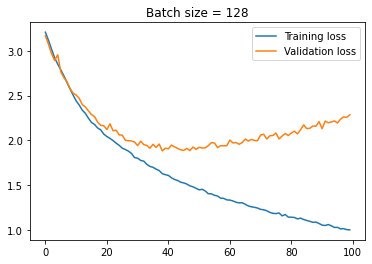


Ilustración 5: Evolución de la función de pérdida para tamaño de batch 128

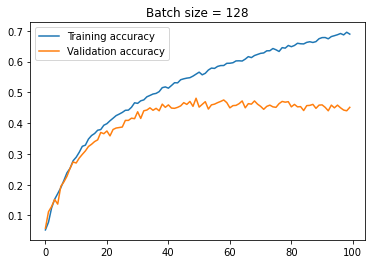


Ilustración 6: Evolución de la accuracy para tamaño de batch 128

Como se puede ver en las ilustraciones anteriores (*Ilustración 5* e *Ilustración 6*) la convergencia de la red se produce de manera lenta (entorno a la época 50 ya se empieza a producir sobreajuste: la función de pérdida en validación empieza a crecer mientras que en los datos de entrenamiento sigue disminuyendo). A continuación, se presentan los resultados obtenidos en test y el tiempo que ha tardado la red en entrenar:

|  |  |  |
| --- | --- | --- |
| **Test loss** | **Accuracy test** | **Training time** |
| 2.26 | 44,96% | ~1 min 30s |

Tabla 4: Resultados en test y tiempo de entrenamiento para tamaño de batch 128

## Batch size = 256

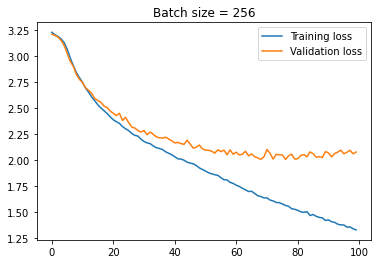


Ilustración 7: Evolución de la función de pérdida para tamaño de batch 256

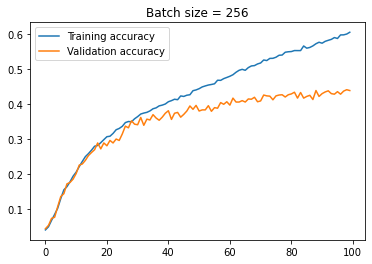


Ilustración 8: Evolución de la accuracy para tamaño de batch 256

Como se puede ver en las ilustraciones anteriores (*Ilustración 7* e *Ilustración 8*) la convergencia de la red no se produce de la misma manera que para los otros tamaños de batch (quizás si se entrenase unas épocas más se podría ver el mismo efecto). A continuación, se presentan los resultados obtenidos en test y el tiempo que ha tardado la red en entrenar:

|  |  |  |
| --- | --- | --- |
| **Test loss** | **Accuracy test** | **Training time** |
| 2.03 | 44,08% | ~50s |

Tabla 5: Resultados en test y tiempo de entrenamiento para tamaño de batch 256

## Comparación y análisis

Tal y como se puede observar en las ilustraciones del subapartado anterior, la forma de todas las gráficas es la misma: la diferencia entre ellas es un desplazamiento horizontal a la derecha (más notorio a medida que se aumenta el tamaño del batch). Esto pone de manifiesto que un tamaño de batch pequeño acelera la convergencia de la red. Esto casa con lo dicho anteriormente y con la teoría. También se puede observar el compromiso que hay entre tamaño de batch y coste computacional: a menor tamaño de batch mayor es el coste computacional. Así mismo también puede verse el compromiso entre el tamaño de batch y el número de épocas: a menor tamaño de batch el número de épocas necesarias para un buen entrenamiento también son menores.

A continuación, se presentan los resultados de test y el tiempo de entrenamiento obtenidos para todos los tamaños de batch:

|  |  |  |  |
| --- | --- | --- | --- |
| **Tamaño de batch** | **Test loss** | **Accuracy test** | **Training time** |
| **32** | 4.16 | 38,88% | ~3 min 20s |
| **64** | 3.10 | 41,16% | ~2 min 10s |
| **128** | 2.26 | 44,96% | ~1 min 30s |
| **256** | 2.03 | 44,08% | ~50s |

Tabla 6: Resultados obtenidos para los diferentes tamaños de batch

Como se puede observar, los valores de *accuracy* obtenidos están alrededor del 42-43% para el modelo *BaseNet*. Esto nos servirá como *baseline* para próximas mejoras. Así mismo, para dichas mejoras se tomarán los siguientes valores para el entrenamiento: número de épocas = 100 y tamaño de batch = 128 ya que con estos valores se consigue un equilibrio en el compromiso rendimiento – coste computacional.

# Ejercicio 2

En este ejercicio se nos plantea realizar mejoras sobre el modelo *BaseNet* para conseguir un mayor rendimiento en la base de datos *CIFAR-100*. A continuación se muestran las mejoras implementadas.

## Mejora 1: Normalización de los datos de entrada

La primera mejora planteada de las listadas en el documento de la práctica es la normalización de los datos de entrada. Es sabido que la normalización de los datos de entrada ayuda al entrenamiento ya que se consigue una convergencia más rápida. Al normalizar los datos lo que se está haciendo es que todas las imágenes de entrada se muevan en el mismo rango de valores para así poder evitar sesgos hacia determinadas imágenes. Si no se normalizan los datos de entradas, la actualización de los pesos usando el algoritmo de *backpropagation* puede ser bastante dispar dependiendo de la escala de las imágenes.

Para la realización de esta mejora se han implementado varias funciones. La primera de ellas ha sido la función *split\_train\_val*, que como su propio nombre indica, recibe un conjunto de datos y realiza la separación en datos de entrenamiento y datos de validación (un 10% de los datos de entrenamiento, tal y como se especifica en el enunciado de la práctica). Esta separación se realiza conservando el porcentaje de clases en ambos conjuntos de datos para que no haya clases infrarrepresentadas.

La siguiente función implementada fue *normalize\_images*. Esta función recibe los datos de entrenamiento y los de test, ambos con sus etiquetas, y el tamaño de batch. El funcionamiento de esta función sigue los siguientes pasos:

* Separación de los datos de entrenamiento iniciales en dos conjuntos: un nuevo conjunto de entrenamiento y un conjunto de validación (se usa la función anteriormente descrita, *split\_train\_val*).
* Se crea el objeto de la clase *ImageDataGenerator* de Keras con el que se va a realizar la normalización de los datos. Hay que establecer los parámetros *featurewise\_center* y *featurewise\_std\_normalization* a *True* para que se realice la normalización que se desea.
* Se obtienen los pesos con los que se va a realizar la normalización de los datos mediante **el conjunto de entrenamiento** (*datagen.fit(x\_train)*). Este aspecto es bastante importante puesto que el conjunto de test y validación tiene que ser normalizado con los mismos parámetros que ha sido normalizado el conjunto de entrenamiento. Si esto no fuera así, la red estaría aprendiendo a clasificar imágenes en una determinada escala y luego cuando tenga que evaluar las imágenes de test o validación, las cuales están en otra escala, los resultados serían pésimos.
* Se crean los iteradores para cada conjunto de datos (entrenamiento, validación y test) con la función *flow* del *datagen*, pasando el tamaño de batch especificado como parámetro a la función y se devuelven estos iteradores.

Finalmente se crearon las funciones *train\_model\_it* y *get\_loss\_accuracy\_it* que no son más que una extensión de las funciones explicadas en el ejercicio 1 para que puedan trabajar con iteradores de datos.

A continuación se presentan los resultados obtenidos tras esta mejora:

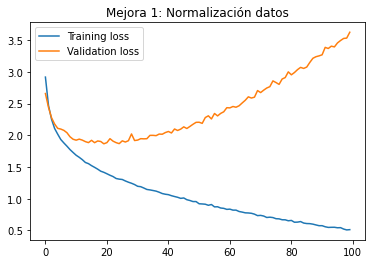


Ilustración 9: Evolución de la función de pérdida para la mejora 1

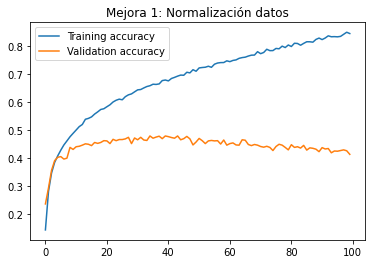


Ilustración 10: Evolución de la accuracy para la mejora 1

|  |  |
| --- | --- |
| **Test loss** | **Accuracy test** |
| 3.47 | 43,34% |

Tabla 7: Resultados de test obtenidos para la mejora 1

Si nos fijamos en los resultados obtenidos en test, se puede observar que son muy similares a los obtenidos en el ejercicio anterior. Esto era de esperar puesto que la normalización de los datos, en general, no suele aportar una mejora en el rendimiento de la red. Lo que sí se consigue es que la convergencia se alcance más rápidamente. Si se comparan estas dos últimas Ilustraciones con *Ilustración 5* e *Ilustración 6* (recuérdese que el tamaño de batch considerado para este ejercicio es de 128) se puede observar que las gráficas tienen la misma forma: la diferencia se encuentra en un desplazamiento horizontal. Esto casa con lo dicho anteriormente y con lo comentado en el ejercicio 1. Aunque la normalización de los datos de entrada no mejore el rendimiento sí que ayuda a la convergencia, por lo que considero que su impacto en la red es beneficioso.

## Mejora 2: Early Stopping

Tal y como se ha podido observar en las gráficas de la mejora anterior, está claro que la red realiza un sobreajuste (“memoriza” los datos de entrenamiento, conduciendo así a una baja capacidad de generalización y obteniendo muchos peores resultados en test que en training). Tal y como se indica en el enunciado de la práctica, una forma de reducir dicho sobreajuste es usando la técnica de *Early Stopping*. El funcionamiento de esta técnica se basa en parar el entrenamiento de la red cuando se detecta que la función de pérdida en el conjunto de validación aumenta en lugar de disminuir. Si se observa la gráfica de la evolución de la función de pérdida en la mejora anterior (*Ilustración 9*) es fácil ver que a partir de la época 20, la función de pérdida en el conjunto de validación aumenta mientras que la de entrenamiento sigue disminuyendo. Es en este punto en el que la red empieza a realizar sobreajuste. Con la técnica de *Early Stopping* lo que se pretende es detectar cuando pasa esto y cortar el entrenamiento en dicho punto.

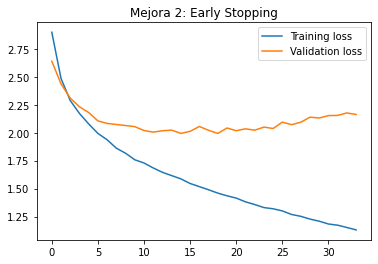
La dificultad del uso de esta técnica reside en fijar el número de épocas para el cual se va a comprobar el decrecimiento de la función de pérdida. Por ejemplo, si este parámetro se establece a pocas épocas puede darse el caso de que, por azar, haya un momento en el que la función de validación empiece a aumentar (pero, tras un determinado número de épocas sigue decreciendo) y se corte el entrenamiento. Por el caso contrario, si se establece a un número de épocas muy alto, los beneficios de la técnica se ven limitados. En nuestro caso, tras numerosas pruebas, he decidido establecer este parámetro a **15 épocas**, ya que con este valor he obtenido buenos resultados.

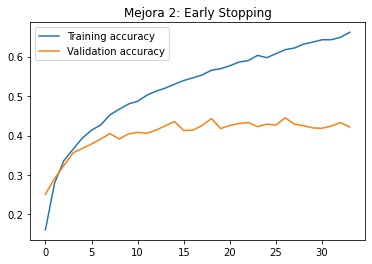
Cabe mencionar también que las mejoras que se van aplicando son incrementales, es decir, la mejora 2 también incorpora la mejora 1. Esto será así para todas las mejoras implementadas en este ejercicio.

La función implementada para esta mejora es *train\_model\_early*, una adaptación de la función encargada del entrenamiento de la red que se encarga de definir el *callback* necesario para implementar dicha técnica en el entrenamiento. Dicho *callback* se ha definido con los siguientes parámetros:

* *monitor*: *val\_loss*, es decir, la métrica que se comprueba para realizar *Early Stopping* es la función de pérdida en el conjunto de validación.
* *patience*: 15, tal y como se ha explicado antes con este valor se han obtenido buenos resultados.
* *restore\_best\_weights*: *True*, para que los pesos sean restablecidos a la época en la que se alcanzó el mínimo en la función de pérdida en el conjunto de validación. Esto es bastante útil ya que permite obtener los mejores pesos encontrados en todo el proceso de entrenamiento de la red.

Para esta mejora se obtuvieron los siguientes resultados:

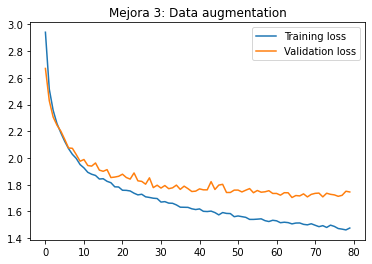


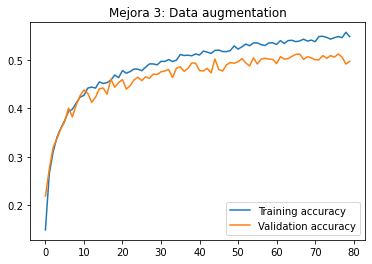


Test loss: 2.0249481201171875

Accuracy test: 0.44200000166893005

## Mejora 3

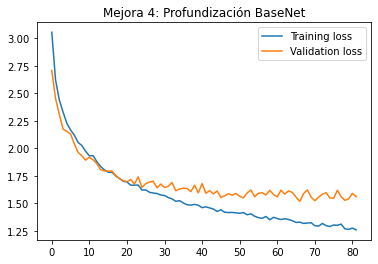


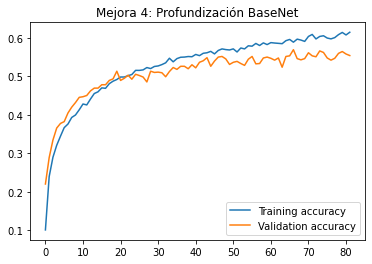


Test loss: 1.6242103576660156

Accuracy test: 0.5235999822616577

## Mejora 4

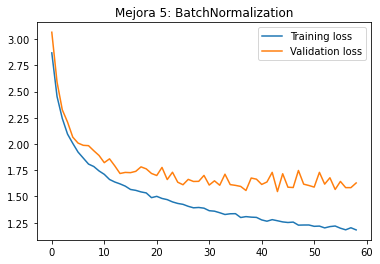


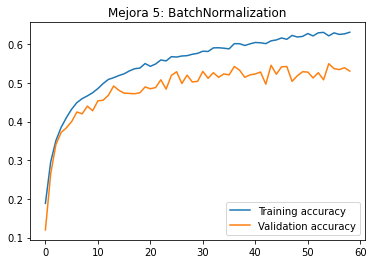


Test loss: 1.4938095808029175

Accuracy test: 0.5612000226974487

## Mejora 5

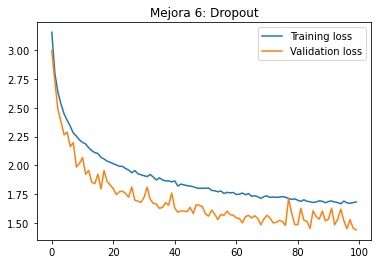


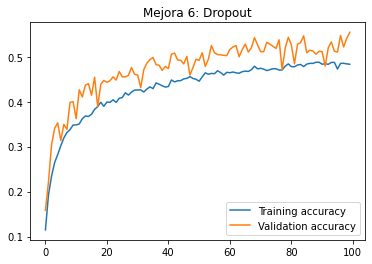


Test loss: 1.4857732057571411

Accuracy test: 0.5551999807357788

## Mejora 6



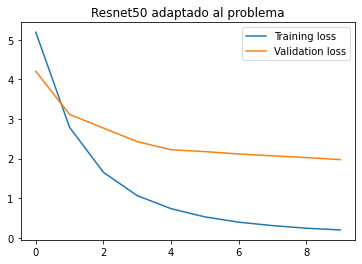


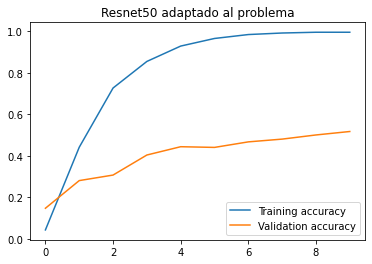
Test loss: 1.4237828254699707

Accuracy test: 0.5655999779701233

# Ejercicio 3

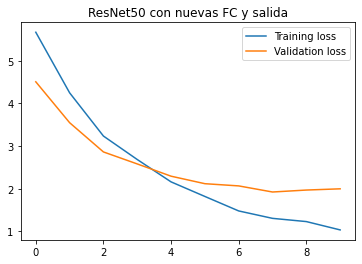
## Apartado A

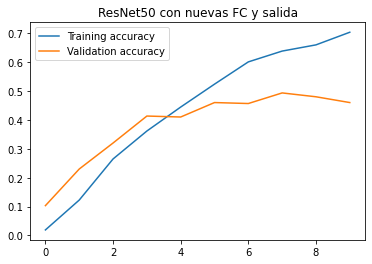




Test loss: 2.3560123443603516

Accuracy test: 0.4207055866718292

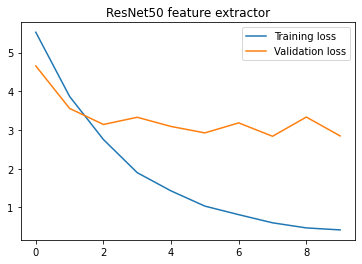


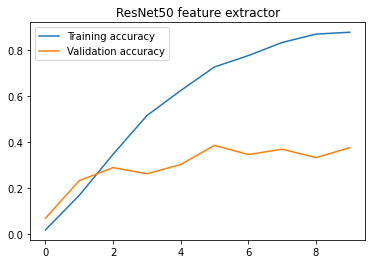


Test loss: 2.446516990661621

Accuracy test: 0.4048796594142914

## Apartado B

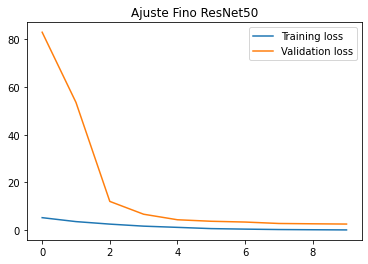


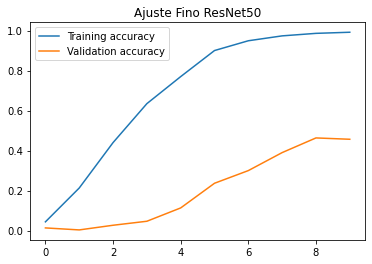


Test loss: 3.6546714305877686

Accuracy test: 0.3125618100166321

## Apartado C





Test loss: 2.8896753787994385

Accuracy test: 0.35146719217300415