VISIÓN POR COMPUTADOR

PRÁCTICAS - PRÁCTICA 2

Pedro Gallego López

Doble Grado de Ingeniería Informática y Matemáticas

Universidad de Granada
Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

28 de noviembre de 2020

Práctica 2

Introducción

Anotaciones

- Se han creado dos ficheros de python distintos conforme a las plantillas, uno para el apartado 1 y 2 y otro para el apartado 3.
- Durante toda la memoria se exponen gráficas de pérdida (situada normalmente a la izquierda a no ser que se indique lo contrario) y la de accuracy (situado a la derecha a no ser que se indique lo contrario).
- En el apartado 3 se han seguido las indicaciones que se daban dentro de la plantilla para los datos de entrenamiento y test.
- En el apartado 3 hay una variable llamada PATH que indica el PATH de los datos. Habrá que modificar la variable dependiendo de donde se ejecute el programa: en ambos caso estarán dentro de una carpeta llaada 'imagenes'
- Se indicará en su oportuno momento pero se advierte que el apartado del Bonus se sitúa en el mismo apartado 2.
- Todo el código ha sido ejecutado en Google colab ya que mi ordenador no era capaz de ejecutar gran parte del código (en concreto el apartado 3).
- Para ejecutar el apartado 3 al final del código hay 4 líneas que ejecutan todo el programa. Al ejecutarlas juntas la RAM de colab llega al límite y se reinicia a mitad: recomiendo que para probar el apartado 3.1.A se comenten las dos últimas líneas. Para el apartado 3.1.B se comente la segunda y la última línea. Para el apartado 3.2 se comenten las tres primeras líneas.

Apartado 1. BaseNet en CIFRAR100

Capas

Para definirlo he creado una función que nos lo devuelve. En esta función creamos a partir de `keras.Sequential()` el modelo.

- La primera capa, que es de convolución, hay que pasarle además el `input_shape` de la imagen, por ser precisamente la primera.
- Usamos `Flatten` para agrupar todo el tensor en un único vector y así poder hacer Linear.

Optimizador

Para este apartado he decidido usar el optimizador **SGD**, ya que viene hasta importado:

- el **learning rate** a 0.01: dandole valores hasta de 0.1 para abajo (hasta 0.01) he observado que se iba mejorando el accuracy medio (ejecutando varias veces para hacer una estimación media) conforme descendía hasta 0.01; viendo que en 0.001 salían peores resultados, he tirado ahora de manera ascendente hasta 0.01 y he visto que más o menos al acercarse al 0.01 tanto por la izquierda como por la derecha me dan valores similares y de los mejores, es por ello que he decidido escoger el 0.01.
- El **decay**, factor por el que se multiplican los pesos para que no crezcan muy rápido y se puedan ver como un gradiente descendente, he decidido poner un número pequeño, en este caso 1e-6.
- Con el **momentum** he visto que poniendo 0.5 el accuracy bajaba fuertemente (0.3 aproximadamente), en 1 veía inestabilidad durante el proceso en la gráfica, así dejé el 0.9 como valor.

Compilador

Como función de pérdida he seleccionado `categorical_crossentropy`. Es una función de pérdida que funciona bien para los problemas de clasificación como el nuestro ya que es una función de pérdida logarítmica multiclase y mide la calidad de la clasificación.

Entrenamiento

He utilizado la función `.fit()`. Para escoger los datos de entrenamiento y validación he usado la clase `ImageDataGenerator` indicando al parámetro `validation_split` que su valor es 0.1. Para seleccionar el número de épocas óptimas he decidido emplear un número grande, 75 épocas.

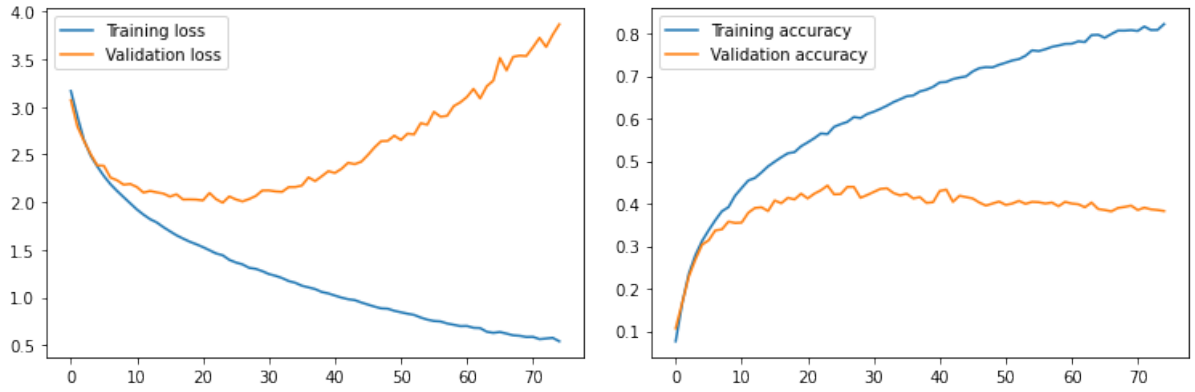

Figura 1.1: 75 épocas

Se observa como los valores de la validación empiezan a separarse de los del entrenamiento, se empieza a observar *overfitting*.

En las gráficas se observa perfectamente como:

- Las pérdidas de la validación alcanzan, en este caso, su mínimo en la época 27, a partir de ahí empiezan a subir.
- El accuracy de la validación alcanza su máximo en la época 27. Se observa que se mantiene más o menos constante, disminuyendo un poco, a partir de esta época.

Para poder coger lo mejor de las dos, he decidido dejar el número de épocas en 27: alcanzamos unas pérdidas mínimas en validación y un accuracy máximo.

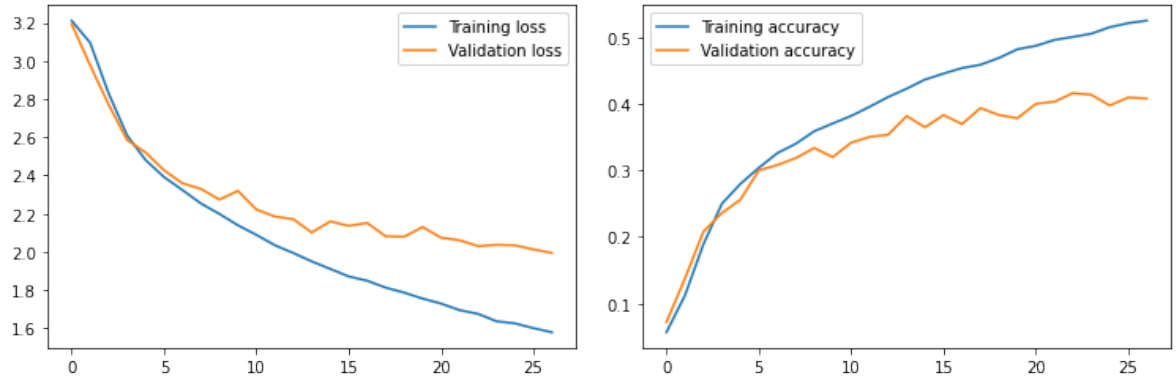

Figura 1.2: 27 épocas. En su entrenamiento hemos conseguido una media de accuracy de 0.53536. Este modelo nos arroja una evaluación de los test de: loss: 1.9901; accuracy: 0.4412

Apartado 2. Mejora del Modelo BaseNet

Normalización de datos

Buscamos normalizar los conjuntos de datos con una media 0 y varianza 1. Creo la instancia de la clase con los parámetros:

1. `featurewise_center = True`
2. `featurewise_std_normalization = True`
3. `validation_split = 0.1`

los dos primeros consiguen que tengamos una media de 0 y una varianza de 1. Además en este apartado, para poder aplicar los cambios a las futuras muestras de entrenamiento y test llamamos a `datagen.fit(x_train)` y a `datagen.standardize(x_test)`. Con `.fit()` aplica todos los cambios definidos en la clase, y con `.standardize()` conseguimos estandarizar con la varianza y la media.

Para **definir los conjuntos** de entrenamiento y de validación he usado el método `.flow()` de `ImageDataGenerator` con un *batch size* de 64. El *batch size* define el número de muestras que van a mandarse a través de la red antes de su actualización, por recomendación se suele poner a 32, 64 ó 128.

Llegados al punto del **entrenamiento**, tenemos que usar ciertos parámetros al usar la clase `ImageDataGenerator`: los parámetros `steps_per_epoch` y `validation_steps` nos indican el número total de los datos de entrenamiento en los que nos lo divide el batch size, cada uno lo contará como una nueva época. Sus valores se calculan como sigue:

$$steps\_per\_epoch = \frac{len(x\_test) \times (1 - SPLIT)}{BATCH\_SIZE}$$

$$validation\_steps = \frac{len(x\_test) \times SPLIT}{BATCH\_SIZE}$$

Ejecutando el modelo sin mejora y con mejora obtenemos:

	modelo sin mejorar	modelo con normalización de datos
loss	1.9572	2.1888
accuracy	0.452	0.4356

Se observa que los datos llegan a ser algo peores incluso. Siempre viene bien que nuestros datos estén normalizados ya que por lo general aumenta el aprendizaje y lleva a una convergencia más rápida.

Aumento de datos

La técnica de aumento de datos consiste en aumentar la cantidad de datos de entrenamiento a partir de los que nos dan. Esto se consigue aplicando sobre los datos iniciales de entrenamiento ciertas operaciones que perturben sus valores, estas perturbaciones pueden ser zoom, giros, simetrías, desplazamientos...

En la práctica lo especificamos en la definición del `ImageDataGenerator`. He añadido:

- `width_shift_range = 0.05`: nos da un desplazamiento horizontal
- `height_shift_range = 0.1`: nos da un desplazamiento vertical

- `horizontal_flip = True`: nos voltea la imagen de manera horizontal. De manera vertical he probado empíricamente que no favorece al rendimiento así que he decidido no ponerlo.
- `rotation_range` no ha arrojado buenos resultados, he decidido no incorporarlo.
- `zoom_range = 0.2`

Con estos valores, ejecutando el modelo sin mejorar y el modelo mejorado, obtenemos los resultados siguientes:

	modelo sin mejorar	modelo con las dos mejoras
loss	2.0658	1.6808
accuracy	0.4288	0.5

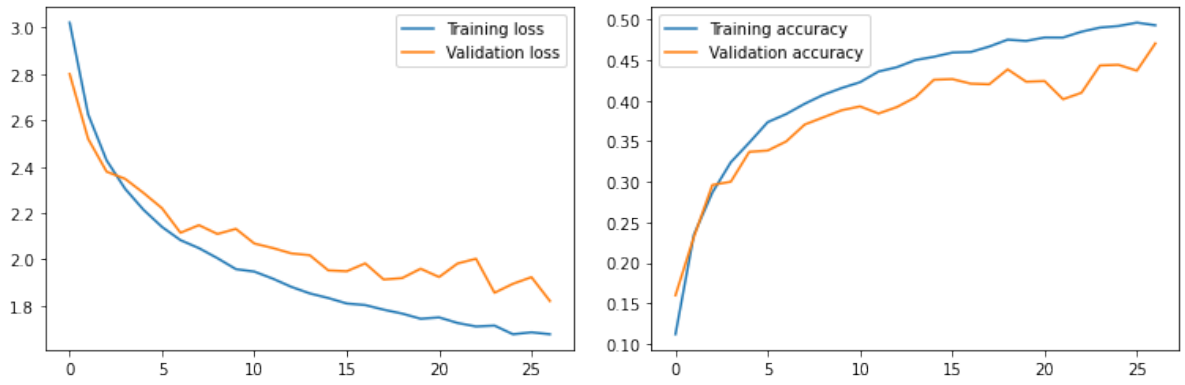

Figura 1.3: 27 épocas

Se empiezan a notar mejorías sustanciales. El *overfitting* empieza a disminuir y se empiezan a igualar ambas gráficas.

Early Stopping

Con esta técnica voy a conseguir que mi modelo logre identificar los pesos asignados en mitad del entrenamiento que "*maximicen*" el rendimiento de mi modelo. En mi caso le he asignado una espera de hasta 20 épocas para ver si, desde una cierta época con unos pesos determinados, se consigue en esas 20 épocas siguientes que el modelo mejore resultados de accuracy y de loss, de no ser así se queda con esos pesos.

Batchnormalization, Dropout, Aumento de Profundidad

He reunido los tres en un apartado porque he decidido hacerlo de forma prácticamente conjunta.

Nos centramos en agregar capas convolucionales al modelo. Se nos avisa del peligro de añadir una capa de *MaxPooling* siempre después de una convolucional por la posible pérdida de información.

El procedimiento que he usado para mejorar el rendimiento de la red, ha sido tanto añadir capas como modificar lo que había. Mencionaré el *Dropout* y el *BatchNormalization* por separado y luego hablaremos de como hemos tratado la red en general, con sus nuevas capas.

- *BatchNormalization*: Estas capas las añado después de cada capa convolucional y totalmente conecada con el fin de que todas las salidas salgan normalizadas con media 0 y varianza 1. Aplicando lo dicho he obtenido los resultados del primer par de imágenes siguientes.

- *Dropout*: Estas capas las he añadido en primera instancia después de cada MaxPooling y antes del último dense. Con esta capa conseguimos disminuir el overfitting que teníamos. Aplicándo las capas dropout de 0.1-0.1-0.25 he obtenido los resultados del segundo par de imágenes siguientes.

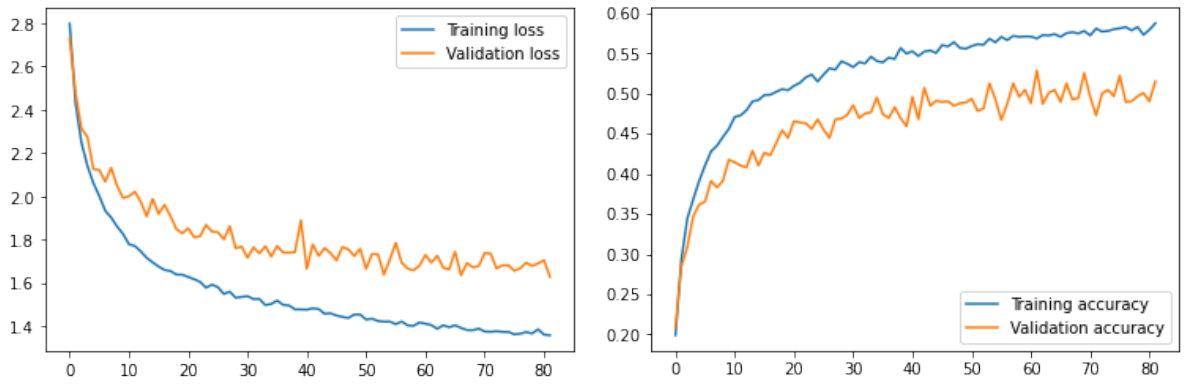

Figura 1.4: Mejora con Batch Normalization. 80 épocas. accuracy 0.5552 y loss 1.5094

	modelo sin mejorar	modelo mejorado con BatchNormalization
loss	2.0658	1.5094
accuracy	0.4288	0.5552

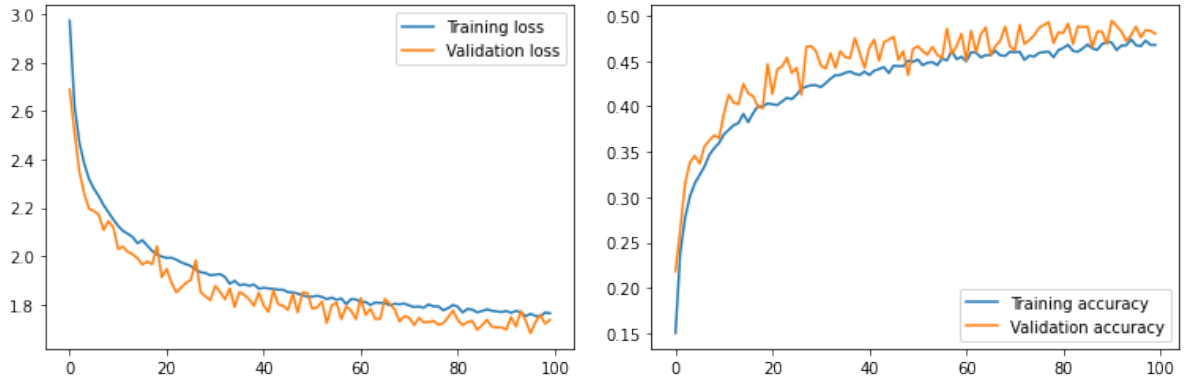

Figura 1.5: Mejora con Batch Normalization y Dropout. 80 épocas. accuracy 0.5364 y loss 1.5421

	modelo sin mejorar	modelo mejorado Dropout y BNormalization
loss	2.0658	1.5421
accuracy	0.4288	0.5364

Se observa como aparte de haber una ganancia el *overfitting* desaparece.

Para mejorar el rendimiento de la red, he decidido aplicar en primer lugar cosas aprendidas en teoría como que una capa de convolución con un kernel de tamaño 5 se puede hacer con dos capas de convolucion con kernel de tamaño 3 ahorrando así operaciones de cómputo.

He decidido también probar qué tal funcionaba el *AveragePooling* en comparación con el *MaxPooling*. En las múltiples pruebas que he realizado el cambio ha sido casi mínimo. Es por ello que no he guardado resultados, pero aún siendo el cambio casi mínimo, el *AveragePooling* salía vencedor. Con ello y con que le veo más sentido a la técnica de *AveragePooling* que a

la de *MaxPooling* he decidido optar a poner en la red la primera. El motivo por el que le veo más sentido es porque consigues información de todas las celdas y juegas con más información, porque ruido puedes tener siempre.

Además he decidido probar con darle profundidad a la red, ya que es un factor clave en el rendimiento de esta, así he ido desarrollando el siguiente diseño provisional.

Diseño provisional

Esta solucion provisional se ha basado en usar un "*bloque*" casi similar de capas de forma repetida:

$$Convolucion-BatchNormalization-Convolucion-BatchNormalization-Pooling-Dropout$$

Se ha repetido el bloque 4 veces. No he aplicado exactamente el mimo bloque ya que al principio hago una convolución suelta para ganar una profundidad de 32 y empezar a operar desde ahí. En otro *bloque* no hago AveragePooling para no empequeñecer tan rápido la imagen y poder seguir teniendo más información.

Ha influido también que a la vez que se le quitaba dimensión (ancho y alto) a los datos, le he ido dando profundidad. En concreto la profundidad dada viene descrita por esta gráfica (las abcisas marcan la capa de convolución):

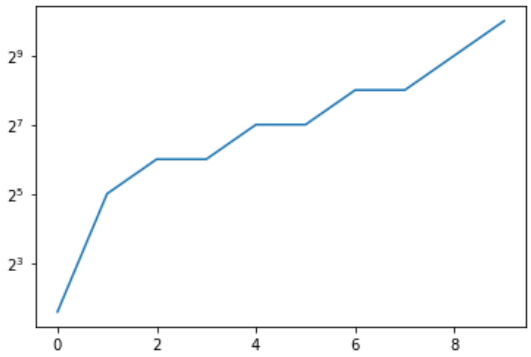

Figura 1.6: Escala logarítmica de la evolución de pronfudidad del tensor a través de la red.

La idea durante todo el proceso ha sido ganar profundidad conforme perdíamos dimensión. Esto es porque empíricamente he visto que los resultados mejoraban siguiendo esta idea. De la misma manera, el hecho de que en los tamaños 64, 128, 256 de profundidad los mantengamos durante dos convoluciones tiene que ver también con que empíricamente he probado que uno más no mejoraba el resultado y uno menos tampoco. En los demás tamaños no he puesto dos convoluciones porque cuando la dimensión de la imagen era grande no conseguía sacar buenos resultados, y cuando la dimensión de la imagen era muy pequeña, aguantar con una profundidad de 1024, por ejemplo, era muy costoso en cuanto a tiempo.

Con este último diseño una época tarda en ejecutarse unos 14 segundos. Un tiempo más alto que el modelo de partida, pero que he considerado bueno para poder lograr los resultados con los tests que he logrado: un accuracy de 0.79, casi duplicando al modelo BaseNet, explorando durante 76 épocas los mejores pesos con un *patience* de 20 épocas. El modelo sale definido en la tabla de capas siguiente.

Tabla de capas del diseño provisional

Layer No.	Layer Type	Kernel size	Input-Output Dimension	Input-Output Channels
1	Conv2D	3	32-32	3-32
2	BatchNorm	-	32-32	-
3	Relu	-	32-32	-
4	Conv2D	3	32-30	32-64
5	BatchNorm	-	30-30	-
6	Relu	-	30-30	-
7	Conv2D	3	30-28	64-64
8	BatchNorm	-	28-28	-
9	Relu	-	28-28	-
10	AveragePool	2	28-14	-
11	Dropout 20	-	14-14	-
12	Conv2D	3	14-14	64-128
13	BatchNorm	-	14-14	-
14	Relu	-	14-14	-
15	Conv2D	3	14-12	128-128
16	BatchNorm	-	12-12	-
17	Relu	-	12-12	-
18	Dropout 20	-	12-12	-
19	Conv2D	3	12-12	128-256
20	BatchNorm	-	12-12	-
21	Relu	-	12-12	-
22	Conv2D	3	12-12	256-256
23	BatchNorm	-	12-12	-
24	Relu	-	12-12	-
25	AveragePool	2	12-6	-
26	Dropout 20	-	6-6	-
27	Conv2D	3	6-6	256-512
28	BatchNorm	-	6-6	-
29	Relu	-	6-6	-
30	Conv2D	3	6-6	512-1024
31	BatchNorm	-	6-6	-
32	Relu	-	6-6	-
33	AveragePool	2	6-3	-
34	Dropout 20	-	3-3	-
35	Dense	-	9216-512	-
36	BatchNorm	-	512-512	-
37	Relu	-	512-512	-
38	Dense	-	512-256	-
39	BatchNorm	-	256-256	-
40	Relu	-	256-256	-
41	Dropout 50	-	256-256	-
42	Dense	-	256-25	-

En las gráficas siguientes, fruto de una de las ejecuciones de este modelo, se puede observar *overfitting*. Pese al gran resultado que ha dado el modelo he decidido trabajar en él para arreglar un poco el *overfitting*.

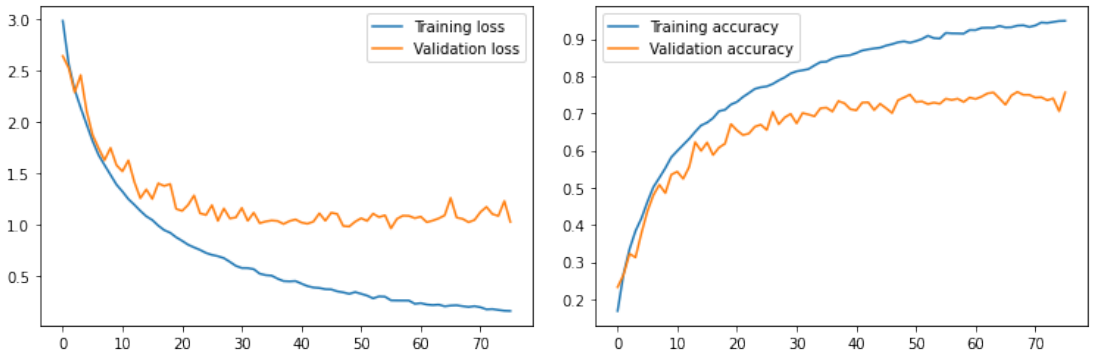

Figura 1.7: 76 épocas. loss: 0.8293 - accuracy: 0.7912

Diseño final

El objetivo que perseguía ahora era reducir el overfitting, para ello decidí aumentar ligeramente los Dropout a 0.35 y en dos ocasiones a 0.5 y a 0.7. Para cambiar estos valores fui subiendo poco a poco viendo como respondía el modelo hasta llegar a estos valores que me convencieron.

Ya que estaba modificando el modelo, quise quitarle tiempo de cómputo, pues tenía capas con muchísima profundidad (hasta 1024) y esto provocaba que la duración media de la época fuese de unos 14 segundos. Para ello lo que hice fue en primer lugar quitar el "bloque" convolucional último, que era el más profundo y bajar la profundidad de aquellas convoluciones que estaban por encima del 256 (dejando este como máximo). Aplicando estos retoques se me quedó el modelo de la tabla que vendrá más adelante titulada como *"Tabla de capas del diseño final"*. Ahora muestro las gráficas obtenidas de *loss* y *accuracy* con este nuevo modelo.

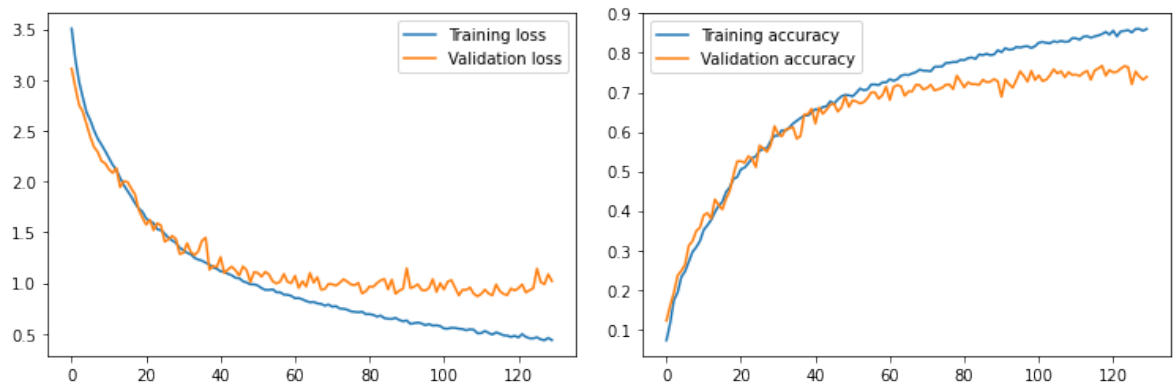

Figura 1.8: Épocas 129. loss: 0.7305 - accuracy: 0.7932

A simple vista se ve como ha mejorado el *overfitting*: prácticamente hasta la época 50 van de la mano validación y test. A partir de la época 50 se sigue viendo un ligero sobreajuste a los datos de entrenamiento, pero mucho menor que el modelo anterior.

Además, se ha conseguido superar el *accuracy* en una de las ejecuciones y no solo eso, sino que se ha conseguido bajar la marca de la función de pérdida.

Se ha conseguido también rebajar el tiempo de la época de unos 14segundos a unos 7 segundos, algo nada despreciable. Consecuentemente, al perder profundidad y neuronas (lo hemos sacrificado por tiempo) hemos aumentado el número de épocas necesarias: se han ejecutado 130 épocas. A efectos prácticos, si hacemos el cálculo con estos datos, han tardado lo mismo cada modelo en entrenarse, luego no hemos conseguido una mejora como tal, simplemente, si se quiere, unos datos más desglosados al haber más épocas en el mismo tiempo.

Tabla de capas del diseño final

Layer No.	Layer Type	Kernel size	Input-Output Dimension	Input-Output Channels
1	Conv2D	3	32-32	3-32
2	BatchNorm	-	32-32	-
3	Relu	-	32-32	-
4	Conv2D	3	32-30	32-64
5	BatchNorm	-	30-30	-
6	Relu	-	30-30	-
7	Conv2D	3	30-28	64-64
8	BatchNorm	-	28-28	-
9	Relu	-	28-28	-
10	AveragePool	2	28-14	-
11	Dropout 35	-	14-14	-
12	Conv2D	3	14-14	64-128
13	BatchNorm	-	14-14	-
14	Relu	-	14-14	-
15	Conv2D	3	14-12	128-128
16	BatchNorm	-	12-12	-
17	Relu	-	12-12	-
18	Dropout 35	-	12-12	-
19	Conv2D	3	12-12	128-256
20	BatchNorm	-	12-12	-
21	Relu	-	12-12	-
22	Conv2D	3	12-12	256-256
23	BatchNorm	-	12-12	-
24	Relu	-	12-12	-
25	AveragePool	2	12-6	-
26	Dropout 35	-	6-6	-
27	Conv2D	3	6-6	256-256
28	BatchNorm	-	6-6	-
29	Relu	-	6-6	-
30	AveragePool	2	6-3	-
31	Dropout 55	-	3-3	-
32	Dense	-	2304-512	-
33	BatchNorm	-	512-512	-
34	Relu	-	512-512	-
35	Dense	-	512-256	-
36	BatchNorm	-	256-256	-
37	Relu	-	256-256	-
38	Dropout 70	-	256-256	-
39	Dense	-	256-25	-

Y hasta aquí sería el modelo que he presentado en mi práctica. Lo que resta de apartado 2 explicaré qué he seguido haciendo, pues mi trabajo no se quedó aquí unque no haya conseguido nada para poder presentarlo en la práctica, y creo que el tiempo dedicado y lo aprendido en ese tiempo he de plasmarlo en la memoria.

Mi intención era lograr la ambiciosa cifra del 0.8 de forma estable. Es decir, que todas las ejecuciones rondasen la cifra 0.8 ya sea por la alta o por la baja. Para ello decidí informarme de técnicas o de algún *"truco"*, pese a que sé que el diseño de redes convolucionales y más a mi nivel se hace a partir de prueba y error, para poder aplicarlo a mi ejercicio a ver si daba

resultado. Aprendí cosas interesantes que no he logrado que se noten en la práctica.

- Usar convoluciones 1×1 para simplificar cálculos: Si nosotros tenemos un tensor de tamaño $32 \times 32 \times 256$ y queremos aplicarle una convolución de kernel 3×3 de 128 unidades, es decir, nos generará un vector (si se conserva el tamaño de la imagen) de $32 \times 32 \times 128$. Para ello se requiere de

$$32 \times 32 \times 128 \times 3 \times 3 \times 256 \approx 300 \; Millones \; de \; operaciones$$

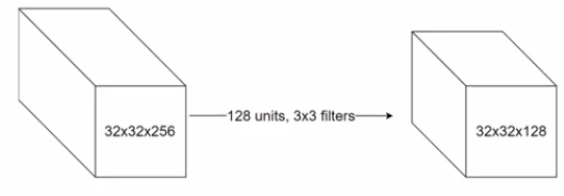

Figura 1.9

Si decidimos usar una capa intermedia de convolución con un kernel de 1×1 y que genere un tensor de profundidad menor, 64 por ejemplo, y luego al tensor resultante le aplicas la capa de convolución que hemos dicho antes para generar el tensor de $32 \times 32 \times 128$ necesitariamos muchísimas menos operaciones:

$$32 \times 32 \times 64 \times 1 \times 1 \times 256 + 32 \times 32 \times 128 \times 3 \times 3 \times 64 \approx 92 \; Millones \; de \; operaciones$$

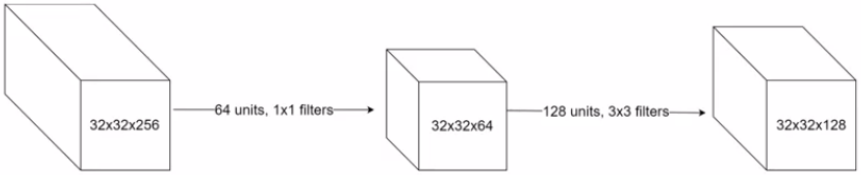

Figura 1.10

Las operaciones se calculan como el número de valores del tensor a calcular por el tamaño del filtro (contando la profundidad).

Esto no lo he aplicado a mi modelo porque el hecho de tener tan poca profundidad hace que la parte de la red a partir del `Flatten` ocupe una gran parte del tiempo computacional, haciendo que no se note esta mejora, ya que solo puede ir entre capas convolucionales. Al añadir profundidad hacemos además que haya más pesos que aprender, lo que provoca que el modelo tarde más en ir cogiendo buenos resultados durante el entrenamiento. Con todos estos impedimentos decidí no implementarlo.

- A pesar de usar `AveragePooling2D`, he visto que se recomienda mucho el uso de `MaxPooling2D` puesto que suele comportarse bien. Bajo mi experiencia he decidido poner `AveragePooling2D`, pero no he dejado de valorar durante todas las pruebas el poner `MaxPooling2D` para ver si se conseguía alguna mejora. La realidad es que no afectaba apenas a los resultados de mi modelo, así que decidí dejarlo como estaba.

No conseguí mejorar la marca del último diseño tras diversas pruebas y cambios en el modelo, así que decidí dejarlo. Con esto ya se da por finalizado el estudio y resolución del apartado 2 junto con el Bonus, que se puede considerar como realizado después de este ejercicio.

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

Recomiendo este apartado ejecutarlo por partes como viene indicado en la introducción. En este apartado se nos pide usar la red **ResNet50** y el conjunto de datos **Caltech-UCSD** (6033 imágenes de 200 especies de pájaros, es decir, 200 clases). Se tendrán 3000 imágenes de entrenamiento y 3033 de test y se dejará un 10 % para la validación. Utilizaremos el modelo **Resnet50** ya preentrenado. Se divide en dos subapartados:

Subapartado 1. Usar ResNet como extractor de características

Lo que nos pide esta parte de la práctica es eliminar las capas FC y salida del modelo **ResNet50** para así, en vez de tener las imágenes clasificadas, lo que tenemos son datos previos a la clasificación que los veremos como *características*.

La idea de este subapartado es utilizar estas características como entrada a otro modelo "sencillo" que crearemos nosotros. Los pasos que vamos a tener que seguir son los siguientes:

1. Cargar los datos de Caltech-UCSD.
2. Instanciamos ResNet50 sin la activación 'softmax' de forma que nos de un vector (ahorrándonos el Flatten).
3. Creamos dos generadores mediante datagen para el entrenamiento y el test.
4. Con los dos generadores creados en el paso anterior generamos las características de entrenamiento y de test
5. Entrenamos el 'modelo sencillo' que hemos creado para clasificar:
   a) Instanciamos el compilador
   b) Compilamos el modelo
   c) Entrenamos el modelo
6. Evaluamos con las características test el 'modelo sencillo' que hemos creado.

Explicada la idea tenemos que pensar ya en generar estos datos, en preprocesarlos a través de *datagen*. Para ello en keras se dispone una función de preprocesado para según que red. En nuestro caso, tenemos que importar

`keras.applications.resnet.preprocess_input`

Así le indicamos al datagen la forma de preparar nuestro generador. Vamos a definir un modelo básico con una sola capa Dense con activación 'softmax' (nos referiremos a él como *modelFC-Basic*) y un módelo con alguna capa más FC que llamaremos *modelFC*.

Tabla de capas de *modelFC*

Layer No.	Layer Type	Kernel size	Input-Output Dimension	Input-Output Channels
1	Dense	-	100352-1024	-
2	Relu	-	1024-1024	-
3	Dense	-	1024-512	-
4	Relu	-	512-512	-
5	Dropout 70	-	512-512	-
6	Dense	-	512-200	-

Se obtienen los siguientes resultados:

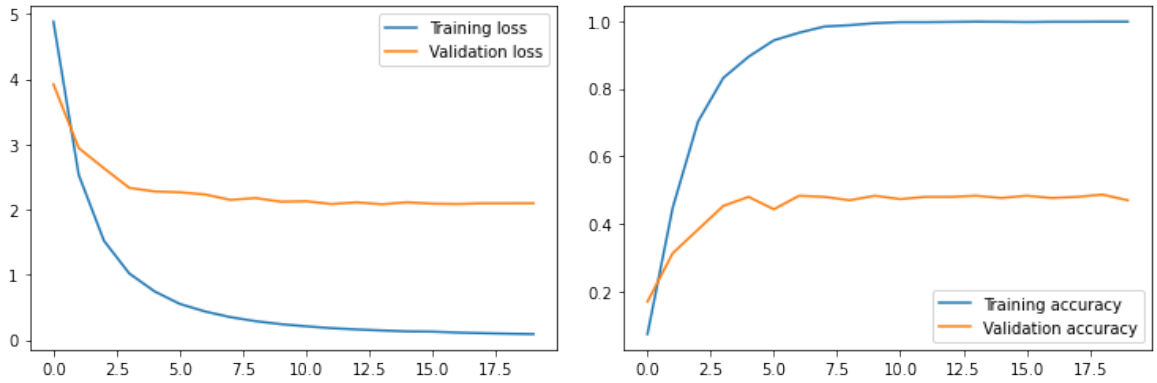

Figura 1.11: Modelo básico de una capa Dense. 20 épocas. loss: 2.3410 - accuracy: 0.4352

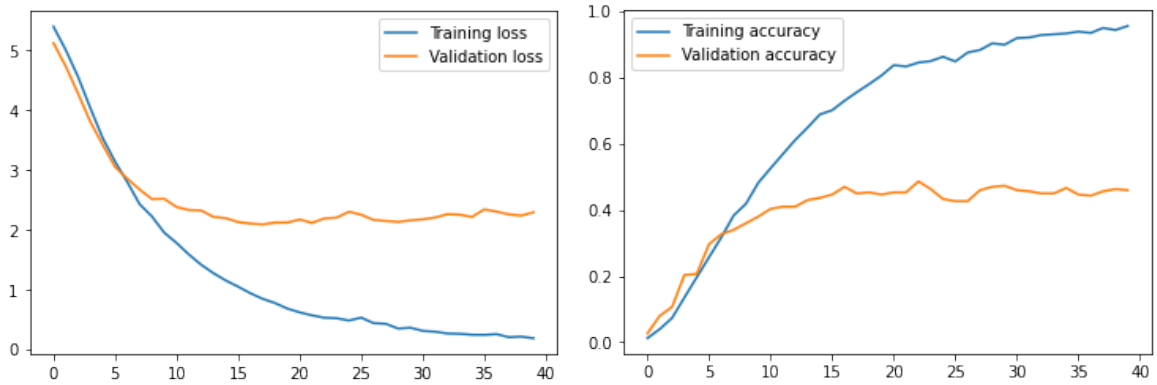

Figura 1.12: Modelo con varias capas FC. 40 épocas. loss: 2.4658 - accuracy: 0.43455

Se ha observado que ambos modelos dan resultados muy parecidos. Es más, en el modelo básico se puede ver como con muy pocas épocas logra estabilizar su accuracy. Sin embargo, el modelo con varias capas FC le cuesta más estabilizarse, tarda más. Es más, se observa como en el modelo básico se obtiene mucho *overfitting* por lo que parece no tener mucho sentido añadir más capas FC.

El siguiente paso es crear un modelo con capas convolucionales, que va a tener un esquema conforme a la siguiente tabla de capas:

Tabla de capas de *modelConv*

Layer No.	Layer Type	Kernel size	Input-Output Dimension	Input-Output Channels
1	Conv	3	7-5	2048-1024
2	BatchNorm	-	5-5	-
3	Conv	3	5-3	1024-1024
4	BatchNorm	-	3-3	-
5	Dropout 50	-	3-3	-
6	Dense	-	9216-1024	-
7	Relu	-	1024-1024	-
8	Dropout 70	-	1024-1024	-
9	Dense	-	1024-200	-

Para este modelo hay que tener en cuenta que tenemos que eliminar también la capa de *avg* para poder introducir capas convolucionales.

Nos ha arrojado los siguientes resultados:

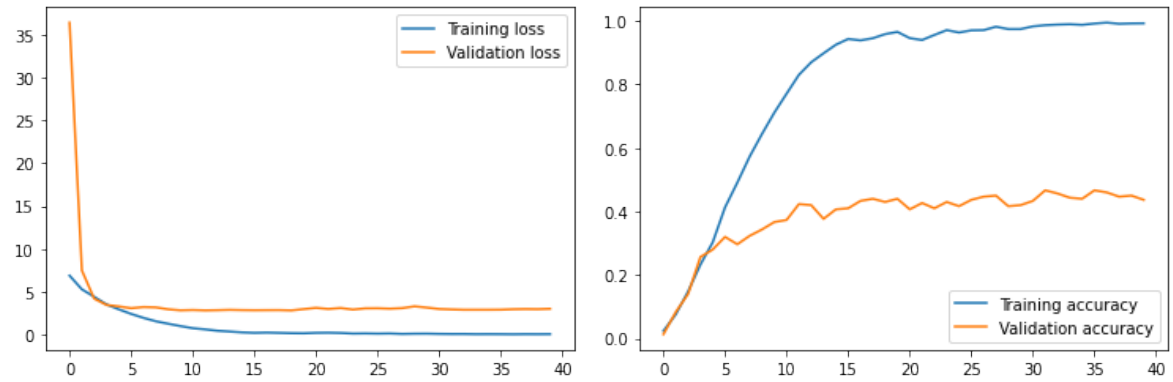

Figura 1.13: Modelo con varias capas FC. 40 épocas. loss: 3.2243 - accuracy: 0.3956

Nos da peores resultados que los anteriores, teniendo más cómputo por detrás. Además nos muestra un *overfitting* muy fuerte. Este modelo sale perdiendo frente a los anteriores.

Subapartado 2. Ajuste fino de toda la red ResNet 50 (Fine Tunning)

En este apartado lo que se nos pide es, a diferencia del subapartado anterior, considerar los "dos" modelos como uno solo. Es decir, quitándole a ResNet la capa FC y salida y agregando nosotros las nuestras para nuestra problema, generaremos un modelo que deberemos entrenar entero. Como la red viene preentrenada, nosotros necesitaremos "poco" trabajo para entrenarla, puesto que simplemente deberemos de ajustar un poco los pesos para nuestro problema.

En vista a los resultados del subapartado anterior, vamos a añadir el *modelFC* para que tenga alguna capa de más y no sea muy básico. Se nos recuerda que tenemos que ejecutar un número pequeño de épocas, pondremos 20.

En lo que difiere la forma de programar este ejercicio del anterior es que ahora estamos con una instancia de *Model* y no de *Sequential*, tendremos que modificar la forma de añadir capas. He obtenido los siguientes resultados:

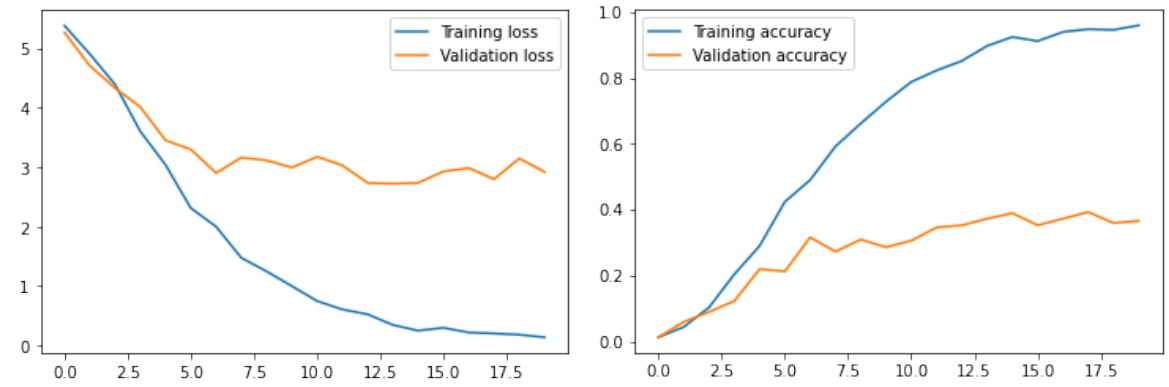

Figura 1.14: 20 épocas. loss: 3.3414 - accuracy: 0.3274

Se puede ver como los resultados son notablemente peores. Puesto que se supone que con esta técnica debería de mejorar algo, no empeorar he decidido simplificar lo añadido: en vez de añadir el *modelFC* he decidido añadir un modelo con las siguientes capas.

Tabla de capas para añdir a ResNet50

Layer No.	Layer Type	Kernel size	Input-Output Dimension	Input-Output Channels
1	Dense	-	100352-2048	-
2	Relu	-	2048-2048	-
3	Dropout 70	-	2048-2048	-
4	Dense	-	2048-200	-

Con esta nueva opción hemos conseguido mejorar los resultados:

Figura 1.15: 20 épocas. loss: 2.6948 - accuracy: 0.4128

Se han conseguido mejorar el accuracy hasta dejarlo relativamente cerca del ejercicio anterior. Hay que tener en cuenta que ha sido entrenado en menos épocas que los apartados enteriores y su accuracy de validación tiene una tendencia alcista que puede desencadenar mejores resultados que el obtenido.

BONUS

El Bonus lo he explicado directamente en el apartado 2. A modo resumen se ha propuesto usar otro tipo de pooling, en concreto `AveragePooling2D`, se han propuesto técnicas para ahorrar operaciones de cómputo,... entre otras muchas cosas.