PRÁCTICA 2: Redes Neuronales Convolucionales

Antonio José Blánquez Pérez

Visión por Computador

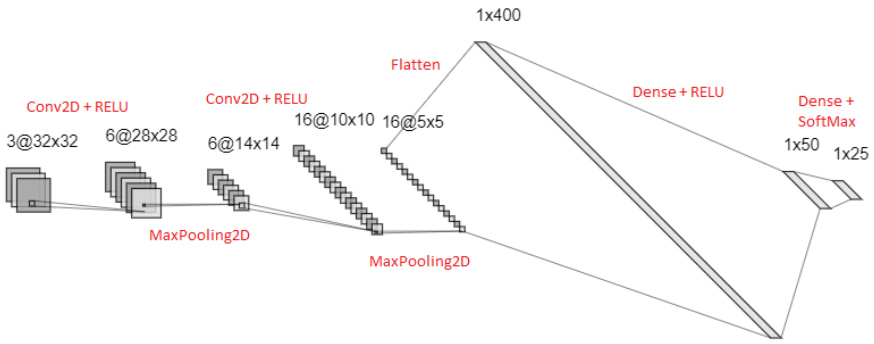
Índice

1. BaseNet en CIFAR100

En este primer apartado se va a implementar la red neuronal BaseNet con Keras, probandola en CIFAR100.

1.1. Implemetación de la red

Lo primero es crear la red, por capas, a partir de Keras. En este caso crearemos un modelo *model* secuencial a partir de $keras.models.Sequencial$, dado que en BaseNet las capas están conectadas de esta manera, secuencialmente, sin ciclos ni saltos. A partir del modelo vamos añadiendo capas con la función $add()$, pasando como parámetro la capa correspondiente con las que proporciona $keras.layers$. Una posible descripción gráfica de BaseNet podría ser la siguiente:



En este caso, siguiendo las indicaciones dadas, las capas se generan en *model* a partir de las siguientes órdenes:

$$model.add(keras.layers.Conv2D(6,(5,5),activation='relu',input\_shape=(32,32,3)))$$
$$model.add(keras.layers.MaxPooling2D((2,2)))$$
$$model.add(keras.layers.Conv2D(16,(5,5),activation='relu'))$$
$$model.add(keras.layers.MaxPooling2D((2,2)))$$
$$model.add(keras.layers.Flatten())$$
$$model.add(keras.layers.Dense(50,activation='relu'))$$
$$model.add(keras.layers.Dense(25))$$

Usaremos $Conv2D$ para las capas convolucionales, pasándole el número de canales que queremos en el output, el tamaño de los filtros que se aprenderán, el tipo de activación que queremos(en nuestro caso RELU, ahorrándonos dicha capa por separado) y las dimensiones de los datos de entrada en el caso de la primera capa, que en nuestro caso son las dimensiones de las imágenes, 32x32, por los 3 canales RGB. $MaxPooling2D$ será en el caso de las capas de Max Pooling, que reducirán dimensionalidad, a los que solo será necesario pasar el tamaño de los kernels. $Flatten$ nos convertirá el tensor en vector unidimensional, no necesita parámetros. Por último $Dense$ se usa para las capas totalmente conectadas, que sólo necesitan el tamaño de la capa del output, es decir, el tamaño de la siguiente capa en la red; además de la especificación de la activación si es

necesaria, que en nuestro caso será RELU sólo en la primera ya que la segunda es la última capa, que tiene un output de tamaño n siendo n el número de clases presente en nuestra base de datos(en nuestro caso $n = 25$), e introduciremos una activación softmax que dará la probabilidad de que el elemento pertenezca a cada clase, como se nos propone en las diapositivas de la práctica, dado que es la que proporciona el output de la red al completo.

Vemos entonces a continuación una comparación de la definición de BaseNet(izquierda) y del resultado de *model.summary*(derecha), que nos mostrará la red que hemos creado con el proceso anterior, viendo que la hemos generado correctamente.

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	-

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 6)         456
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 6)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 16)        2416
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 50)                20050
_________________________________________________________________
dense_1 (Dense)              (None, 25)                1275
=================================================================
Total params: 24,197
Trainable params: 24,197
Non-trainable params: 0
_________________________________________________________________
```

1.2. Análisis de la red

Podemos separar la red en dos partes, siendo la primera la parte convolucional. Este conjunto de capas se encargará de encontrar patrones mediante los filtros aplicados en cada capa, que serán aprendidos en el proceso de entrenamiento para el reconocimiento de las imágenes de nuestra base de datos, es decir, se encarga de extraer información de la imagen. Aunque la operación de convolución por sí sola sólo es capaz de reconocer patrones como bordes o texturas, la composición de este tipo de capas hace que sean capaces de reconocer patrones cada vez más complejos; pensemos que en una convolución condensamos en un solo píxel todo el vecindario de este, por lo que cuando volvamos a acceder a este píxel en la siguiente convolución tendremos, en el mismo espacio, más información ya que está más condensada. Esto lo podemos potenciar incluso más si reducimos incluso más las imágenes entre capa y capa, ya que condensamos aún más la información, y es por ello que se alterna entre capa convolucional y Max Pooling.

La segunda parte es una red neuronal multicapa densa, es decir, contiene todas sus capas totalmente conectadas. Esta parte recoge como input el output de la última capa de la parte convolucional, que ya se supone que contiene imágenes con distintos patrones o características, lo aplana (convierte el conjunto tridimensional ancho×alto×nºimg en un vector unidimensional) y lo introduce en una red neuronal de dos capas totalmente conectadas que se encargará de clasificar las imágenes. Como ya se ha comentado antes, a través de un output de tamaño nº de clases y una activación SoftMax obtendremos la probabilidad de pertenencia de una imagen a cada una de las distintas clases. El resto de capas que no son esta van directamente seguidas de una activación RELU, la cual se usa exhaustivamente en Deep Learning ya que, aunque es simple y compu-

tacionalmente eficiente, es suficiente para enriquecer la red como capa no lineal entre operaciones lineales, permitiendo predecir funciones más complejas.

Comentar que no es necesario usar padding dado que el objetivo, como ya hemos comentado es ir reduciendo dimensionalidad en la imagen, por lo que no tiene sentido introducir elementos externos a la imagen que además disminuyan la reducción, es contraproducente.

1.3. Compilación y entrenamiento del modelo

Antes de compilar es necesario establecer los parámetros del optimizador elegido, que en este caso ha sido Adam. Se ha elegido ya que dicho modelo es el que mejor resultados ha dado para estos modelos de clasificación de imágenes en algunos análisis científicos que se han consultado[1][2]. Adam es una mejora de SGD que varía el learning rate a partir de dos momentos específicos según β_1 y β_2. Los parámetros se han intentado seleccionar por prueba y error guiándose por el error de validación, pero se ha comprobado que son los mismos parámetros por defecto los que mejor funcionan, ya que ya están bastante ajustados.

Como función de pérdida se ha usado $categorical\_crossentropy$ ya que vamos a realizar una clasificación multiclase, mientras que la métrica que vamos a usar es la $accuracy$. Una vez compilado el modelo guardamos los pesos para poder realizar varios entrenamientos y que sean comparables, con vista al siguiente apartado.

A la hora de entrenar vamos a separar primero un 10 % del conjunto de entrenamiento, el cual hemos obtenido junto al de test de la función $cargarImagenes$ dada para ello, en un conjunto de validación, que nos servirá para ver la bondad del modelo obtenido del entrenamiento antes de usarlo en test, con vista también sobre todo al siguiente apartado. Para ello se ha utilizado $ImageDataGenerator$ con $validation_split$, además de para separar en batches durante el entrenamiento. Se ha elegido un tamaño de batch de 64, basandose también la documentación científica consultada antes mencionada, comprobándose que respecto a valores más bajos, como 32, los valores de accuracy de entrenamiento y validación están más cercanos manteniéndose la bondad de la solución, teniendo en cuenta también que no es necesario hacerlo más pequeño ya que la red no es demasiado compleja.

Entonces se entrena el modelo, que se ha comprobado empíricamente que es suficiente con 30 épocas para obtener un buen resultado, ejecutando $model.fit$ con los parámetros pertinentes adaptados a las características ya comentadas.

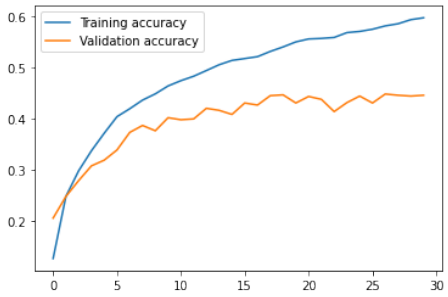
1.4. Resultados

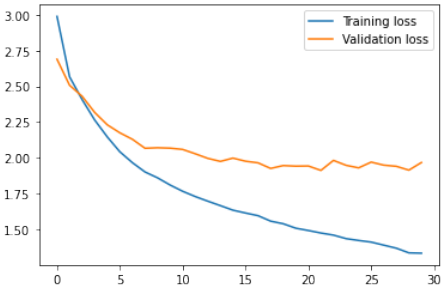
Tras varias ejecuciones, dado que hay componentes estocásticos(denotar que esto puede influir en que los resultados varíen en distintas ejecuciones, lo cual es algo a tener en cuenta), obtenemos unos valores medios aproximados que

[1] A Comprehensive Analysis of Deep Regression, 2020. Stephane Lathuilière, Pablo Mesejo...
[2] Adam: A Method for Stochastic Optimization, 2015. Diederik P. Kingma, Jimmy Ba

aparecerán en la tabla de más adelante.

Podemos observar la evolución de una ejecución en las siguientes gráficas, realizadas con la función *mostrarEvolucion* dada para ello.





Por último evaluamos con *model.evaluate* el conjunto de test en el modelo, podemos ver una comparación de los valores medios obtenidos en la siguiente tabla. Los resultados son decentes para la complejidad de la red y de la base de datos.

	Train	Validation	Test
Loss	1.30	1.95	1.97
Accuracy	0.58	0.44	0.44

Podemos observar unos resultados bastante buenos para lo simple del modelo, aunque mejorables. Sobre todo destacar que hay algo de sobreajuste que podría ser corregido.

2. Mejora del modelo BaseNet

En este apartado se van a introducir distintas mejoras a BaseNet con el objetivo de mejorar la accuracy del modelo para test, guiándonos por el conjunto de validación. Se van a comparar las mejoras con un sólo conjunto de validación en lugar de usando validación cruzada por temas de eficiencia ya que, además, las mejoras deberían implicar un incremento notable de la accuracy por lo que no debería de afectar. En cada uno de los subapartados se comentará en qué consiste cada mejora y como influye a los resultados, comentando los valores medios aproximados a las ejecuciones hechas y adjuntando los gráficos de alguna ejecución en concreto. Las pruebas se irán haciendo de manera secuencial, añadiendo la mejora y arrastrándola con la siguiente si ésta da buenos resultados o desechándola si no parece ser útil para nuestro modelo. Comentar que dado que van a hacerse cambios en los conjuntos de entrenamiento y validación, se van a generar objetos *ImageDataGenerator* separados y se utiliza $train_test_split$ para separar los conjunto en lugar de que lo haga la clase internamente.

2.1. Normalización

La primera opción por la que se ha optado ha sido normalizar los datos, tanto de los datos de entremamiento como los de validación y test, ajustándolos al conjunto de entrenamiento(los de test al conjunto de test), Centrando los

datos ajustando la media a 0 y la desviación típica a 1. Esto se ha hecho usando los parámetros de $featurewise\_center$ y $featurewise_std_normalization$ de los objetos $ImageDataGenerator$, además de la función fit necesaria para que tengan efecto dichos parámetros.

Si volvemos a entrenar el modelo, partiendo de los pesos que guardamos tras compilar, se observa una pequeña mejora en los resultados, se mantendrá la normalización igualmente ya que puede ser útil para posteriores mejoras.

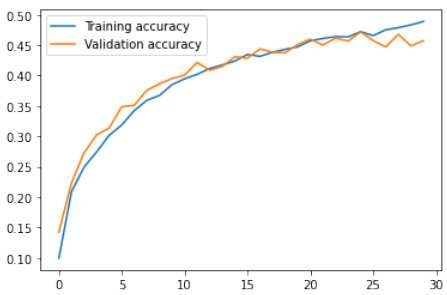
En este caso no tiene sentido adjuntar resultados ya que son prácticamente los mismos a los anteriores.

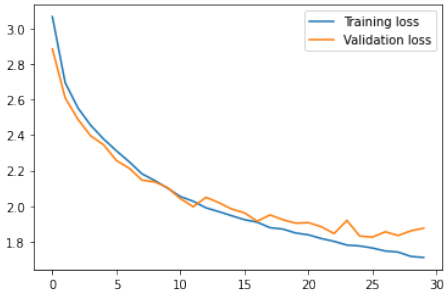
2.2. Data Augmentation

Después se ha añadido data augmentation, en este caso se ha incluído la posibilidad de que las imágenes roten 45 grados antes de entrar a la red, o la de que hagan un flip horizontal. Esto se ha conseguido con los parámetros de $ImageDataGenerator\ rotation_range$ y $horizontal_flip$ respectivamente. La técnica de data augmentation hace que, al variar la perpectiva de las imágenes, se consiga mayor generalidad al entrenar y por tanto se rebaje el overfitting y por tanto loss y accuracy de entrenamiento y validación están mucho más cercanos. Vamos a modificar algunos datos de manera aleatoria y no ha introducir nuevos datos modificados porque de esta manera ya funciona suficientemente bien y así no consumimos más memoria.

En este caso conseguimos los siguientes resultados tras entrenar el modelo, en el formato comentado anteriormente.

	Train	Validation
Loss	1.73	1.87
Accuracy	0.50	0.45





Se puede ver claramente como se ha corregido prácticamente del todo el sobreajuste, por lo que ha resultado una gran mejora.

2.3. Aumento de profundidad

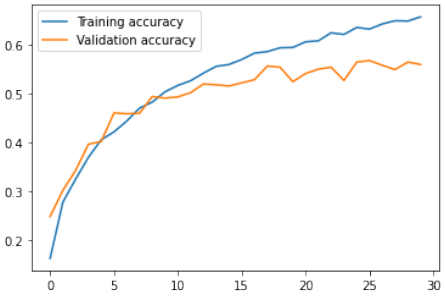
Otra posible mejora es aumentar la profundidad de la parte convolucional. Para este caso la intuición dice que añadiendo más capas mejorará el resultado, sin embargo no es tan sencillo como añadir capas per se. Se han consultado

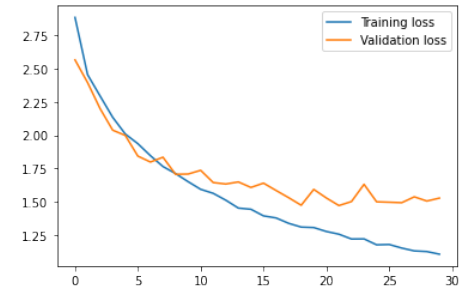
algunos papers al respecto[3][4] y se han sacado algunas conclusiones. La primera es que hay que tener cuidado de dejar demasiado pequeñas las imágenes, ya que con el uso de demasiadas capas convolucionales va empequeñeciendo el tamaño de los filtros hasta que empieza a ser contraproducente; la segunda es que hay que ir reduciendo el tamaño del kernel. Con estas ideas se han añadido algunas capas convolucionales seguidas y MaxPooling solo después de una de cada dos capas, eso sí añadiendo padding a las imágenes para que de esa manera podamos controlar el tamaño de las imágenes. La red ha quedado como se muestra en el siguiente esquema:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 32, 32, 16)        448
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 16)        2320
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 32)        4640
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 32)        9248
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 8, 8, 64)          18496
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_4 (Dense)              (None, 50)                51250
_________________________________________________________________
dense_5 (Dense)              (None, 25)                1275
=================================================================
Total params: 124,605
Trainable params: 124,605
Non-trainable params: 0
_________________________________________________________________
```

Tras entrenar este nuevo modelo con nuestra base de datos obtenemos los siguientes resultados.

	Train	Validation
Loss	1.10	1.50
Accuracy	0.65	0.56





[3]He, Kaiming, et al. "Deep residual learning for image recognition. 2016"
[4]Sun, Shizhao, et al. .On the depth of deep neural networks: A theoretical view. 2016"

Vemos que se ha mejorado bastante el rendimiento, quedando aún algunos flecos, como que hemos aumentado el sobreajuste, que intentaremos corregir con otras mejoras.

2.4. Early Stopping

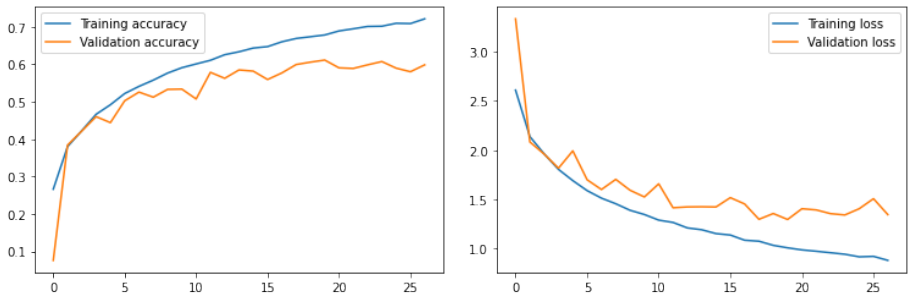
Como la anterior mejora empeora el tiempo de ejecución, se ha decidido que la siguiente sea Early Stopping. Esta función tan solo monitoriza una variable, en nuestro caso val_accuracy ya que es la que nos interesa, y corta la ejecución cuando esta variable no mejore en un número de épocas que nosotros debemos estableces. Early Stopping se debería usar ya que en el peor caso no afecta en nada y en el mejor nos acorta el tiempo de ejecución y, al guardar los mejores pesos, siempre nos iguala o nos mejora el resultado, eso sí, con un valor correcto de paciencia. Requiere ajustar el valor de paciencia, que en nuestro caso se ha establecido en 7 épocas.

En este caso tampoco tiene sentido adjuntar resultados ya que al entrenar al modelo estos son prácticamente los mismos a los anteriores, pero lo mantendremos por las ventajas antes comentadas.

2.5. Batch Normalization

Batch Normalization nace de la idea de que, si normalizamos los datos a la entrada, por qué no hacerlo también para cada capa. No hay gran cosa que destacar sobre esta función, podemos acceder a ella desde $keras.layers.BatchNormalization$ y no requiere ningún comando para esta aplicación. El único hecho de interés es que este funcionalidad se puede usar antes o después de la función de activación, en este caso y tras probar de ambas maneras ha resultado ser mejor después de la función de activación. Este modelo también debería de reducir el sobreajuste, ya que es un regularizador. Tras ejecutar el entrenamiento con esta mejora los resultados han sido los siguientes.

	Train	Validation
Loss	0.94	1.34
Accuracy	0.70	0.60

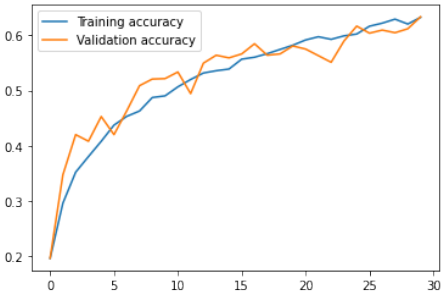


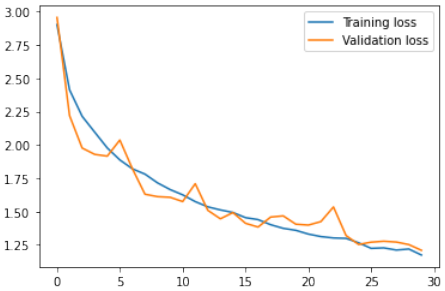
Vemos una buena mejora, dejándonos ya un 60 % de accuracy, aunque se mantiene el sobreajuste. Sin embargo, es una mejora considerable y nuestro modelo comienza a clasificar de manera bastante decente a esta base de datos.

2.6. Dropout

Dropout es la última mejora planteada, es también un regularizador y funciona de tal forma que bloquea algunas entradas a algunas neuronas, de manera que estas se especialicen, mejorando así el rendimiento. Como parámetro deberemos pasarle el porcentaje de entradas que bloquea, en nuestro caso será un 25 %.

	Train	Validation
Loss	1.17	1.20
Accuracy	0.63	0.63





Vemos una gran mejora, habiendose corregido todo el sobreajuste y dando un buen resultado, como es un 63 % de accuracy tanto para el conjunto de train como para el de validación.

2.7. Modelo final

Tras todas estas mejoras se ha visto conveniente aplicar las 6 recomendaciones que se aportaban en el ejercicio, quedando el modelo tal que así.

Layer No.	Layer Type	Kernel Size	I|O dim.	I|O chan.
1	Conv2D	3	32|32	16|16
2	Relu	-	32|32	-
3	BatchNorm.	-	32|32	-
4	Conv2D	3	32|32	16|16
5	Relu	-	32|32	-
6	BatchNorm.	-	32|32	-
7	MaxPooling2D	2	32|16	-
8	Conv2D	3	16|16	16|32
9	Relu	-	16|16	-
10	BatchNorm.	-	16|16	-
11	Conv2D	3	16|16	32|32
12	Relu	-	16|16	-
13	BatchNorm.	-	16|16	-
14	MaxPooling2D	2	16|8	-
15	Conv2D	3	8|8	32|64
16	Relu	-	8|8	-
17	BatchNorm.	-	8|8	-
18	Conv2D	3	8|8	64|64
19	Relu	-	8|8	-
20	BatchNorm.	-	8|8	-
21	MaxPooling2D	2	8|4	-
22	Dropout	-	1024|1024	-
23	Linear	-	1024|50	-
24	Relu	-	50|50	-
25	BatchNorm.	-	50|50	-
26	Dropout	-	50|50	-
27	Linear	-	50|25	-

Dicho modelo mejora a BaseNet también en test. Se ha introducido en un objeto *ImageDataGenerator* los datos de test y se han normalizado. Tras eso se ha utilizado la función *calcularAccuracy* dada y se ha comprobado que efectivamente, mejora el modelo BaseNet, podemos ver a continuación una comparación de los valores.

	BaseNet	BaseNet mejorada
Test accuracy	0.44	0.65

Como vemos hemos mejorado en un 21 % la eficacia de BaseNet, por lo que se considera satisfecho el objetivo del ejercicio.

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

ResNet50 es una red preentrenada, en nuestro caso para ImageNet, presente en *keras.applications.resnet*50. En este apartado vamos a aplicar diferentes técnicas que nos permitirán usarla en la base de datos Caltech-UCSD, la cual contiene 6033 imágenes de 200 clases distintas de pájaros. Remarcar en este apartado que los resultados están sujetos a estocasticidad y que, debido al gran esfuerzo computacional que requiere el entrenamiento de los modelos, no se han

realizado demasiadas pruebas y estos pueden variar en un cierto margen en la práctica, aunque no debería ser demasiado dramático.

3.1. Adaptación de ResNet50 a Caltech-UCSD

Para los dos primeros que vamos a probar se va a prescindir de la última capa densa y la función de activación, quedando el extractor de características de ResNet. El caso básico es introducir dichas características en una capa densa de 200 neuronas(la longitud de nuestra salida) y una activación SoftMax, que nos clasicarán las características consiguiendo de nuevo un clasificador a partir de las dos estructuras. El segundo modelo que se va a comprobar es exactamente el mismo que el anterior pero con otra capa densa, de manera que el clasificador conste dos y no de una capa. Por último se va a intentar añadir alguna capa extra a la vista de los resultados de los otros modelos prescindiendo de AVG Pooling.

Lo primero es crear dos objetos de la clase $ImageDataGenerator$ para los conjuntos de entrenamiento y test. En este caso, como medida preventiva, se normalizarán los datos del conjunto de entrenamiento, separando además un 10 % para validación. Se usará un tamaño de batch de 64, igual que en los apartados anteriores, y un número de épocas de 12, ya que la ejecución será bastante costosa. La compilación del modelo se realiza exactamente igual que durante el resto de la práctica, no obstante se invita a leer las reflexiones de la conclusión de este apartado, ya que esto ha podido ser un pequeño error.

Llamamos entonces a ResNet con los argumentos $include\_top$ a $false$ de manera que no incluya ni la capa densa ni la activación como queríamos, además debemos asegurarnos de que los pesos sean los calculados con ImageNet utilizando el parámetro $weights$ dado para ello y, de paso, añadir la capa AVG Pooling con $pooling =' avg'$. Como sólo vamos a entrenar la/s nueva/s capa/s del modelo, marcamos $resnet.trainable = False$, de manera que los pesos se queden congelados y sólo se entrenen los de las capas que añadamos.

Hay que tener en cuenta que antes de entrenar los modelos tenemos que cargar la base de datos y definir los conjuntos de test y entrenamiento, desde la carpeta imagenes en este caso en situada en la carpeta raíz del Drive enlazado a Colab, con la función $cargarDatos$ dada para ello. **Importante**: si se desea hacerlo en local, habrá que cambiar el path que se le pasa a dicha función. Tras esto ya podemos ajustar el objeto ImageDataGenerator para entrenamiento a los datos de entrenamiento con la función fit. Se compilarán los modelos con el mismo optimizador, función de pérdida y métrica que se llevan usando a lo largo de la práctica, y se entrenarán también de la misma manera que en ejercicios anteriores, con la función fit del modelo pasando los conjuntos de entrenamiento y validación a través del objeto $ImageDataGenerator$, junto al resto de parámetros necesarios para el experimento ya comentado.

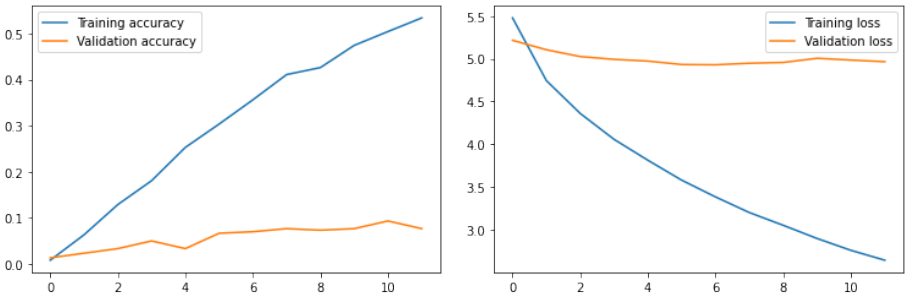
3.1.1. Adaptación cambiando sólo la capa de salida y activación

Para el primer modelo añadiremos haremos que el output de nuestro resnet(que ahora mismo es un vector de características) sea la entrada de una capa

de 200 neuronas, activada por una función SoftMax que intentará clasificar las imágenes.

Tras el entrenamiento, podemos ver los siguientes resultados devueltos por las funciones dadas para ello, *mostrarEvolucinycalcularAccuracy*.

	Train	Validation
Loss	2.64	4.96
Accuracy	0.53	0.07



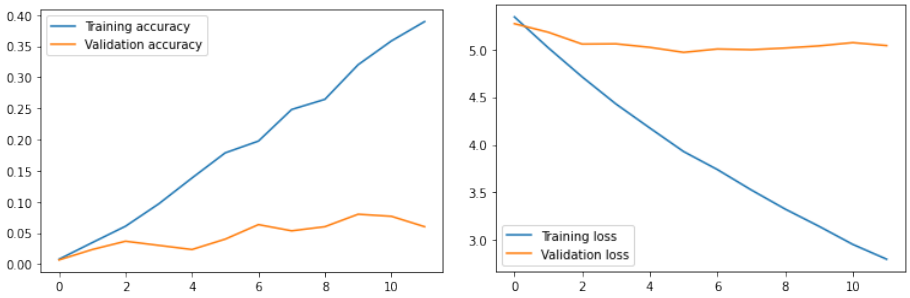
Test accuracy: 0.009

Observamos un mal resultado, algo por debajo del 1 % de accuracy. Se puede ver un claro sobreajuste, lo cual era esperable dado que estamos usando un extractor de características muy potente y entrenado para una base de datos mucho más grande y general para la nuestra que es más reducida y específica. Se podría intentar tratarlo con Batch Normalization o Dropout, pero parece ser demasiado grande como para que esto resulte.

3.1.2. Adaptación cambiando capa de salida y activación con dos capas FC

Para el segundo modelo añadimos también la última capa con 200 neuronas con la activación SoftMax, pero esta vez una capa FC más oculta, que se ha elegido que sea de 400 neuronas, el doble de la capa de salida. Vemos los resultados del entrenamiento con nuestro conjunto de datos, con el mismo formato anterior a continuación.

	Train	Validation
Loss	2.79	5.04
Accuracy	0.38	0.06



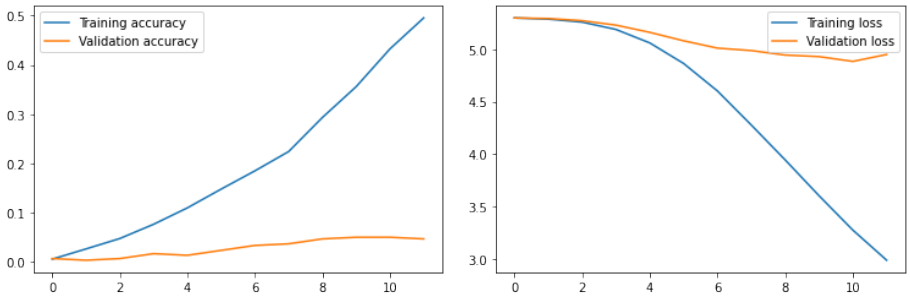
Test accuracy: 0.011

Observamos también un mal resultado como en el caso anterior, con algo menos de sobreajuste pero, a pesar de la mejora que proporciona la capa FC extra(aunque, al menos bajo mi punto de vista, no debería mejorar ya que una capa más aumentaría los parámetros y a más parámetros bajo los mismos datos debería crecer el sobreajuste), malo igualmente, por el mismo motivo que en el caso anterior.

3.1.3. Adaptación introduciendo nuevas capas

Para este tercer modelo en el que tenemos algo más de libertad, trataremos de intentar reparar en la medida posible el sobreajuste añadiendo una capa de Batch Renormalization, que puede mejorar a Batch Normalization en este caso, y otra de Dropout a 0.5, antes de Global Average Pooling y la capa de salida. Tras el entrenamiento, podemos ver los siguientes resultados, devueltos como siempre en el formato ya comentado.

	Train	Validation
Loss	2.98	4.95
Accuracy	0.49	0.04



Test accuracy: 0.005

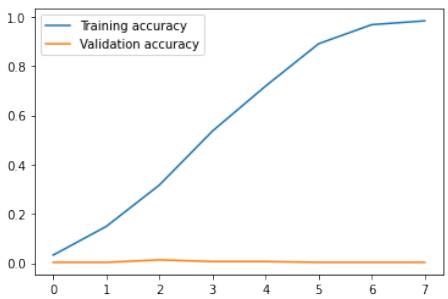
A la vista de los resultados es evidente que no se ha conseguido el objetivo, dado que los resultados son igual de malos o peores. Es evidente que este sobreajuste es complicado de corregir, y probablemente lo que ha ocurrido es que intentando regularizar hayamos añadido demasiados parámetros con los mismos datos, por lo que crece el sobreajuste y lo único que hacemos es aumentar la

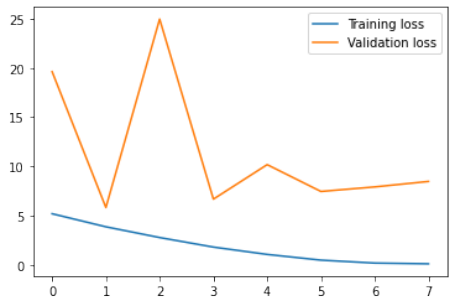
complejidad del modelo. No parece lógico añadir más capas convolucionales ni más capas densas, por lo que no se ve otra alternativa clara.

3.2. Ajuste fino de ResNet50 a Caltech-UCSD

En este apartado se va a intentar un ajuste fino de la primera red implementada en el apartado anterior basada en ResNet. Un ajuste fino es básicamente reentrenar la red pero con los pesos que se nos proporcionan del preentrenamiento. Esto se hace realizando el mismo proceso explicado en el apartado correspondiente pero dejando $resnet.Trainable = True$, de manera que no solo se entrenarán los pesos de las nuevas capas si no que lo hará la red completa, ajustandose de manera fina(de ahí el nombre ajuste fino) los pesos dados por ResNet50 de Keras. En este caso se van a realizar 8 épocas en lugar de 12 porque la ejecución tendrá un coste computacional enorme, de manera que esta sea, al menos, abordable.

	Train	Validation
Loss	0.09	8.46
Accuracy	0.98	0.003





Test accuracy: 0.002

Podemos apreciar el mismo comportamiento, en este caso más destacado, que en los casos anteriores. Es lógico(aunque exagerado, se valorará en la conclusión del apartado) ya que en este caso no solo estamos usando un modelo sobreescalado para el problema, si no que lo estamos entrenando completamente, por lo que se ajustará más todavía al conjunto de entrenamiento produciendo un gran sobreajuste.

3.3. Conclusión

Como conclusión para este apartado de transferencia de modelos podemos sacar varias conclusiones. La primera es ver lo fácil que es usar un modelo preentrenado para nuestra base de datos, lo cual puede ser útil si no tenemos la capacidad, o el interés, de diseñar nuestro propio modelo. La segunda es que, sin embargo, este modelo preentrenado debe de estar medianamente escalado a nuestro problema, ya que si como es el caso el modelo está sobredimensionado obtendremos overfitting, mientras que en el caso contrario, en el que el modelo sea demasiado simple para el problema, el modelo no será capaz de aprender lo suficiente y obtendremos underfitting. A pesar de esto, los resultados obtenidos

son demasiado malos, por lo que tras algunas reflexiones se ha llegado a la conclusión de que el problema en algunos casos podría ser el learning rate, sobre todo en el ajuste fino, dado que el reentrenamiento debería de haber sido más suave y es muy probable que se hubieran sacado mejores resultados. No obstante, aunque dada la complejidad de los modelos no se han podido volver a reproducir las ejecuciones, es muy probable que los resultados hubieran mejorado, pero a la vista de los resultados es prácticamente seguro que se va a seguir dando sobreajuste por lo que se creen correctas las conclusiones a las que se han llegado.