Visión por Computador

grado en ingeniería informática

PRÁCTICA 3: REDES NEURONALES CONVOLUCIONALES

Realizado por:
Alba Casillas Rodríguez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2021-2022

Índice

IMPORTANTE

Antes de la correción de la práctica se debe de tener en cuenta las siguientes aclaraciones:

- Para la realización del segundo ejercicio de la práctica (Mejora del modelo BaseNet), se ha hecho un estudio incremental de las sucesivas mejoras. El entrenamiento de estos modelos incrementales **ha sido comentado en el código** debido al alto tiempo de ejecución. Esto quiere decir que **únicamente se ejecutará** el modelo resultante tras todas las mejoras, ergo, **solamente** aparecerán en la ejecución del código las **gráficas del modelo final**.
- Se ha decido separar la ejecución de los ejercicios 1 y 2 de la ejecución del código del ejercicio 3 para poder facilitar la ejecución de la práctica completa, ya que este último consume bastantes recursos. Por ende, los dos primeros ejercicios se encontrarán en un fichero llamado ***P3_1y2.py*** y el tercero se encontrará en ***P3_3.py***.
- El **bonus**, dado que tiene que ver con mejorar los resultados obtenidos de los dos primeros ejercicios, también se encontrará en el fichero ***P3_1y2.py***. Manteniendo solamente para ejecutar **solamente** aquella solución considerada final.

1. Apartado 1: BaseNet en CIFRAR100

Comenzamos creando un modelo base llamado BaseNet, que tras su entrenamiento y ejecución nos dará un porcentaje de clasificación de referencia para las posteriores mejoras. El modelo BaseNet consta de dos módulos convolucionales (conv-relu-maxpool) y dos capas lineales. La arquitectura precisa se define a continuación:

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	-

Figura 1: Arquitectura del modelo BaseNet.

1.1. Apartado 1.1

Familiarizarse con la arquitectura BaseNet ya proporcionada, el significado de los hiperparámetros y la función de cada capa. Crear el código para el modelo BaseNet.

Se comienza el desarrollo del trabajo usando el conjunto **CIFAR100**. Dicho *dataset* se compone de 60000 imágenes a color (tres canales) con una resolución de 32x32 píxeles y de 100 clases de imágenes distintas uniformemente distribuidas (600 imágenes por clase).

Para esta práctica, se decide utilizar un subconjunto de 15000 imágenes pertenecientes a 25 clases, de las cuales 12500 imágenes serán reservadas para *training* (con un 10 % de ellas para *validación*) y las 2500 restantes para *test*.

Trabajaremos definiendo inicialmente el **modelo BaseNet**, el cual se compone de dos módulos convolucionales (*Conv + ReLU + MaxPool*) y dos capas densamente conectadas (*linear layers* o *fully connected*), según especifica el propio enunciado de la práctica.

Por tanto, este primer apartado está dedicado a crear el código que refleje de manera fiel la arquitectura del modelo definido, del cual observaremos cada capa

con mayor detalle.

Como paso inicial, se establece la forma de la entrada de la primera capa de la red. Como las imágenes de CIFAR100 tienen un tamaño de 32x32 píxeles y 3 canales (RGB), el tamaño de la entrada será (32x32x3).

Se crea el modelo mediante:

model = Sequential()

forzando a que todas las capas de la red vayan una detrás de otra de forma secuencial, sin permitir saltos ni ciclos entre las capas. Las capas que lo forman son:

- **Convolución 2D[1]: 6 kernels 5x5**.

  Conv2D(filters, kernel_size, padding, input_shape)

  donde:

  - **filters** es el número de filtros de salida.
  - **kernel_size** es una tupla que especifica el alto y ancho de la ventana de convolución 2D.
  - **padding** es el padding aplicado.
  - **input_shape** es el tamaño de la imagen de salida.

  Consiste en la aplicación de una convolución con un kernel 5 x 5 x 3 (ya que se tiene en cuenta de que, en esta primera capa, se realiza la convolución para todos los canales de la entrada).

  Por cada convolución realizada, se obtiene un mapa de características. Por lo que al aplicar 6 filtros, obtendremos 6 canales en la salida.

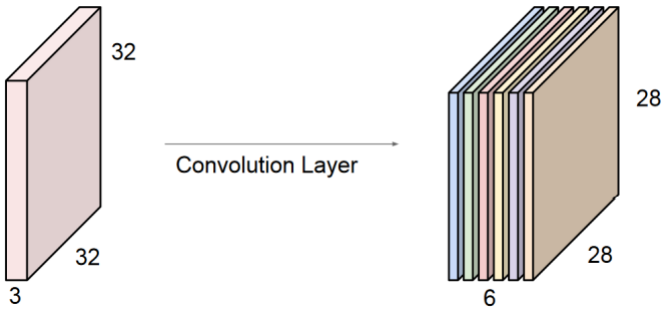


Figura 2: Comportamiento de la convolución aplicada (6 kernels 5x5).

Establecemos el parámetro de *padding* a *"valid"* para indicar que solo se tiene que aplicar la convolución donde se pueda ajustar el kernel. De esta forma, se debe de tener en cuenta que al no poder aplicar padding en los bordes, perderemos un poco de tamaño en la salida. Concretamente, si el kernel tiene tamaño 5, perdemos en cada lado la parte entera de la mitad del tamaño del kernel, es decir: $\lfloor \frac{5}{2} \rfloor = 2$. Por tanto, perdemos esta cantidad en los bordes de izquierda, derecha, arriba y abajo, pasando de un tamaño de 32 x 32 x 3 a **28 x 28 x 6**.

- **ReLU[2]:**

Activation('relu')

Tras la convolución, se aplica la ***función de activación*** mediante la capa *Activation* de **Keras**, cuyo único parámetro es el tipo de activación.

Para ello se hará uso de la función no lineal ***ReLU*** (*Rectified Linear Unit*):

$$\boldsymbol{ReLU(x) = max(0,x)}$$

Con el uso de esta capa se pretende *elegir*, de entre todos los valores obtenidos en el bloque de salida de la convolución, aquellos valores que resulten más relevantes. Es decir, la *no linealidad* de la operación permite eliminar valores poco interesantes.

Esta capa es esencial para el *proceso de aprendizaje* ya que de tomar únicamente decisiones lineales, el proceso solamente se limitaría a transformar y mezclar información, pero nunca a tomar decisiones sobre lo verdaderamente relevante.

- **MaxPooling 2D[3]:**

MaxPooling2D(pool_size)

donde:

- **pool_size** es el factor de reducción de la entrada que, como se indica en el enunciado, debe de ser 2 para reducir a la mitad.

Max Pooling es una técnica utilizada para reducir la dimensionalidad de una capa a lo largo de sus dimensiones espaciales (ancho y alto). Por tanto, el tamaño de salida es de **14 x 14 x 6**.

Cabe destacar que **Keras** permite añadir en su método el valor de otro hiperparámetro que modifica el tamaño del volumen salida: *stride*, aunque no se hará uso del mismo para que de esta forma tome el mismo valor que *pool_size*, es decir, 2.

- **Convolución 2D: 16 kernels 5x5**.

  Se vuelve a aplicar convoluciones con un kernel 5 x 5, con la diferencia de que el diseño especifica el uso de 16 filtros, obteniendo como consecuencia un bloque menos ancho y largo pero de mayor profundidad.

  De nuevo, perdemos dimensionalidad al no hacer uso de *padding*, por lo que el tamaño de salida de esta capa será de **10 x 10 x 16**.

- **ReLU:**

  Tras la convolución anterior, se aplica de nuevo a la salida la función de activación ReLu.

- **MaxPooling 2D:**

  De igual manera, aplicamos una reducción de la dimensionalidad del bloque a la mitad. Ahora el tamaño de salida es de **5 x 5 x 16**.

- **Linear[4]:**

  Se añade una capa densamente conectada. Para ello, se debe de realizar previamente un *aplanamiento*[5]:

  Flatten()

  Con esta capa, se convierte el bloque en un vector unidimensional. De manera que nuestro bloque 5 x 5 x 16 pasa a ser un vector de $5 \cdot 5 \cdot 16 =$ **400 elementos**.

  Tras aplanar la entrada, se aplica con la capa densamente conectada:

  Dense(units)

  donde:

  - **units** indica el número de neuronas de la capa. Este parámetro coincide con el tamaño de salida de esta capa.

- **ReLU:**

  Tras la capa densamente conectada, se aplica a la salida la función de activación ReLu.

- **Linear:**

  Como capa final, se aplicará otra capa densamente conectada. Esta vez el parámetro *units* tendrá valor 25, ya que finalmente se tendrá tantas neuronas como clases tenga nuestro problema (25).

  Además, al estar tratando con un problema multiclase, esta última capa deberá dar como salida la clasificación (etiqueta) de la imagen, por lo que se aplicará una activación ***softmax*** para transformar las salidas de las neuronas en la probabilidad de pertenecer a cada clase.

  Activation('softmax')

  La interpretación probabilística de este clasificador viene dada por la expresión

  $$P(y_i|x_i; W) = \frac{e^{fy_i}}{\sum_j^k e^{f_j}}$$

  haciendo referencia a la probabilidad (normalizada) asignada a la etiqueta correcta y_i dada la imagen x_i y parametrizada por W; siendo $f = f_1, f_2, ..., f_k$ las puntuaciones del vector de salida y k el número de clases de nuestro problema [6].

Una vez definido el modelo, se debe establecer algunos parámetros más antes de su compilación:

- **Número de épocas**: Determina el número de pasadas completas a través del conjunto de datos de entrenamiento. El número de épocas no es muy significativo aún, ya que no se ha visto los valores del error de validación y entrenamiento, que nos permitirá refinar este parámetro. Inicialmente, se establecerá este valor a 30 épocas.
- **Tamaño del batch**: Hace referencia al número de muestras que se procesan por cada actualización del modelo, es decir, la red neuronal ajustará sus pesos en base a procesar batches, donde cada batch es un conjunto de imágenes pertenecientes al conjunto de entrenamiento que se procesarán en cada una de las épocas.

  Se aconseja el uso de un tamaño de batch potencia de 2 para aprovechar mejor la arquitectura hardware de la GPU. Atendiendo a la literatura[8], normalmente no se utiliza un tamaño demasiado grande ya que en la práctica se ha observado que un aumento del tamaño produce una degradación significativa en la calidad del modelo, debido a un despliegue de su capacidad de generalización. Por tanto, se ha decidido usar para la práctica un tamaño de batch de 32, siendo uno de los tamaños más utilizados y prometedores a dar un resultado razonable.

- **Optimizador**: Como se ha visto a lo largo de esta asignatura, el optimizador a utilizar será ***SGD, Gradiente Descendente Estocástico***[7]. Los algoritmos de optimización iterativa con gradiente convergen en un óptimo local, es por eso que controlar la velocidad y dirección con la que nos vomemos por la superficie es muy importante. Sin embargo, se ha decidido para la práctica utilizar los valores por defecto de SGD, siendo *learning_rate = 0.01*, ya que resulta un valor razonable, y *momentum = 0* por considerar complicado encontrar a priori un valor adecuado para el mismo.

- **Función de pérdida**: Al estar tratando un problema de clasificación multiclase probabilística (haciendo uso del clasificador probabilístico softmax), se hace evidente el hecho de que la función de pérdida a utilizar será la ***cross-entropy*** o ***entropía cruzada***[6].

  La entropía cruzada entre una distribución 'verdadera' p y su distribución estimada q es definida como

  $$H(p, q) = -\sum_x p(x) \log q(x)$$

  Por tanto, el uso del clasificador softmax minimiza la entropía cruzada entre las probabilidades de clase estimadas ($q = e^{f_{y_i}} / \sum_j^k e^{f_j}$) y la distribución 'verdadera'.

- **Métrica**: La métrica utilizada en este problema será ***accuracy*** o ***precisión***, haciendo referencia a la proporción de aciertos sobre el número total de elementos.

Una vez aclarados todos los parámetros necesarios, se realiza la compilación del modelo:

compile(loss=keras.losses.categorical_crossentropy, optimizer=SGD(), metrics=['accuracy'])

Por último, antes de entrenar se debe calcular y guardar los pesos aleatorios con los que comienza la red. Para ello, se usa la función *get_weights()*. Este valor será útil para poder reestablecerlos posteriormente y reentrenar el modelo y para compararlos con futuros pesos al realizar mejoras.

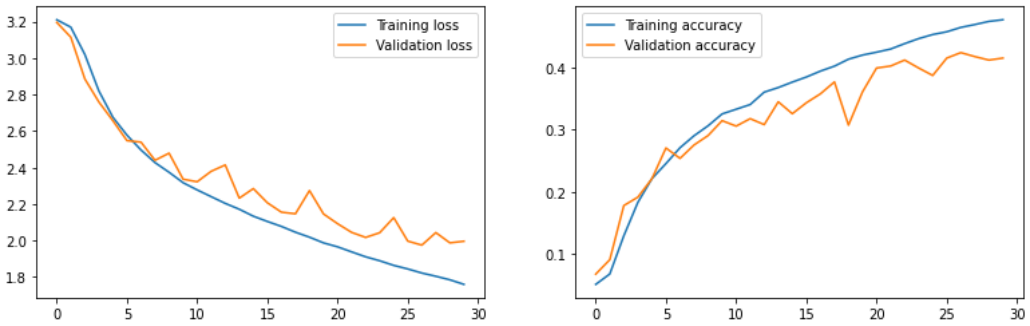
1.2. Apartado 1.2

Entrenar el modelo y extraer los valores de 'accuracy' y función de pérdida para el conjunto de test. Presentar los resultados de entrenamiento y test usando las funciones proporcionadas.

Para el entrenamiento del modelo, se utiza la función *fit*, la cual se encargará de actualizar continuamente los pesos:

fit(x_train, y_train, batch_size, epochs, validation_split)

Donde *validation_split*, como ya se ha mencionado anteriormente, será el 10 % de las imágenes de entrenamiento reservadas para la validación. Si la muestra con la que entrenamos el modelo es lo suficientemente representativa, el conjunto de test deberá obtener resultados semejantes.



(a) Evolución de la función de pérdida. (b) Evolución del accuracy.

Figura 3: Evolución del entrenamiento del modelo.

Se puede comprobar observando los resultados que, a medida que el valor de la función de pérdida va disminuyendo para el conjunto de entrenamiento, también lo hace para el conjunto de validación, a diferencia de que este último, al aumentar las épocas, empieza a tomar valores ligeramente más constantes, dejando de tener la tendencia que toma el conjunto de entrenamiento (empezándose a notar *overfitting*). Este error visualiza un resultado pobre del modelo, debido a que sus valores se quedan entorno a 2, siendo un resultado bastante alto.

Se aprecia el mismo comportamiento con la evolución de la métrica accuracy: a pesar de que el valor de validación suba junto al de entrenamiento, este no se incrementa de la misma manera que el conjunto de entrenamiento durante el paso de las épocas. Además, no consigue superar un valor de precisión del 50 %, ya que se encuentra muy cercano al 0.4, lo que nos lleva a pensar, de nuevo, que el resultado obtenido por el modelo no es satisfactorio.

Para presentar los resultados obtenidos para el conjunto de test, se hará uso de la función *evaluate*, la cual predice la salida para una entrada dada y calcula la función de métrica especificada en el método *compile* ya utilizado.

score = model.evaluate(x_test, y_test)

El valor de la función de pérdida y la métrica estarán almacenados en las posiciones 0 y 1 de la variable dónde se guarda el valor devuelto por evaluate, respectivamente.

Los resultados obtenidos son:

```
TEST ---- Perdida:  2.006
TEST ---- Accuracy:  0.4192
```

Aunque el resultado obtenido depende ligeramente de propia ejecución, no se percibe una variación significactiva entre una ejecución y otra. Por tanto, en base a los valores proporcionados, donde la función de pérdida se encuenta ligeramente cercana al 2 y la precisión se sitúa en torno al 0.4, siendo valores similares a los obtenidos anteriormente en los conjuntos de entrenamiento y validación; se puede confirmar que se obtiene un resultado poco satisfactiorio que podremos intentar mejorar en apartados posteriores de la práctica.

2. Apartado 2: Mejora del modelo BaseNet

Ahora el objetivo es crear, haciendo de elecciones juiciosas de arquitectura e implementación, una red profunda mejorada partiendo de BaseNet. Una buena combinación de capas puede hacer que la precisión del nuevo modelo se acerque al 50 % sobre nuestros datos de CIFAR-100. Para mejorar la red, puede considerar añadir cualquier combinación de entre las siguientes opciones de mejora:

En este apartado, como indica el enunciado, se realizará una serie de mejoras acumulativas sobre nuestro modelo BaseNet con el objetivo de obtener unos valores de error y precisión razonables en el conjunto de validación. Una vez decididas las mejoras, se utilizará el conjunto de test para compararlo con los resultados del apartado anterior.

2.1. Normalización de los datos

La primera mejorar añadida consiste en normalizar a escala los datos de la entrada para conseguir un entrenamiento más sencillo y sólido. Para ello, haremos que los datos de entrada tengan y media $\mu = 0$ y desviación típica $\sigma = 1$.

Se utiliza la clase *ImageDataGenerator*[9]

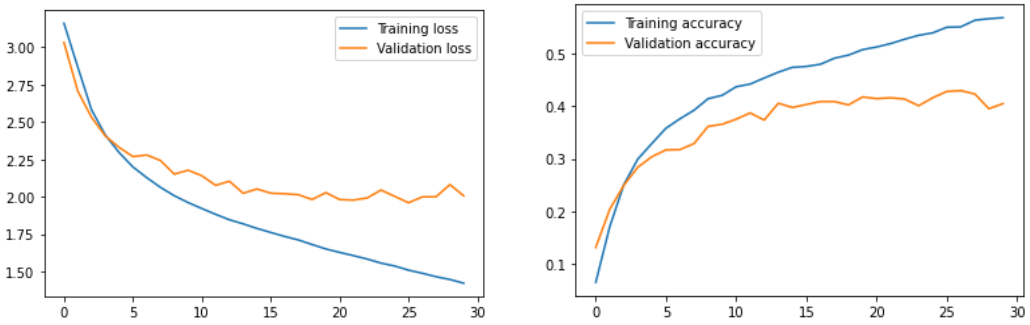
ImageDataGenerator(featurewise_center,featurewise_std_normalization, validation_split)

donde:

- **featurewise_center** establece la media a 0 si su valor es True.
- **featurewise_std_normalization** normaliza la desviación típica si su valor es True.
- **validation_split** establecerá el conjunto de validación, el cual seguirá siendo de un 10 %.

El objeto generado por este método se debe de entrenar con los datos de entrenamiento y tras esto, se generarán dos *flow* para el entrenamiento y validación, donde la función se encarga de cargar el conjunto de datos de imágenes en la memoria y generar lotes de datos aumentados de manera aleatoria. Finalmente, el uso de la función *fit* entrena el modelo con los datos generados batch a batch. Este método es ejecutado en paralelo para una mayor eficiencia.

Los resultados obtenidos son los siguientes:



(a) Evolución de la función de pérdida.

(b) Evolución del accuracy.

Figura 4: Evolución del entrenamiento del modelo tras añadir normalización de datos.

Se puede apreciar claramente cómo ha habido un cambio en la evolución de ambos conjuntos de datos, ya que tras unas pocas épocas los valores de entrenamiento y validación comienzan a ser cada vez más distantes entre ellos: en la función de périda los valores de validación quedan bastante por encima de los de entrenamiento mientras que quedan bastante por debajo en los valores de accuracy.

Por tanto, aunque los valores obtenidos en el conjunto de entrenamiento hayan mejorado con respecto a la propuesta inicial del modelo, el estancamiento en los valores de validación a lo largo de las épocas están señalando la presencia de *overfitting*, puesto que el modelo parece estar memorizando los datos de entrenamiento y aportando buenos resultados con ellos, pero no está aprendiendo de los mismos como para mostrar apenas casi ninguna mejoría en validación.

Aún así, se decide mantener esta mejora debido a que, aunque no haya sido significativa, es conveniente mantener los datos en los mismos rangos para dar robustez al entrenamiento.

2.2. Aumento de los datos

La siguiente mejora planteada consiste en aumentar los datos. Esto conlleva un efecto importante: aumenta la ***diversidad*** de los datos, lo que significa que en cada etapa del entrenamiento el modelo se encontrará con una versión diferente y artificial de las imágenes originales.

Este aumento de la diversidad es necesaria, ya que como hemos podido observar anteriormente, el modelo tiende a ajustarse demasiado a los datos de entrenamiento y a mostrar un rendimiento más deficiente en datos de prueba, conduciendo a errores de predicción. Por lo tanto, la mayor diversidad derivada del aumento de datos reduce la varianza del modelo al mejorar la ***generalización***.

Se probará con las transformaciones indicadas en el enunciado de la práctica, las cuales se añadirán en el método especificado en el apartado anterior: *ImageDataGenerator*:

- **Flip horizontal**: Como su nombre indica, básicamente voltea tanto las filas como las columnas horizontalmente. Resulta una mejora razonable teniendo en cuenta los tipos de clases que se pueden encontrar en el modelo, representando objetos/figuras que no sufrirán una gran diferencia al voltearlas. Para establecer esta modificación, simplemente se deberá establecer el parámetro *horizontal_flip* a verdadero.

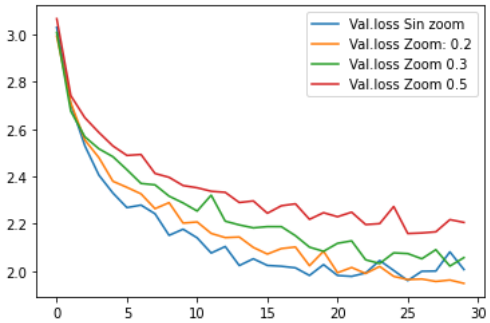
- **Zoom**: Un aumento de zoom amplía aleatoriamente las imágenes y agrega nuevos valores de píxeles alrededor de la imagen o interpolándolos. El rango para el zoom será de *[1-valor , 1+valor]*. Además, cabe mencionar que con valores de zoom inferiores a 1.0 se acercará la imagen mientras que con valores mayores a 1.0 el objeto de la imagen se encontrará más alejado.

  La introducción de zoom puede ser una mejora interesante, puesto que estaría proporcionando información de una imagen a distintas escalas. Se probará con distintos valores, pero para que el conjunto de datos siga siendo interpretable, introduciremos un zoom lo suficientemente bueno como para notar un cambio pero de no más de un 50 % para que los objetos sigan siendo reconocibles. Así pues, probaremos con rangos entre [0.8 , 1.2], [0.7 , 1.3] y [0.5 , 1.5].

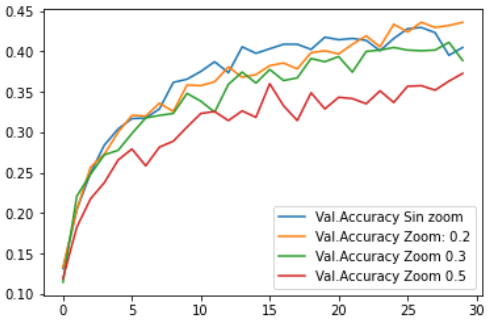
- **Combinación de Flip horizontal + zoom**: Se probará con un combinación con ambas mejoras con el objetivo de obtener la mejor generalización del conjunto de datos posible.

Para saber con qué rango de valores de *Zoom*, se decide visualizar una gráfica que compare los valores de la función de pérdida y accuracy para el conjunto de datos de *validación*. Se recuerda que esta gráfica no será mostrada al ejecutar

el programa puesto que solo se mantendrá el entrenamiento de aquel valor que proporcionó mejores resultados.



(a) Evolución de la función de pérdida.

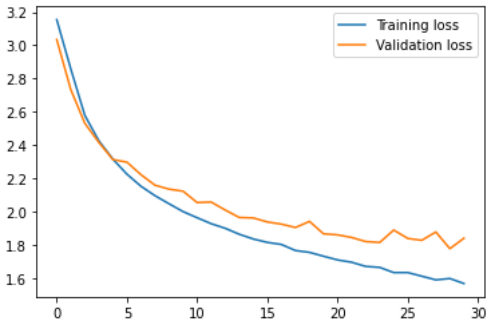


(b) Evolución del accuracy.

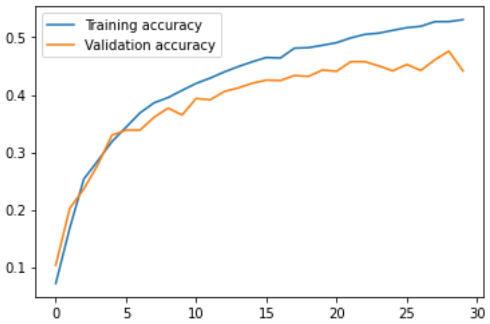
Figura 5: Comparativa de la evolución con el uso de distintos rangos de Zoom para Validación.

Tras mostrar la gráfica, se observa que el comportamiento de ninguno de los rangos de zoom elegidos es el esperado: todos ofrecen peores resultados en comparación a no utilizarlo (línea azúl de la Figura 5). Aún así, nos quedaremos con un Zoom del 20 % (línea naranja), ya que parece que en las últimas épocas alcanza a los resultados que solo usan normalización, para realizar la siguiente comparativa de técnicas.

Ahora la decisión se encuentra entre elegir si usar horizontal flip, un zoom de 0.2 o una combinación de ambos, por lo que se analizan sus gráficas:

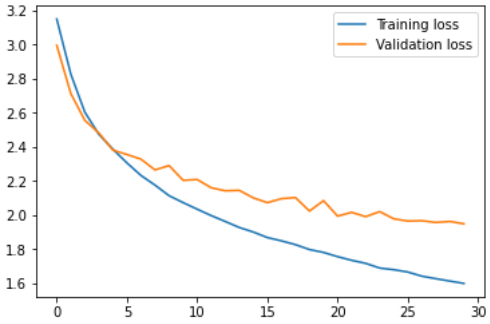


(a) Evolución de la función de pérdida.

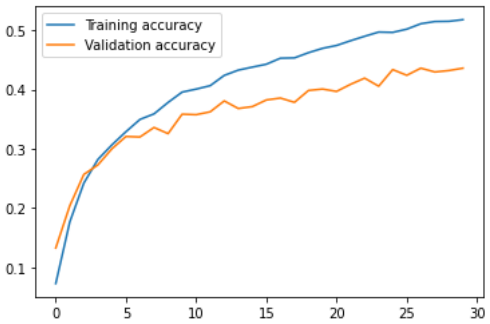


(b) Evolución del accuracy.

Figura 6: Evolución del entrenamiento del modelo tras añadir Horizontal Flip.

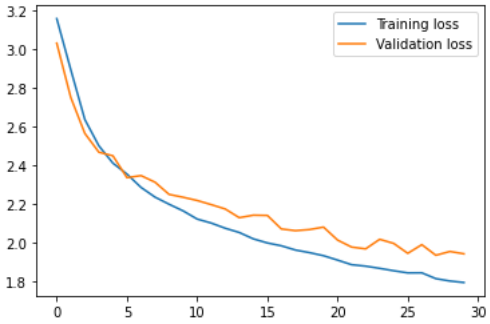


(a) Evolución de la función de pérdida.

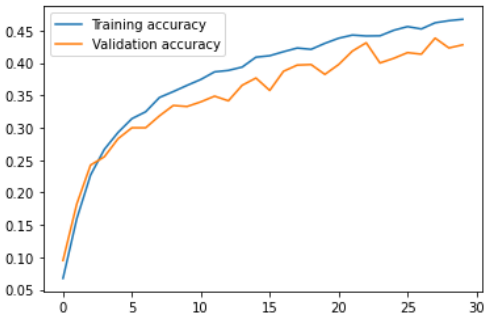


(b) Evolución del accuracy.

Figura 7: Evolución del entrenamiento del modelo tras añadir Zoom de 0.2

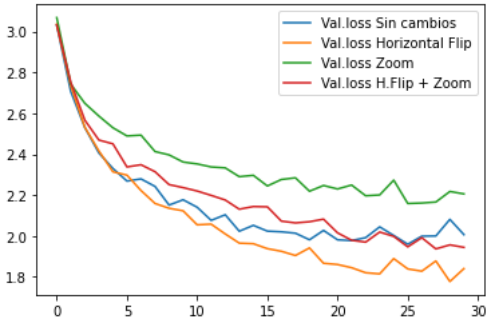


(a) Evolución de la función de pérdida.

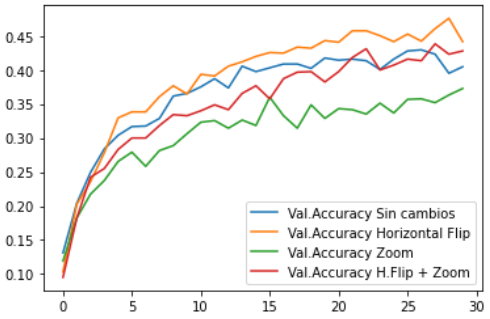


(b) Evolución del accuracy.

Figura 8: Evolución del entrenamiento del modelo tras añadir Horizontal Flip + Zoom de 0.2.



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 9: Comparativa de la evolución con el uso de distintas técnicas de Aumento de Datos.

Tras visualizar la comparativa de los tres tipos de transformaciones, se observa que *horizontal_flip* es la técnica que proporciona mejores aportaciones. Observando las gráficas de la Figura 6 se puede observar como los resultados obtenidos comienzan a bajar ligeramente del 2.0 en la función de pérdida e incluso alcanzan un valor cercano a 0.45 en accuracy, en comparación a solo normalizar los datos, que se quedaba sobre el 0.4.

Cómo estas transformaciones se realizan aleatoriamente, quizás un aumento de épocas hubiese supuesto una mejora más sustancial, a pesar de que añadir este aumento de datos ha resultado en una ejecución algo más lenta. Por esto mismo, se considerará intentar conseguir una mejora más grande en el modelo utilizando solamente los datos normalizados y, en estapas más tardías del entrenamiento, incluir el uso de datos con ***volteos horizontales*** y obtener un ajuste más fino del modelo que le permita realizar una buena generalización.

2.3. Aumento de la profundidad de la red

Se propone como siguiente mejora un aumento en la profundidad de la red, por lo que se añadirán más ***bloques convolucionales***. Dado que cada bloque es una representación diferente de la imagen de entrada, la concatenación de capas convolucionales permitirá una descomposición jerárquica de la misma. Por tanto, añadir más bloques convolucionales permitirá obtener una jerarquía de características más poderosa.

Para la práctica, se va a probar con dos arquitecturas diferentes, aunque ambas mantendrán ciertas características en común que merece la pena destacar:

- **Bloques convolucionales**: Para estas dos arquitecturas nuevas, se hará uso de cuatro bloques convolucionales en lugar de dos, como ocurría con el modelo inicial.
- **Tamaño del kernel**: Se cambiará el uso de kernels de 5 x 5 por el uso de kernels 3 x 3.

  Esta decisión es tomada tras basarnos en contribuciones realizadas a lo largo de la literatura como la de **VGGNet**[10], donde se utiliza varios núcleos de convolución consecutivos de 3 x 3 para reemplazar los núcleos de convolución más grandes (11 x 11, 7 x 7, 5 x 5).

  Esta decisión se debe a que el número de parámetros crece cuadráticamente con el tamaño del kernel. Esto hace que los núcleos de convolución grandes no resulten rentables cuando aumenta el número de canales. Es decir, una convolución 5 x 5 con K características necesitará $25K^2$ pesos, mientras que dos convoluciones 3 x 3 solamente necesitará $9K^2$ pesos.
- **Pooling**: No se añadirá ninguna operación de MaxPooling nueva, lo que quiere decir que únicamente se hará una redimensión cada dos capas convolucionales. Se ha tomado esta decisión ya que colocar siempre una capa de este tipo tras cada capa convolucional conduciría a una pérdida excesiva de información al reducir el número de unidades.

Esta decisiones nos ayudarán a disminuir el coste de las convoluciones y a mejorar el resultado, con la simple intuición de que usaremos kernels más pequeños y tendremos como resultado más capas y redes más profundas con las que obtener una mayor riqueza de información en nuestra jerarquía de representaciones.

Ahora bien, la diferencia subyacente entre las dos arquitecturas propuestas será el número de filtros usado. Se debe de tener en cuenta que los filtros que operan directamente sobre los valores de los píxeles sin procesar (primera capa) aprenderán a extraer características de bajo nivel, como líneas, bordes, esquinas, etc., por lo que comenzar extrayendo cartacterísticas con 6 filtros únicamente, como se propone

inicialmente, puede resultar escaso. Además, las siguientes capas serán composiciones de los filtros de niveles inferiores, por lo que las capas se irán especializando.

Por tanto, se comenzará usando un número de filtros mayor que el inicial y también se irá incrementando en números *potencias de 2* en las siguientes capas. La razón para esta última decisión se debe a que por convención se suele utilizar potencias de 2 en los hiperparámetros cuando no se tiene ningun otro motivo para elegir otros valores específicos[11].

Otra diferencia que encontraremos entre ambas arquitecturas será la adición de una segunda capa densamente conectada en la segunda arquitectura propuesta, formando una red de 500 x 150 x 25 neuronas encargadas de clasificar según la información extraida por la red convolucional.

Por tanto, las arquitecturas propuestas son las siguientes:

Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Conv2D	3	32|30	3|16
2	Relu	-	30|30	-
3	Conv2D	3	30|28	16|32
4	Relu	-	28|28	-
5	MaxPooling2D	2	28|14	-
6	Conv2D	3	14|12	32|64
7	Relu	-	12|12	-
8	Conv2D	3	12|10	64|128
9	Relu	-	10|10	-
10	MaxPooling2D	2	10|5	-
11	Linear	-	3200|150	-
12	Relu	-	150|150	-
13	Linear	-	150|25	-

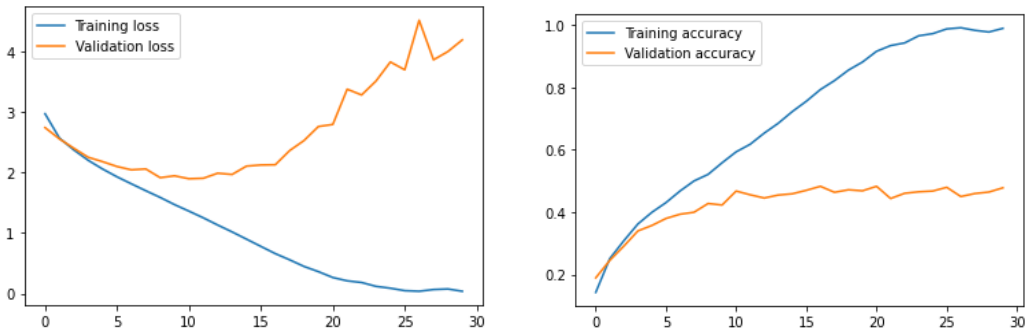
Cuadro 1: Arquitectura del primer modelo propuesto como mejora.

Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Conv2D	3	32|30	3|32
2	Relu	-	30|30	-
3	Conv2D	3	30|28	32|64
4	Relu	-	28|28	-
5	MaxPooling2D	2	28|14	-
6	Conv2D	3	14|12	64|128
7	Relu	-	12|12	-
8	Conv2D	3	12|10	128|256
9	Relu	-	10|10	-
10	MaxPooling2D	2	10|5	-
11	Linear	-	6400|500	-
12	Relu	-	500|500	-
13	Linear	-	500|150	-
14	Relu	-	150|150	-
15	Linear	-	150|25	-

Cuadro 2: Arquitectura del segundo modelo propuesto como mejora.

De momento, y como se ha mencionado en el apartado anterior, el modelo ha sido entrenado con los datos normalizados, pero sin el aumento de datos elegido.

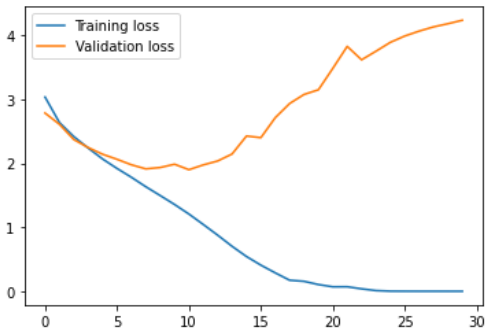
Una vez comentados todos los detalles, se obtienen los resultados:



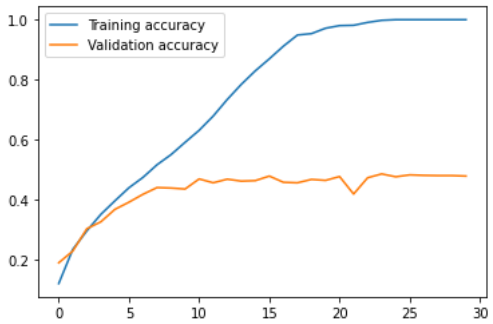
(a) Evolución de la función de pérdida.

(b) Evolución del accuracy.

Figura 10: Evolución de la primera arquitectura propuesta.



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 11: Evolución de la segunda arquitectura propuesta.

No es difícil observar cómo ambos resultados son bastante decepcionantes, ya que existe una gran cantidad de *overfitting*. Nos hemos encontrado con el gran problema de las Redes Neuronales: el modelo usa una cantidad tan grande de parámetros que comienza a fijarse en detalles poco relevantes del problema, en vez de en su esencia, y es capaz de ajustar perfectamente en función a los datos de entrenamiento; es por eso que para este conjunto de datos se observa una tasa de precisión del 100 % y, sin embargo, no está aprendiendo, por lo que los resultados de validación no han sufrido apenas mejora.

En este punto del problema, la mejor solución es intentar disminuir el overfitting que se ha producido antes de tomar ninguna decisión sobre las arquitecturas propuestas.

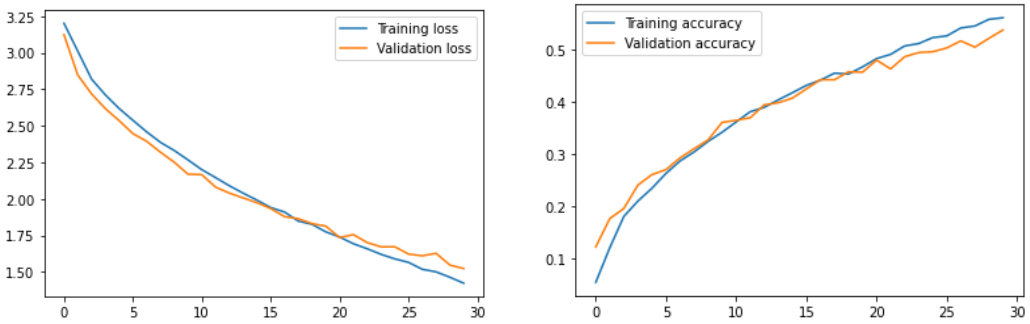
2.4. Dropout

La ***regularización*** consiste en eliminar unidades de los tensores de forma que cuando propagamos un lote hacia arriba para actualizar la red con sus valores, no todas las unidades participarán. Esta es una buena técnica para reducir el *overfitting*, ya que obliga a las unidades utilizadas desacoplarse unas de otras y a especializarse en lo esencial del problema.

Con Dropout se puede llevar a cabo esta técnica, eliminando de modo aleatorio un porcentaje de unidades durante el entrenamiento. No existe un criterio que confirme dónde se debe hacer uso de esta regularización, aunque ateniendo a la literatura [12], se sabe que para los conjuntos de datos CIFAR-10 y CIFAR-100 supone una mejoría añadir Dropout en todas las capas. Por tanto, se decide no solo utilizar esta técnica tras las capas densamente conectadas (donde se utilizan comunmente, al encontrar mucha redundancia en estas capas), sino también tras las capas convolucionales.

Se recomienda, además, usar un porcentaje que se encuentre entre el 20 % y 50 %, ya que una probabilidad muy baja conllevará un efecto mínimo, pero una probabilidad muy alta resultará en que el modelo sufra de *underfitting*, al limintar los parámetros en exceso. Por tanto, se ha decidido usar un porcentaje del 25 % excepto una del 50 % situada tras la última capa convolucional y también en la segunda arquitectura propuesta, detrás de la primera capa densamente conectada, para compensar el aumento del número de parámetros en esas capas.

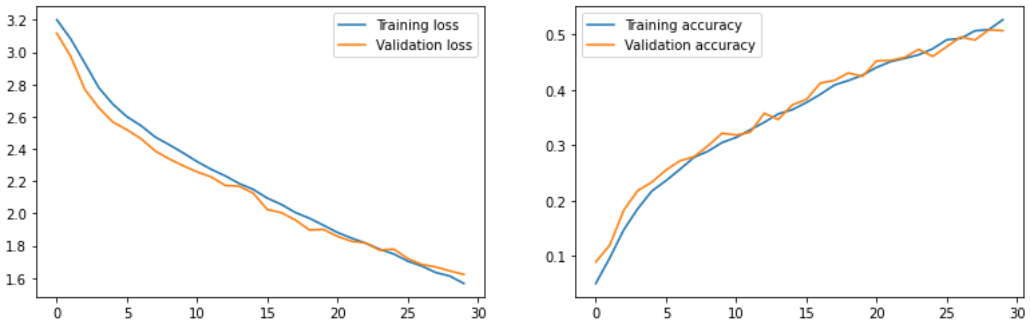
Se obtienen los siguientes resultados:



(a) Evolución de la función de pérdida.

(b) Evolución del accuracy.

Figura 12: Evolución de la primera arquitectura propuesta + Dropout.



(a) Evolución de la función de pérdida.

(b) Evolución del accuracy.

Figura 13: Evolución de la segunda arquitectura propuesta + Dropout.

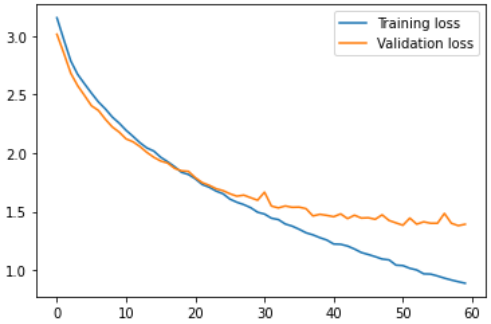
Ahora sí, se ha obtenido un modelo que denota mejoría. Se ha eliminado el overfit que encontrábamos anteriormente y la evolución de los datos de validación mejoran a medida que lo hacen los de entrenamiento.

No solamente encontramos un hecho positivo en que los datos de validación no se quedan estancados, también se ha obtenido mejores resultados a nivel de porcentajes: la función de pérdida alcanza un valor de 1.50 prácticamente mientras que el accuracy ha subido algo más del 0.5, valores muy positivos en comparación a los obtenidos en el entrenamiento inicial.

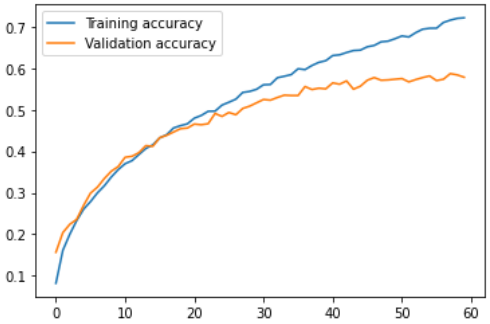
Tras observar la tendencia de las evoluciones, las cuales se mantienen crecientes constantemente, nos lleva a pensar que un aumento de épocas podría ser beneficioso si el comportamiento de los resultados se mantiene. Este aumento podría conllevar una mayor mejoría de nuestras métricas y que el modelo generalice mejor.

2.5. Aumento de épocas

Se realizará de nuevo el proceso anterior aumentando el número de épocas al doble para ver como reacciona el modelo:

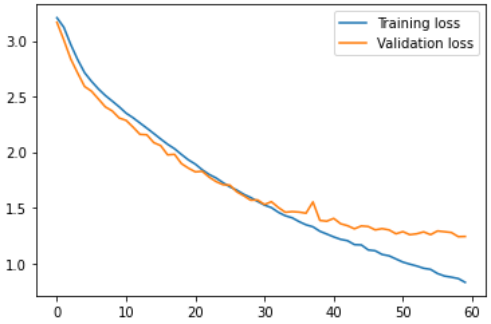


(a) Evolución de la función de pérdida.

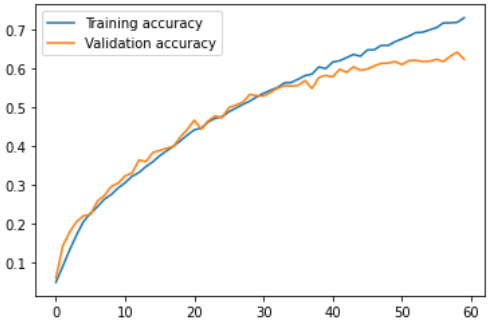


(b) Evolución del accuracy.

Figura 14: Evolución de la primera arquitectura propuesta + Dropout + Aumento de épocas.



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 15: Evolución de la segunda arquitectura propuesta + Dropout + Aumento de épocas.

Tras aumentar las épocas, es cierto que los valores de la función de error en validación se mantienen en torno al 1.5, pero el accuracy sufre una mejora sustancial rondando el 0.6. En ambas arquitecturas se observa que durante las últimas épocas los valores en validación comienzan a estancarse; aún así, se contempla que en la segunda arquitectura los valores de validación son bastante buenos y se mantienen más cercanos a los de entrenamiento. Es por ello que nos decidimos por *la segunda arquitectura propuesta*, con la que podemos probar a usar otros mecanismos de ajuste para intentar mejorar el modelo.

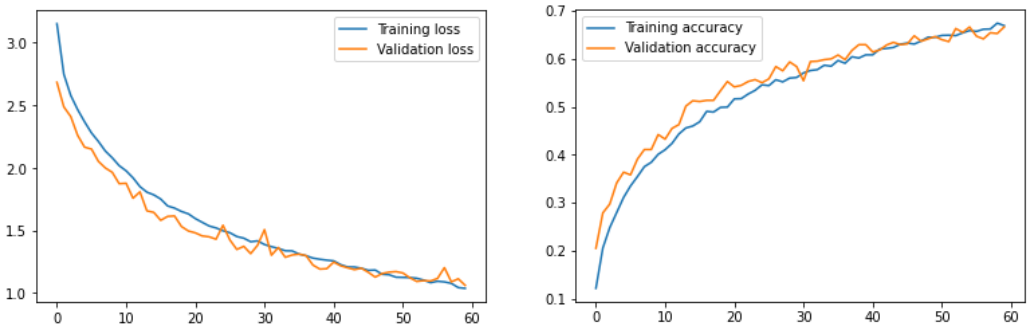
2.6. Batch Normalization

A continuación, se probará a utilizar una Batch Normalization, técnica que normaliza la salida de una determinada capa con la intención de mejorar el entrenamiento. Se ha estudiado que en redes profundas, una tasa de aprendizaje demasiado alto puede causar que los gradientes se atasquen en mínimos locales deficientes pero, sin embargo, Batch Normalization ayuda a abordar estos problemas, ya que al normalizar las activaciones en la red, evita que los pequeños cambios en los parámetros se amplifiquen en cambios más grandes y subóptimos en las activaciones de los gradientes[13].

En el enunciado del problema y en general en la literatura, se recomienda hacerlo después de las capas convolucionales, antes de aplicar la función de activación. Sin embargo, se propone también probar después de dichas capas ReLu, por lo cual se probará de ambas maneras. Además, se colocará delante o detrás de cada función de activación, independientemente de si es tras una capa convolucional o una capa densamente conectada, ya que suele tener un impacto positivo en tras ambos pasos.

Cabe destacar también, que en este punto del problema, se realizará el entrenamiento del modelo añadiéndole *horizontal_flip*, técnica realizada como una de las primeras mejoras, con el objetivo de intentar obtener un resultado lo más generalizado posible.

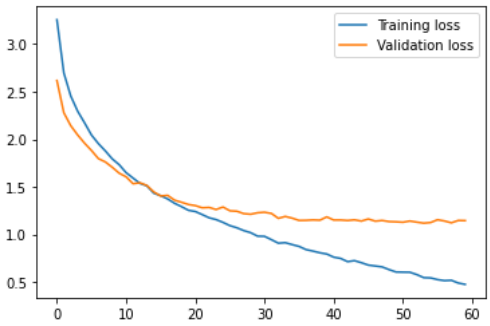
Para ello, solo se debe llamar al método *BatchNormalization()* de **Keras**. Los resultados obtenidos son los siguientes:



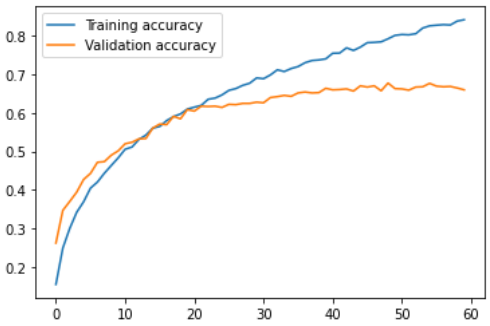
(a) Evolución de la función de pérdida.

(b) Evolución del accuracy.

Figura 16: Evolución del entrenamiento del modelo tras añadir Batch Normalization antes de la función de activación.



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 17: Evolución del entrenamiento del modelo tras añadir Batch Normalization después de la función de activación.

No cabe duda que se puede afirmar que, para nuestro problema, la capa de Batch Normalization actúa mejor si se aplica ***antes*** de la función de activación, ya que colocarla tras esta aumenta el *overfitting* entre ambos conjuntos de datos. Colocarla después ha hecho más estrecha la diferencia entre ambos conjuntos, ya que los valores obtenidos en validación son casi idénticos a los obtenidos en el conjunto de entrenamiento. Se mejoran los resultados de validación los cuales ahora se encuentran alrededor del 1.15 para la función de pérdida y entre un 0.6 - 0.7 en accuracy, siendo un resultado muy positivo.

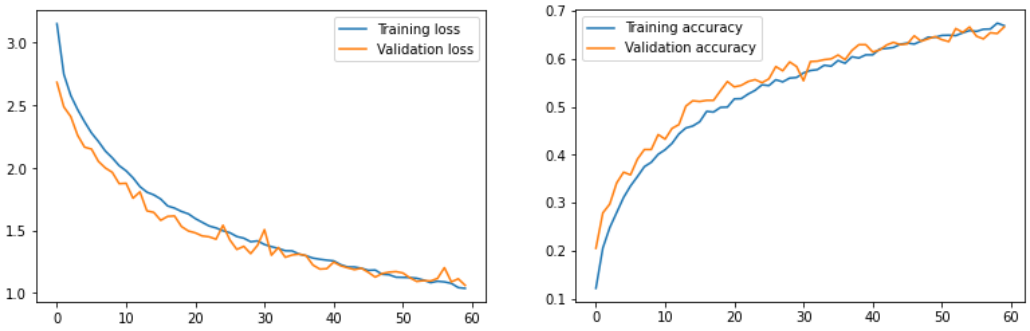
2.7. Estimación del modelo final

En el apartado anterior ya se visualizó claramente cuál es el modelo que mejor se ajusta a nuestro problema. A continuación, se muestra su arquitectura:

Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Conv2D	3	32|30	3|32
2	Batch.Norm	-	30|30	-
3	Relu	-	30|30	-
4	Conv2D	3	30|28	32|64
5	Batch.Norm	-	28|28	-
6	Relu	-	28|28	-
7	MaxPooling2D	2	28|14	-
8	Dropout	-	14|14	-
9	Conv2D	3	14|12	64|128
10	Batch.Norm	-	12|12	-
11	Relu	-	12|12	-
12	Conv2D	3	12|10	128|256
13	Batch.Norm	-	10|10	-
14	Relu	-	10|10	-
15	MaxPooling2D	2	10|5	-
16	Dropout	-	5|5	-
17	Linear	-	6400|500	-
18	Batch.Norm	-	500|500	-
19	Relu	-	500|500	-
20	Dropout	-	500|500	-
21	Linear	-	500|150	-
22	Batch.Norm	-	150|150	-
23	Relu	-	150|150	-
24	Dropout	-	150|150	-
25	Linear	-	150|25	-

Cuadro 3: Arquitectura del modelo final.

Y los resultados obtenidos:



(a) Evolución de la función de pérdida. (b) Evolución del accuracy.

Figura 18: Evolución del modelo con todas las mejoras realizadas.

Para evaluar el conjunto de test, se necesita normalizar el conjunto de test utilizando la media y desviación del conjunto de entrenamiento. Se obtiene de resultado:

```
TEST ------ Perdida:   1.0276
TEST ------ Accuracy:   0.6796
```

Se puede concluir que el resultado es razonablemente satisfactorio, teniendo en cuenta de que el resultado de partida (0.4196) no llegaba siquiera al 50 % de precisión y ahora, tras todas las mejoras, se supera el 65 %.

3. Apartado 3: Transferencia de modelos y ajuste fino con ResNet50 para la base de datos Caltech-UCSD

En este apartado de la práctica, se parte de una Red Neuronal Convolucional compleja y potente: **ResNet50**, la cual puede ser descargada desde la propia librería de **Keras**[15]. Se trabajará haciendo uso de la base de datos **Caltech-UCSD**[14], un conjunto de 6033 imágenes de 20 especies distintas de pájaros. El conjunto de entrenamiento estará compuesto por 3000, donde un 10 % será usado para validación, y las 3033 imágenes restantes serán destinadas al entrenamiento.

El objetivo de este ejercicio será presentar las dos formas de trabajar con arquitecturas ya existentes de Redes Convolucionales:

- **Extractor de características**: donde no se utilizarán las capas finales de la red, ya que estas son las más específicas (y ResNet50 ya ha sido pre-entrenada con ImageNet); pero utilizaremos el grueso del modelo para añadir capas que se adecuen a nuestro problema.
- **Ajuste fino sobre el vector**: donde no se quitan las capas finales, pero se modifican los pesos de la red ligeramente. Esto es útil porque las redes neuronales conocidas y que son utilizadas actualmente ya han encontrado un óptimo, pero debemos realizar modificaciones para encontrar el óptimo de nuestro propio problema.

3.1. Ejercicio 1

Usar ResNet50 como un extractor de características para los datos de Caletch-UCSD. En concreto realizar los siguientes experimentos:

3.1.1. Apartado A

a) Adaptar el modelo ResNet50 entrenado con ImageNet a los datos de Caltech-UCSD y estimar su desempeño con estos datos. b) Eliminar las capas finales FC y la salida, sustituirlas por nuevas FC y salida y re-entrenarlas con Caltech-UCSD. c) Comparar resultados con el modelo anterior en el que únicamente se cambia y reentrena la salida.

En este primer apartado se pide eliminar la salida y las capas finales *Fully connected* (la última capa de ResNet50) y sustituirlas por nuevas capas densamente conectadas, de manera que se deberá entrenar las capas añadidas.

Primero, se debe preprocesar nuestras imágenes antes de pasárselas a ResNet50 y para ello se hará uso del método *ImageDataGenerator*, aportándole una función de preprocesado proporcionada por **Keras**: *preprocess_input*, la cual simplemente

adapta las imágenes al formato requerido por el modelo.

Cabe destacar que tras crear el objeto devuelto por *ImageDataGenerator* no necesitará ningún ajuste con el método *fit*, al contrario del ejercicio anterior, ya que *preprocess_input* no realiza ninguna normalización de datos[17].

Tras esto, se define el modelo **ResNet50**[15] preentrenado en ImageNet y sin la última capa:

ResNet50(include_top, weights, pooling)

donde:

- **include_top** indica si incluir o no la capa densamente conectada, por lo cual se establecerá a *False*.
- **weights** indica los pesos. Se igualará este parámetro a *'imagenet'* para que sea pre-entrenados con ellos.
- **pooling** cuando include_top es False, se elige el tipo de pooling a emplear. Se establecerá este parámetro a *'avg'* para emplear un AveragePooling.

Una vez definido el modelo, se extraen las características de los conjuntos entrenamiento y test mediante el método *predict*. Dicho método devuelve un array de predicciones para las imágenes de entrada (pasadas en lotes mediante el método *flow*); sin embargo, nuestro ResNet50 del cual se le ha eliminado la parte clasificadora, estará transformando cada imagen en un vector de 2048 características. Estas características serán la entrada del modelo que se creará a continuación.

De primeras, se pide crear un modelo que contenga únicamente una capa densamente conectada adecuada a la dimensionalidad del problema. Al eliminar la capa totalmente conectada de ResNet50, el vector de características extraído por Resnet50 tendrá dimensionalidad *(2048,)*, el cual será el tamaño de entrada de nuestra capa. La capa a añadir, además, será de *200* unidades, es decir, el número de clases a clasificar que tiene nuestro problema seguida de la función clasificadora *softmax*. Por tanto, nuestro modelo simplemente será:

Tras la definición del modelo, se compila haciendo uso de los mismos parámetros que se utilizaron para la realización del ejercicio 2: una función de pérdida de *entropía cruzada* por tratarse de un problema de clasificación, *Gradiente Descendente Estocástico, SGD* como optimizador y de métrica, *accuracy*.

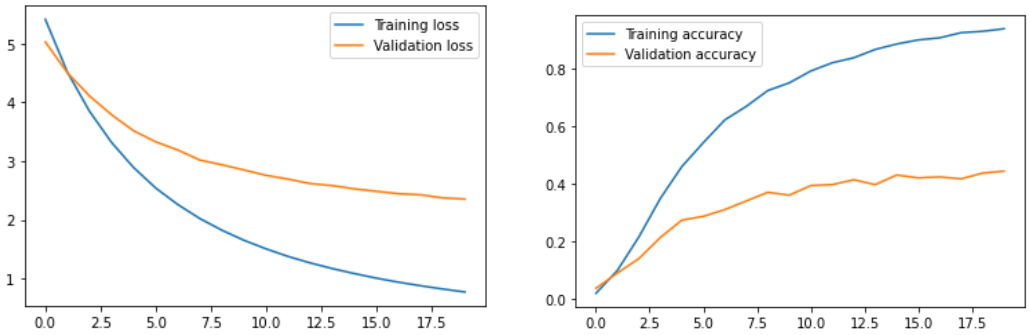
Se entrena el modelo con las carcaterísticas obtenidas anteriormente, un tamaño de batch de 32 (al igual que en el ejercicio anterior, el cuál ofreció un buen funcionamiento) y un porcentaje de validación del 10 %. Se ha decidido entrenar el

Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Lienar	-	2048|200	-

Cuadro 4: Arquitectura del modelo clasificador (usado tras ResNet50) para reentrenar con Caltech-UCSD.

modelo haciendo uso de 20 épocas aunque en función de los resultados obtenidos, más adelante se decidirá si modificar este parámetro o no.

Los resultados obtenidos son los siguientes:



(a) Evolución de la función de pérdida. (b) Evolución del accuracy.

Figura 19: Evolución del clasificador de 200 neuronas + extractor de características ResNet50.

```
TEST ---- Perdida:  2.561
TEST ---- Accuracy:  0.394
```

En el ejercicio también se pide añadir, a parte de una capa densamente conectada de la dimensionalidad de nuestro problema, otro tipo de capas (Dropout, BatchNormalizarion, varias FCs...).

ResNet50, que es de por sí una red potente, está siendo entrenada con muchas imágenes (ImageNet), y va a ser usada en un conjunto muy reducido en comparación (Caltech-UCSD); por lo que se decide no usar más de dos capas densamente conectadas para el clasificador.

La primera capa densa utilizará 1024 unidades, es decir, la mitad del tamaño

completo del vector de cartacterísticas, y la segunda capa será de nuevo 200.

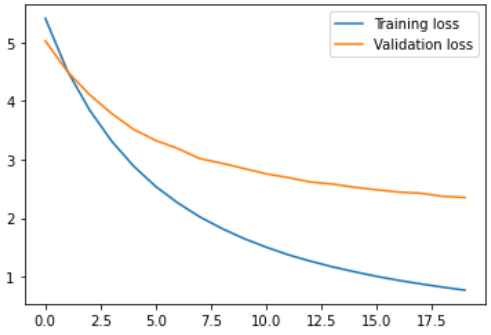
En un intento de reducir el *overfitting* en la mayor medida posible, se introducirá una capa de un Dropout de 0.4 ,ya que como en la primera capa densamente conectada tenemos una gran cantidad de neuronas, se ajustará una probabilidad casi del 50 % de que se ignore una neurona sin llegar a producir una regularización agresiva que impida al modelo aprender.

Por tanto, la arquitectura propuesta se expone a continuación:

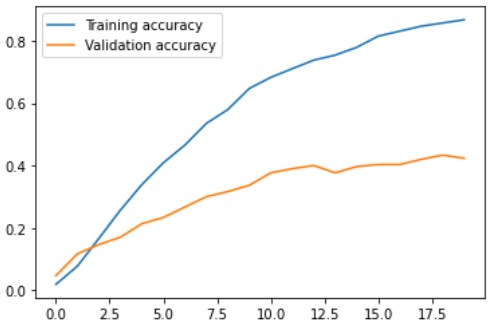
Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Lienar	-	2048|1024	-
2	Relu	-	1024|1024	-
3	Dropout	-	1024|1024	-
4	Linear	-	1024|200	-

Cuadro 5: Segunda arquitectura del modelo clasificador (usado tras ResNet50) para reentrenar con Caltech-UCSD.

Cuyos resultados ofrecidos son:



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 20: Evolución del clasificador de 1024 x 200 neuronas + extractor de características ResNet50.

```
TEST ------ Perdida:   2.363
TEST ------ Accuracy:   0.414
```

Tras observar los resultados de la Figura 19, donde solamente se añade una capa densamente conectada de la dimensionalidad de nuestro problema, se aprecia como los resultados dejan mucho que desear. Los resultados ofrecidos en entrenamiento convergen fácilmente, acercándose al 0 en la función de pérdida y al 1 en accuracy. Por el contrario, los valores de validación tienen un comportamiento completamente diferente, ya que apenas llegan a un 40 % de precisión. Tras medir su desempeño con el conjunto de test, se confirma que los resultados del modelo son bastante decepcionantes.

Se aprecia una gran cantidad de *overfitting*. Se podría pensar que con la siguiente estructura propuesta, al utilizar regularización, este overfitting podría reducirse, pero desgraciadamente, tras ver los resultados de la Figura 20, aunque es cierto que el sobreajuste parece ser algo menor ya que el desempeño de los valores en entrenamiento han empeorado ligeramente en comparación con la otra arquitectura propuesta, los valores de validación parecen seguir siendo bastante malos, subiendo algo más del 40 % pero no marcando, en absoluto, una mejora significativa.

Al comprobar los resultados obtenidos en test, y como se ha dicho anteriormente, sufren una leve pero no sustancial mejora.

La causa de este sobreajuste puede deberse a que estamos utilizando un extractor de características entrenado con millones de instancias sobre un conjunto muy pequeño, de poco más de 6000 imágenes. Este gran número de parámetros hace que el modelo no sea capaz de generar bien y acaba ajustándose y memorizando por completo los datos de entrenamiento, pero sin aprender de verdad, por lo que no consigue un buen desempeño ni en validación ni en test.

3.1.2. Apartado B

Eliminar las capas de salida FC y AveragePooling. En este momento tendrá un extractor de características. Añada nuevas capas que mezclen dichas características y den una clasificación. Entrene la red resultante y compare sus resultados con los del punto A.

En este apartado se procede de manera similar al anterior, con la diferencia de que nuestra definición del modelo **ResNet50** será sin la capa de *AveragePooling*. Para ello, en la definición del modelo se establecerá el parámetro *pooling* a None, de manera que la salida del modelo será el tensor de salida del último bloque convolucional.

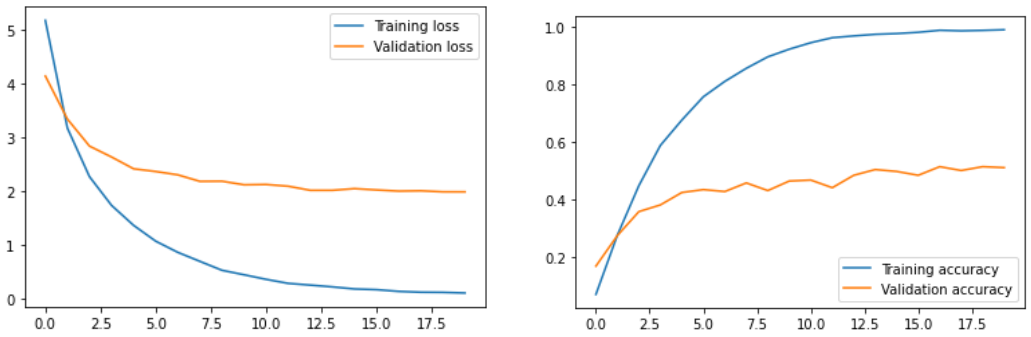
A parte de las capas densamente conectadas que se mantendrán iguales con respecto al modelo anterior, se añade una capa convolucional con *512* filtros. El tamaño del kernel utilizado será de 3 x 3 para intentar realizarla con un coste poco

elevado y se añadirá una capa de BatchNormalization y de Dropout para intentar reducir el sobreajuste. También se añadirá una capa *GlobalAveragePooling2D* tras el bloque convolucional añadido en compensación a la que hemos eliminado para añadir más capas.

La arquitectura del modelo y resultados se muestran a continuación:

Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Conv2D	3	7|5	2048|512
2	Relu	-	5|5	-
3	Batch.Norm	-	5|5	-
4	Avg.Pool	-	5|5	-
5	Dropout	-	5|5	-
6	Linear	-	12800|1024	-
7	Relu	-	1024|1024	-
8	Batch.Norm	-	1024|1024	-
9	Dropout	-	1024|1024	-
10	Linear	-	1024|200	-

Cuadro 6: Tercera arquitectura del modelo clasificador (usado tras ResNet50) para reentrenar con Caltech-UCSD.



(a) Evolución de la función de pérdida.

(b) Evolución del accuracy.

Figura 21: Evolución del modelo con capas que mezlcen características + clasificación

```
TEST ---- Perdida:  2.177
TEST ---- Accuracy:  0.471
```

Aunque el desempeño de los datos de validación tras mezclar las características con nuevas capas parece haber mejorado un poco en comparación con lo obtenido en el apartado A, puesto que parece haber alcanzado una precisión de aproximadamente 0.5, seguimos encontrando una gran cantidad de overfitting en nuestros datos. Al final, la opción más coherente sería utilizar una base de datos que contuviese un mayor número de imágenes. Al calcular los datos en el conjunto de test, nos hemos quedado cerca de alcanzar el 50 % de precisión pero, aún así, no se ha obtenido un resultado satisfactorio.

3.2. Ejercicio 2

Realizar un ajuste fino de toda la red ResNet50, al conjunto de datos Caltech-UCSD. Recordar que el número de épocas debe ser pequeño.

En este apartado, se utiliza ResNet50 para realizar un ajuste fino, también llamado *fine-tunning*, donde se reentrenará la red adaptándola a nuestro problema concreto. Para ello, como mínimo es necesario añadir al final del modelo una capa densamente conectada con tantas neuronas como clases tenga nuestro problema, es decir: 200, y la activación softmax.

En nuestro caso, se ha decidido utilizar más de una capa densamente conectada, de igual manera que se hizo en el ejercicio anterior, de la mitad del tamaño del vector de cartacterísticas, utilizando 1024 x 200 neuronas. A diferencia de los ejercicios anteriores, esta vez el modelo es de tipo *Model()*, permitiendo ciclos y saltos entre capas, como ocurre en ResNet50.

De esta forma, se trabajará con un modelo *end-to-end* comportándose como un único bloque que tiene de entrada las imágenes y de salida las probabilidades devueltas por la activación softmax:

Model(inputs, outputs)

donde:

- **inputs** será resnet50.input.
- **outputs** será igualada a la predicción devuelta por la capa de salida.

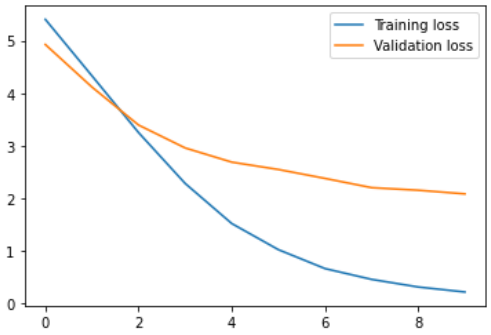
La arquitectura utilizada se mantiene como la del apartado 3.A.b:

Layer No.	Layer Type	Kernel size (conv layers)	Input|Output dimension	Input|Output channels (for conv layers)
1	Lienar	-	2048|1024	-
2	Relu	-	1024|1024	-
3	Dropout	-	1024|1024	-
4	Linear	-	1024|200	-

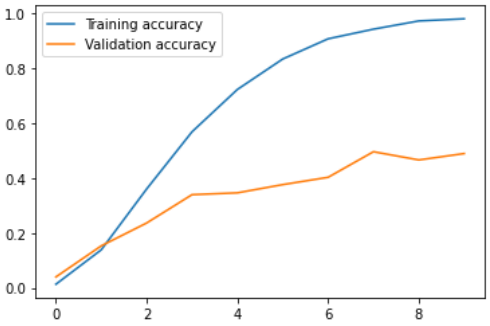
Cuadro 7: Arquitectura propuesta para realizar el ajuste fino de ResNet50.

Es importante congelar la base convolucional creada con nuestra red ResNet50 antes de compilar y el modelo. Esto se realiza mediante ***resnet50.traineable= False***, evitando que los pesos de una capa determinada se actualicen durante el entrenamiento.

Tras compilar y entrenar el modelo, los resultados son los siguientes:



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 22: Evolución del modelo mediante extracción de características fijas.

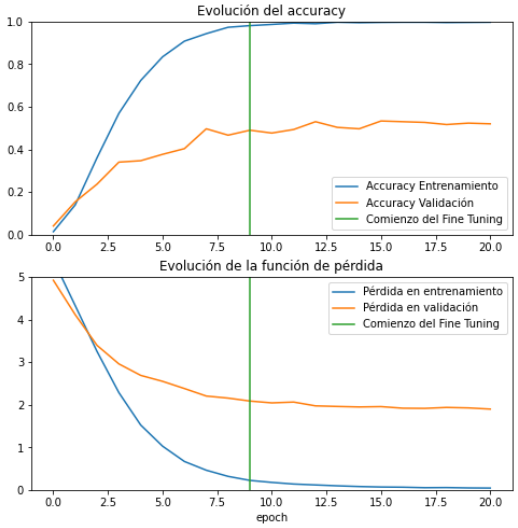
```
TEST ——— Perdida:  2.454
TEST ——— Accuracy: 0.414
```

Se ha utilizado por recomendación un número de épocas pequeño, quizás debido al elevado tiempo de ejecución que toma entrenar el modelo, por lo que solamente se realizaron 10 épocas en este expreimento. Ahora mismo simplemente se ha hecho, de nuevo, otra extracción de características mediante el entrenamiento de algunas capas sobre el modelo base ResNet50.

Una forma de aumentar más el rendimiento es *ajustar los pesos* de las capas superiores del modelo previamente entrenado junto con el entrenamiento del clasificador que se ha agregado. El proceso de entrenamiento obligará a ajustad las ponderaciones de mapas de características genéricas a características asociadas específicamente con el conjunto de datos.

Mediante ***resnet50.traineable = True*** se descongelan las capas superiores y se vuelve a compilar el modelo que hemos establecido para que los cambios surjan efecto y poder reanudar el entrenamiento. Esta continuación del entrenamiento consiste en que, habiendo entrenado ya 10 épocas, ahora se entrenarán otras 10 épocas siguientes, partiendo desde la última época calculada anteriormente.

Tras realizar este proceso, se obtienen los siguientes resultados:



(a) Evolución de la función de pérdida.

Figura 23: Evolución del modelo mediante ajuste refinado de ResNet50.

```
TEST ------ Perdida:  2.326
TEST ------ Accuracy: 0.446
```

Al observar los resultados, se aprecia una ligera mejoría en los resultados obtenidos con respecto a la Figura 22 (antes de que comenzase el fine-tuning), aunque no llega a ser sustancial: simplemente se pasa de un 0.40 aproximadamente a casi 0.45 de precisión (según se observa en las gráficas y en los valores obtenidos para el conjunto de test). Teniendo en cuenta, además, de que se ha utilizado la misma arquitectura que en el apartado 1.A.b) de este mismo ejercicio, hacer un ajuste refinado si ha mejorado un poco con respecto a utilizar ResNet50 únicamente como extractor de características, donde solo se conseguía un 0.41 de accuracy en test.

Por desgracia, esto no ha conseguido eliminar el overfitting, tarea que, aunque se auemente el número de épocas o se cambie de arquitectura haciendo uso de diferentes combinaciones con las técnicas propuestas, o incluso mediante un aumento de datos (si el objetivo de estos ejercicios hubiese sido mejorar los resultados obtenidos), parece difícil de lograr. El uso de una red menos profunda o la elección de un conjunto de imágenes más abundante seguramente hubiese proporcionado mejores resultados.

4. Bonus

Hay muchas otras posibilidades para mejorar el modelo BaseNet sobre CIFAR-100 usando combinaciones adecuadas de capas. Siéntase libre de probar sus propias ideas o enfoques interesantes de AA / VC sobre los que haya leído.

En el mundo de la Visión por Computador, evaluar qué elementos aportan o no en una red neuronal es un proceso empírico de prueba y error, por lo que queda fuera de nuestro alcance poder comprobar las miles de combinaciones de arquitecturas e hiperparámetros. Es por eso que para la realización de este bonus se elige una mejora que, aún no encontrándose dentro de las opciones de mejoras del ejercico 2 de esta misma práctica, tiene una gran posibilidad de ser prometedora a mejorar nuestro modelo BaseNet.

La mejora propuesta se basa en evaluar el comprotamiento de disintos ***optimizadores*** sobre el modelo ya mejorado anteriormente. Los optimizadores son algoritmos utilizados para cambiar los pesos ω de la función de coste $J(\omega)$ de la Red Neuronal con el objetivo de reducir las pérdidas.

Durante la práctica, se ha hecho uso de *Gradiente Descendente Estocástico, SGD*, que aunque ha generado de por sí buenos resultados en nuestro problema, es un algoritmo que puede generar complicaciones a la hora de converger en un extremo, donde la tasa de aprendizaje (*learning_rate*), la cual controla a qué velocidad se adapta el modelo a un problema, es un factor crucial: establecerlo demasiado alto pude hacer que el algoritmo diverja, y establecerlo muy bajo puede prolongar en exceso el tiempo de convergencia.

Aunque se recomienda generalmente decrementar gradualmente la tasa de aprendizaje a lo largo del tiempo [19], se va a evaluar distintas políticas de adaptación para la velocidad y dirección con la que debemos movernos por la superficie:

- **AdaGrad** [20]: La motivación detrás de AdaGrad, *Adaptive Gradient*, es tener una tasa de aprendizaje adaptativa para cada uno de los pesos. Dicha tasa disminuirá con el número de iteraciones.

  La idea de utilizar diferentes tasas de aprendizaje es para poder adaptarse tanto a las *características densas* (aquellas que tienen un gran número de características distintas de cero y que necesitan actualizaciones más pequeñas) y a las *funciones escasas* (aquellas con gran cantidad de características de valor cero, donde las actualizaciones pueden ser más grandes).

Por tanto, AdaGrad adapta la tasa de aprendizaje de cada peso dividiéndolo por la raíz cuadrada de la suma de sus *valores históricos* al cuadrado.

$$\omega_t = \omega_{t-1} - \eta' \bigtriangledown J(\omega)$$

donde:

$$\eta'_t = \frac{\eta}{\sqrt{(\alpha_t + \epsilon)}} \quad \text{y} \quad \alpha_t = \sum_{i=1}^{t-1} (\bigtriangledown J(\omega))^2 \tag{1}$$

siendo η la tasa de aprendizaje inicial, ϵ el término de suavizado que evita la división por cero y ω los pasos de los parámetros.

Los parámetros establecidos por **Keras** son los siguientes:

Adagrad(learning_rate, initicial_accumulator_value, epsilon)

donde

- **learning_rate** es el valor inicial de la tasa de aprendizaje. Se establece a 0.001.
- **initial_accumulator_value** es el valor incial de los acumuladores (valores de impulso por parámetro). Se establece a 0.1.
- **epsilon** es un valor flotante pequeño utilizado para mantener estabilidad numérica. Se establece a 1e-07.

- **RMSProp** [21]: *Root mean square propagation* es muy similar a AdaGrad, cuya diferencia está en la forma en que se gestionan los gradientes anteriores: mientras que AdaGrad tiene en cuenta todo el historial de gradientes, RMSProp solo considera los gradientes actuales para el peso.

  Para ello, hace uso de promedios de gradientes móviles ponderados exponencialmente para evitar dismuir las tasas de aprendizaje de manera muy agresiva.

$$\omega_t = \omega_{t-1} - \eta' \bigtriangledown J(\omega)$$

donde:

$$\eta'_t = \frac{\eta_{t-1}}{\sqrt{(v_t + \epsilon)}} \quad \text{y} \quad v_t = \gamma v_{t-1} + (1-\gamma) \bigtriangledown J(\omega_{t-1}))^2 \tag{2}$$

Los parámetros establecidos por **Keras** son los siguientes:

RMSprop(learning_rate, rho, momentum, epsilon, centered)

donde

- **learning_rate** es el valor inicial de la tasa de aprendizaje. Se establece a 0.001.
- **rho** es el factor de descuento para el historial/próximo gradiente. Se establece a 0.9.
- **momentum** utilizado para cambiar la dirección de avance mediante combinación lineal de gradientes anteriores y actual. Por defecto se encuentra a 0.0.
- **epsilon** es un valor flotante pequeño utilizado para mantener estabilidad numérica. Se establece a 1e-07.
- **centered** se establece a False para que los gradientes se normalicen por el segundo momento descentrado. Aunque establecerlo a True puede ayudar con el entrenamiento, estaríamos añadiendo más complejidad en términos de cálculo y memoria.

- **AdaDelta** [22]: es también una extensión de AdaGrad, y similar a RMSProp, donde su mejora recae en la introducción de una *ventana de historial* que establece un número fijo de gradientes pasados a tener en cuenta durante el entrenamiento. Se adapta dinámicamente en el tiempo usando solo información de primer orden.

$$\omega_t = \omega_{t-1} - \eta' \bigtriangledown J(\omega)$$

donde:

$$\eta'_t = \frac{\sqrt{D_{t-1}+\epsilon)}}{\sqrt{(v_t+\epsilon)}}$$

con:

$$D_t = \gamma v_{t-1} + (1-\gamma) v_t^2 \quad \text{y} \quad v_t = \gamma v_{t-1} + (1-\gamma) \bigtriangledown J(\omega_{t-1}))^2 \tag{3}$$

Los parámetros establecidos por **Keras** son los siguientes:

Adadelta(learning_rate, rho, epsilon)

donde

- **learning_rate** es el valor inicial de la tasa de aprendizaje. Se establece a 0.001.
- **rho** es el factor de descuento para el historial/próximo gradiente. Se establece a 0.95.
- **epsilon** es un valor flotante pequeño utilizado para mantener estabilidad numérica. Se establece a 1e-07.

- **Adam** [23]: *Adaptive Moments* es una actualización de RMSProp donde se acelera y desacelara gracias al *momentum* a la vez que mantiene la tasa de aprendizaje adaptativa. Adam emplea un promedio en decadencia exponencial de gradientes pasados (donde tal promedio de gradientes ayudan a amortigurar las oscilaciones y acelerar el aprendizaje).

$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(v_t+\epsilon)}} * m_t$$

donde:

$$m_t = \gamma_1 m_{t-1} + (1-\gamma_1) \bigtriangledown J(\omega_{t-1})) \quad \text{y} \quad v_t = \gamma_2 v_{t-1} + (1-\gamma_2) \bigtriangledown J(\omega_{t-1}))^2 \quad (4)$$

Los parámetros establecidos por **Keras** son los siguientes:

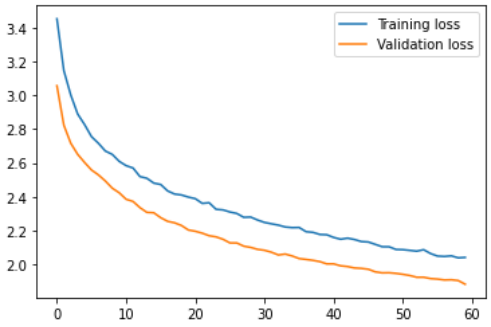
Adam(learning_rate, beta_1, beta_2, epsilon, amsgrad)

donde

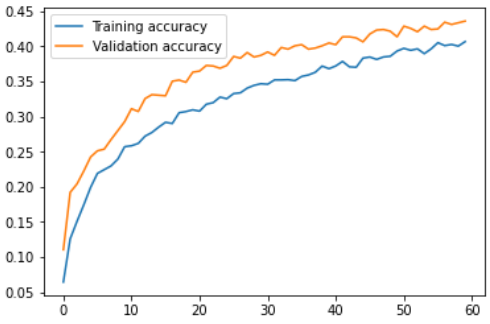
- **learning_rate** es el valor inicial de la tasa de aprendizaje. Se establece a 0.001.
- **beta_1** es la tasa de caída exponencial apra las estimaciones de primer momento. Se establece a 0.9.
- **beta_2** es la tasa de caída exponencial apra las estimaciones de segundo momento. Se establece a 0.999.
- **epsilon** es un valor flotante pequeño utilizado para mantener estabilidad numérica. Se establece a 1e-07.

- **amsgrad** por defecto se encuentra a False para no aplicar la variante de AMSGrad.

Se muesta a continuación los resultados obtenidos sobre nuestro modelo:

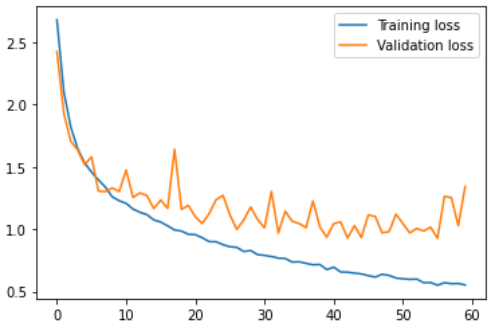


(a) Evolución de la función de pérdida.

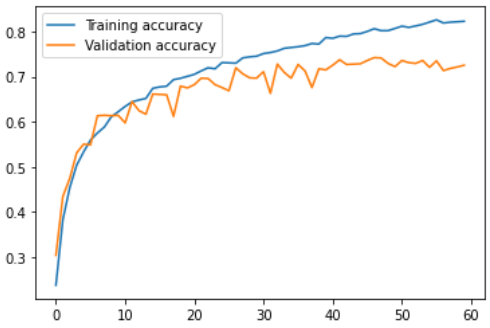


(b) Evolución del accuracy.

Figura 24: Evolución del entrenamiento del modelo utilizando optimizador AdaGrad.

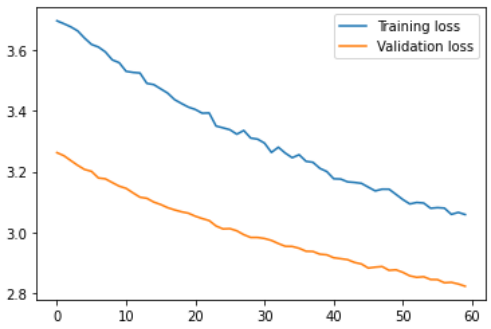


(a) Evolución de la función de pérdida.

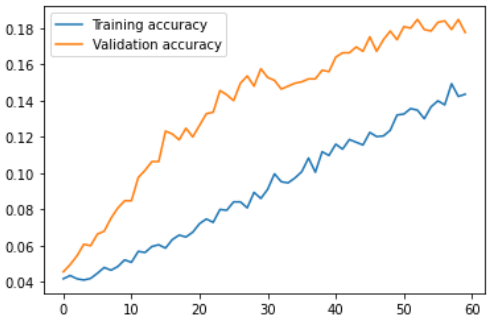


(b) Evolución del accuracy.

Figura 25: Evolución del entrenamiento del modelo utilizando optimizador RMSProp.

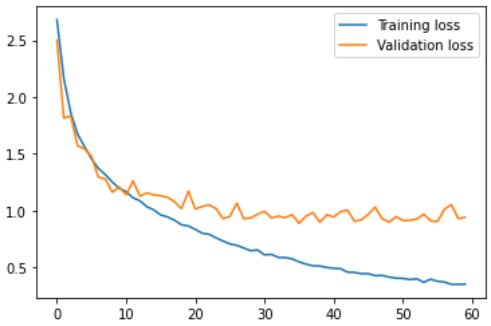


(a) Evolución de la función de pérdida.

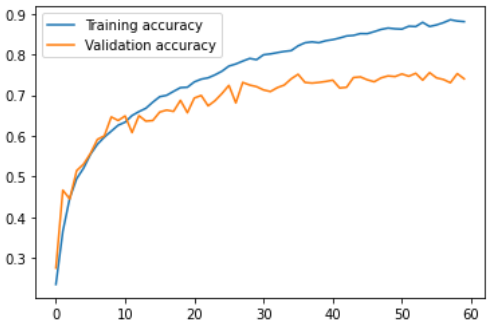


(b) Evolución del accuracy.

Figura 26: Evolución del entrenamiento del modelo utilizando optimizador AdaDelta.

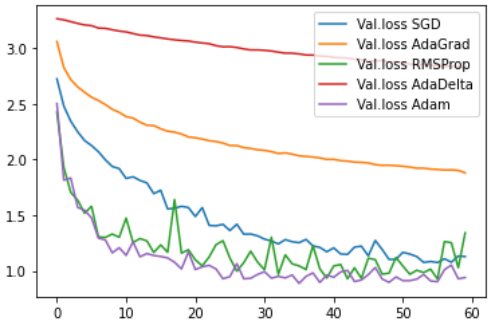


(a) Evolución de la función de pérdida.

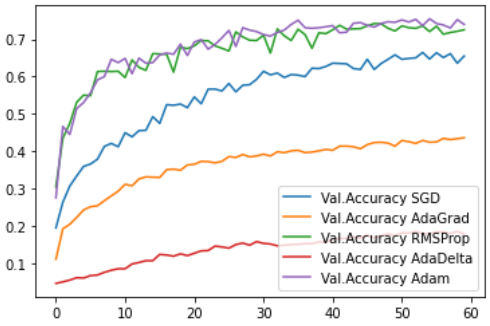


(b) Evolución del accuracy.

Figura 27: Evolución del entrenamiento del modelo utilizando optimizador Adam.



(a) Evolución de la función de pérdida.



(b) Evolución del accuracy.

Figura 28: Comparativa de la evolución con el uso de distintas técnicas de optimización en validación.

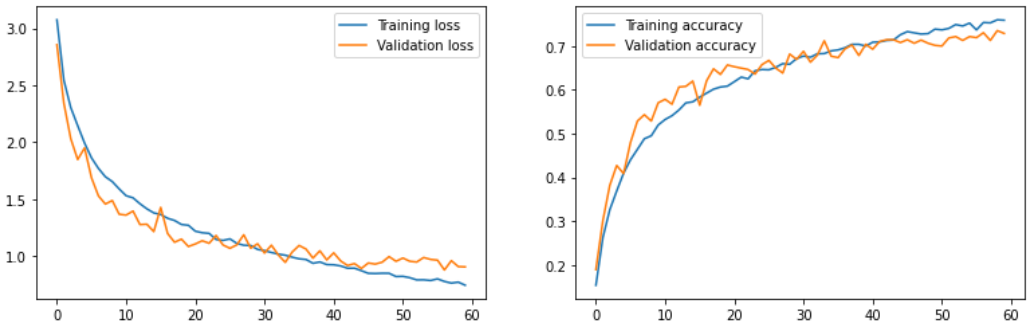
No es dificil darse cuenta de que los optimizadores *AdaGrad* y *AdaDelta* no han dado, en absoluto, buenos resultados. Al contrario, han empeorado bastante nuestro modelo. En cuanto a AdaGrad, se puede observar que, aunque el accuracy no haya superado el 0.45 en 60 épocas, mantiene una tendencia creciente constantemente; por lo que su mal desempeño puede deberse a su principal desventaja: es increíblemente lento. Esto se debe a que la suma del gradiente al cuadrado solo crece y nunca se reduce, por lo que una gran cantidad de iteraciones conduce a una convergencia lenta.

AdaDelta ofrece con diferencia el peor resultado, siendo un hecho que resulta llamativo ya que es un algoritmo muy similar a RMSProp, el cual ha tenido un comportamiento completamente difertente. El numerador en AdaDelta actúa como un término de aceleración, acumulando gradientes previos en una ventana, mientras que el denominador se relaciona con AdaGrad, en el sentido de que la información del gradiente al cuadrado por dimensión ayuda a igualar el proceso realizado en cada una de estas dimensiones, pero se calcula en una ventana para que el progreso se realice más adelante en el entrenamiento [22]; y esto podría ser justamente el motivo de los resultados obtenidos: el uso de la ventana no se adecúa bien para el número de épocas que utilizamos en nuestro problema, dejándonos sin garantía alguna de convergencia.

Por tanto, la decisión final queda entre *RMSProp* Y *Adam*, los cuales los dos ofrecen resultados muy similares; sin embargo, atendiendo a la evolución de pérdida, Adam parece ofrecer un mejor comportamiento (sin oscilaciones tan remarcadas) y resultados, ofreciendo unos valores más cercanos al 1.0 que RMSProp. Por lo cual, se decide ***Adam*** como el algoritmo de optimización que mejores prestaciones ofrece.

Sin embargo, en sus resultados se puede observar como con el paso de las épocas se va incrementando ligeramente el *overfitting*, por lo que además se ha decidido emplear un Dropout más agresivo que el que se utilizó en el Ejercicio 2 de esta misma práctica, estableciendo un porcentaje del 50 %-60 % de probabilidad de eliminar una neurona durante el entrenamiento.

Los resultados tras esta modificación son:



(a) Evolución de la función de pérdida. (b) Evolución del accuracy.

Figura 29: Evolución del entrenamiento del modelo utilizando optimizador Adam + Droput más severo.

Gracias al Dropout, no solo mantenemos buenos resultados que han mejorado con respecto al Ejercicio 2, sino que además ha desaparecido el sobreajuste, obteniendo unos resultados en validación muy cercanos al 1.0 en pérdida y prácticamente de un 75 % de precisión. Dicho esto, se procede a evaluar el desempeño con el conjunto de test:

```
TEST ------ Perdida:  0.8475
TEST ------ Accuracy:  0.7491
```

Se puede concluir que hemos mejorado notablemente el modelo, visualizándose esta mejora también para los resultados de prueba, llegando al 75 % de accuracy en el resultado final.

Referencias

[1] Conv2D. *Keras Conv2D and Convolutional Layers.*
https://www.pyimagesearch.com/2018/12/31/
keras-conv2d-and-convolutional-layers/

[2] ReLu. *Layer activation functions.*
https://machinelearningmastery.com/rectified-linear-activation-\
function-for-deep-learning-neural-networks/

[3] MaxPooling2D. *MaxPooling2D layer.*
https://keras.io/api/layers/pooling_layers/max_pooling2d/

[4] Dense. *Dense layer.*
https://keras.io/api/layers/core_layers/dense/

[5] Flatten. *Flatten layer.*
https://keras.io/api/layers/reshaping_layers/flatten/

[6] Softmax. *Softmax classifier.*
https://cs231n.github.io/linear-classify/#svm-vs-softmax

[7] SGD. *SGD optimizer.*
https://keras.io/api/optimizers/sgd/

[8] Batch size. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. Nitish S.K, Dheevatsa M, Jorge N., Mikhail S., Ping T.P.T*
https://arxiv.org/pdf/1609.04836.pdf

[9] ImageDataGenerator. *Preprocessing image with ImageDataGenerator.*
https://www.tensorflow.org/api_docs/python/tf/keras/
preprocessing/image/ImageDataGenerator

[10] VGGNet. *Very Deep Convolutional Networks for Large-Scale Image Recognition.Karen Simonyan, Andrew Zisserman.*
https://arxiv.org/pdf/1409.1556.pdf

[11] Numbers Power of 2. *Why is it recommended to use powers of 2 for no. of filters?*
https://www.kaggle.com/questions-and-answers/177793

[12] Dropout. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Nitish S., Geoffrey H., Alex K., Ilya S., Ruslan S.*
https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

[13] BatchNormalization. *BatchNormalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Sergey Ioffe, Christian Szegedy.*
https://arxiv.org/pdf/1502.03167.pdf

[14] Caltech-UCSD. *Caltech-UCSD Birds 200.*
http://www.vision.caltech.edu/visipedia/CUB-200.html

[15] ResNet50. *ResNet Keras documentation.*
https://keras.io/api/applications/resnet/#resnet50-function

[16] ResNet50 Paper. *Deep Residual Learning for Image Recognition. Kaiming H., Xiangyo Z, Shaoqing R., Jian S.*
https://arxiv.org/pdf/1512.03385.pdf

[17] Fit method. *ImageDataGenerator fit method.*
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

[18] Fine-tune. *Transfer learning and fine-tuning.*
https://www.tensorflow.org/tutorials/images/transfer_learning

[19] Deep Regression. *A comprehensive Analysis of Deep Regression. Stéphane L., Pablo M., Xavier A., Member IEEE and Radu H.*
https://arxiv.org/pdf/1803.08450.pdf

[20] AdaGrad. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. John D., Elad H., Yoram S.*
https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

[21] RMSProp. *Rmsprop Divide the gradient by a running average of its recent magnitude.Tijmen T., Geoffrey H.*
https://www.youtube.com/watch?v=AM9c7zN2KwU

[22] AdaDelta. *Adadelta: An adaptive learning rate method. Matthew D. Zeiler.*
https://arxiv.org/pdf/1212.5701.pdf

[23] Adam. *Adam: A method for stochastic optimization. Diederik P., Jimmy L.*
https://arxiv.org/pdf/1412.6980.pdf