Práctica 2
Visión por Computador

Redes neuronales convolucionales

Paula Iglesias Ahualli
Noviembre de 2019

Paula Iglesias Ahualli
Noviembre de 2019

CONTENIDO

Paula Iglesias Ahualli
Noviembre de 2019

PRÁCTICA 1 VISIÓN POR COMPUTADOR

1. EJERCICIO 1: BASENET EN CIFAR100

Editar BaseNet para diseñar una arquitectura de red profunda más precisa, con el conjunto de datos de CIFAR100.

ARQUITECTURA

Utilizamos la siguiente arquitectura propuesta:

Layer No.	Layer Type	Kernel size	Input dimension	Output dimension	Input channels	Output channels
1	Conv2D	5	32	28	3	6
2	Relu		28	28		
3	MaxPooling2D	2	28	14	6	16
4	Conv2D	5	14	10		
5	Relu		10	10		
6	MaxPooling2D	2	10	5		
7	Linear		400	50		
8	Relu		50	50		
9	Linear		50	25		

FUNCIONES

MODELO

"Transcribimos" directamente la arquitectura que especificamos arriba en la función *base_net()*.

ENTRENAMIENTO

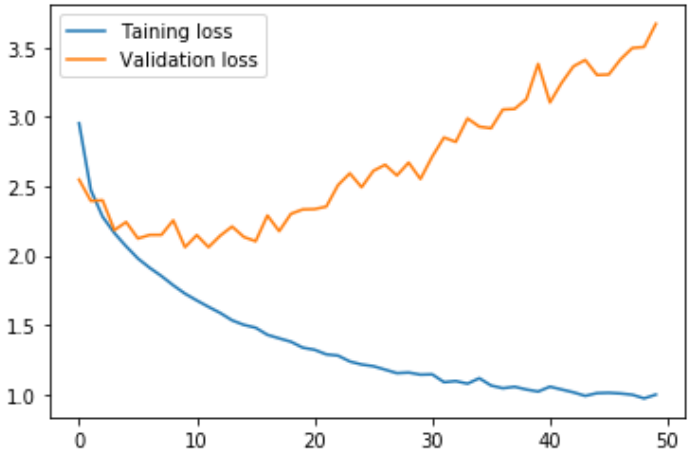
Utilizamos el optimizador SGD y compilamos con la función de los *"categorical_crossentropy"* y la métrica *"accuracy"*.

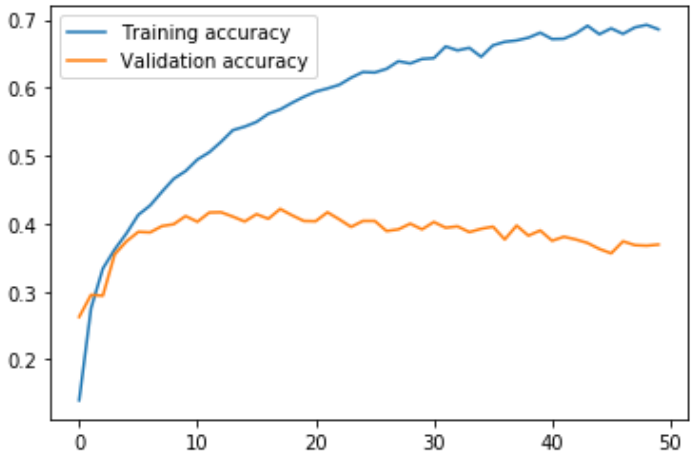
Como no hacemos uso de Data Augmentation podemos usar la función *fit()*.

RESULTADOS

Obtenemos los siguientes resultados en cuanto a *loss* y *accuracy*.

Loss	3.6710
Accuracy	0.3692





Podemos observar cómo la función de error de validación llega a un punto en el que no disminuye, incluso experimenta subidas. Podemos interpretarlo como *overfitting*. Se podría haber parado mucho antes la ejecución, pero he elegido utilizar 50 épocas para comparar fielmente con las mejoras que haremos en el siguiente apartado.

2. Ejercicio 2: BaseNet en CIFAR100

Crear una red profunda mejorada respecto al modelo anterior. La precisión debería acercarse al 50%

Red más profunda y capas de normalización

Layer No.	Layer Type	Kernel size	Input dimension	Output dimension	Input channels	Output channels
1	**Conv2D**	3	32	30	3	32
2	**Relu**		30	30		
3	**Batch Normalization**		30	30		
4	**Conv2D**	3	30	28		32

5	**Relu**		28	28		
6	**Batch Normalization**		28	28		
7	**MaxPooling2D**	2	28	14		
8	**Dropout**		14	14		
9	**Conv2D**	3	14	12	32	64
10	**Relu**		12	12		
11	**Batch Normalization**		12	12		
12	**Conv2D**		12	10	64	64
13	**Relu**		10	10		
14	**Batch Normalization**		10	10		
15	**MaxPooling2D**		10	5		
16	**Dropout**		5	5		
17	**Conv2D**		5	3	64	128
18	**Relu**		3	3		
19	**Batch Normalization**		3	3		
20	**MaxPooling2D**		3	1		
21	**Dropout**		1	1		
22	**Linear**		128	256		
23	**Relu**		256	256		
24	**Linear**		256	25		

CAPAS DE CONVOLUCIÓN

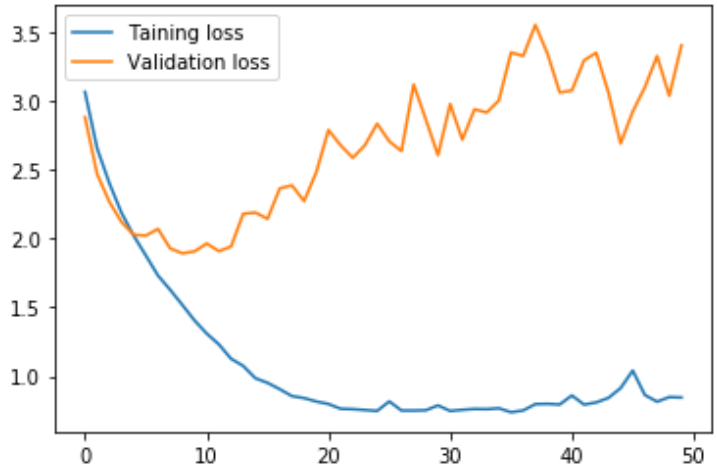
Añadimos más capas de convolución con canales de salida crecientes, para lo cual disminuímos el tamaño del kérnel a 3.

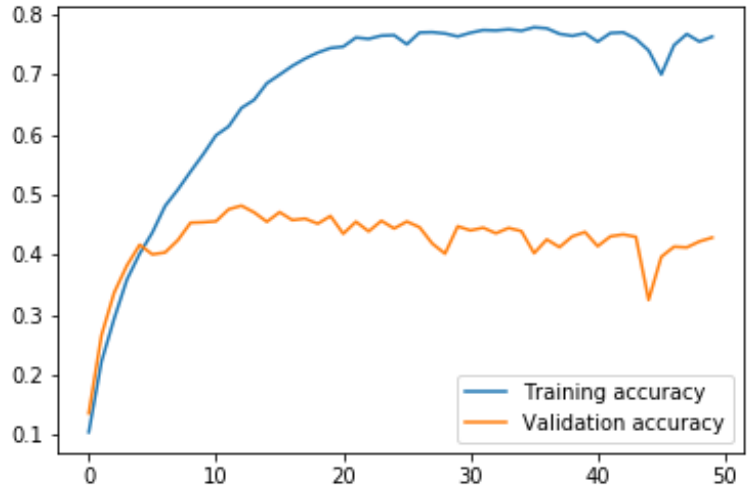
MaxPooling

Para no perder demasiada información, no hacemos maxpooling después de capa convolucional.

Loss	3.4021
Accuracy	0.4284

Podemos ver una mejora increíble en precisión. Sin embargo, vemos que tenemos muchísimo overfitting.





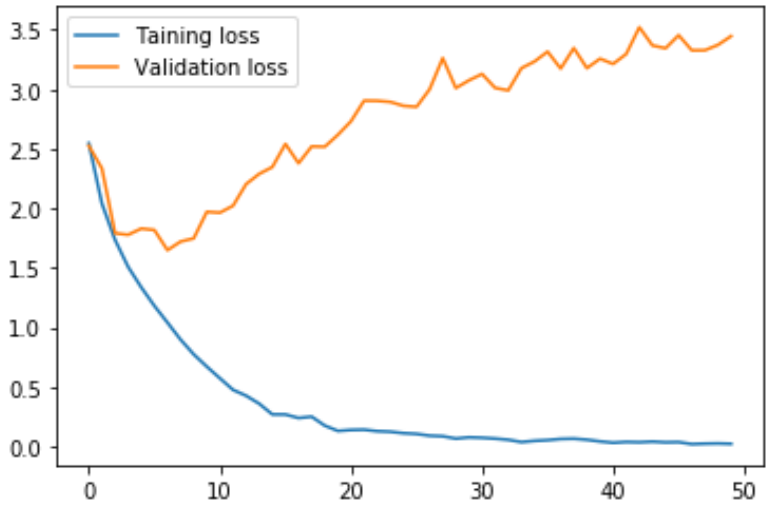
BATCH NORMALIZATION

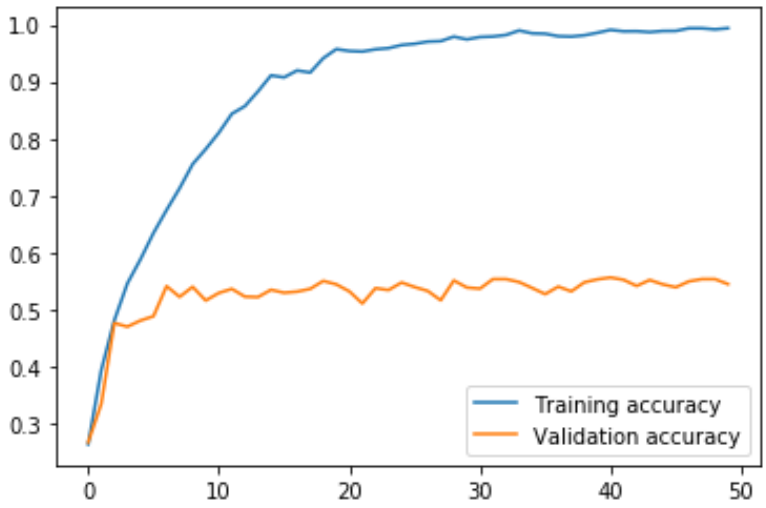
Para reducir el sobreajuste y mejorar el entrenamiento añadimos capas de normalización, después de las capas de convolución.

Probaremos los resultados de, añadir, además, capas de normalización antes y después de las capas de activación.

Loss	3.4472
Accuracy	0.5444

Mejora muy significativa (seguimos teniendo problema de overfitting).

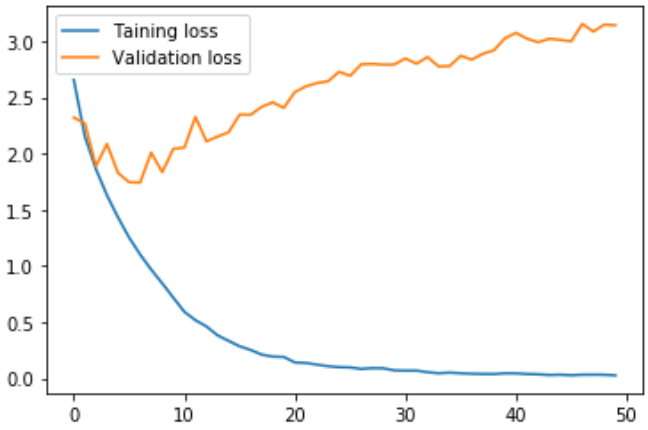


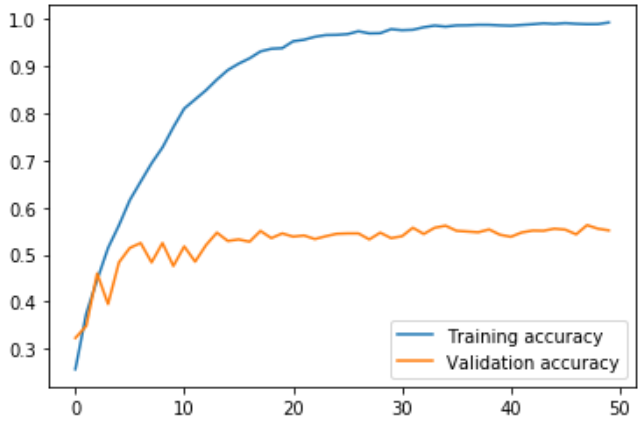


Ahora vamos a probar a añadir capas de normalización después de las de activación.

Vemos que obtenemos una mínima mejora.

Loss	3.1421
Accuracy	0.5512





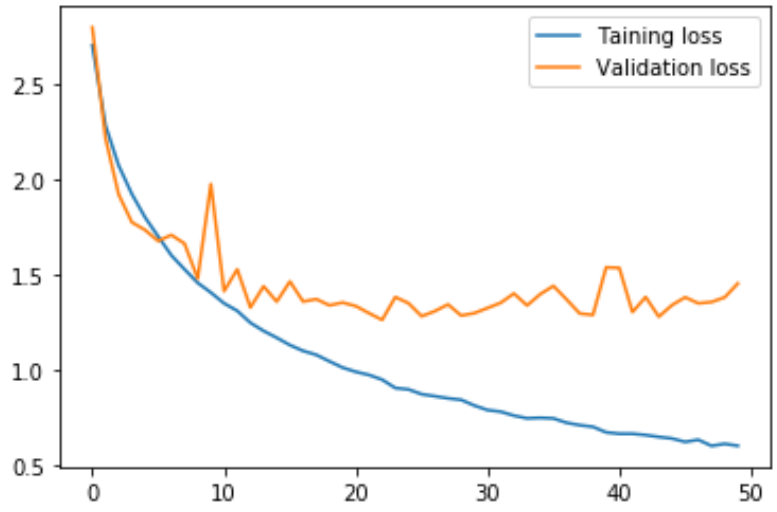
DROPOUT

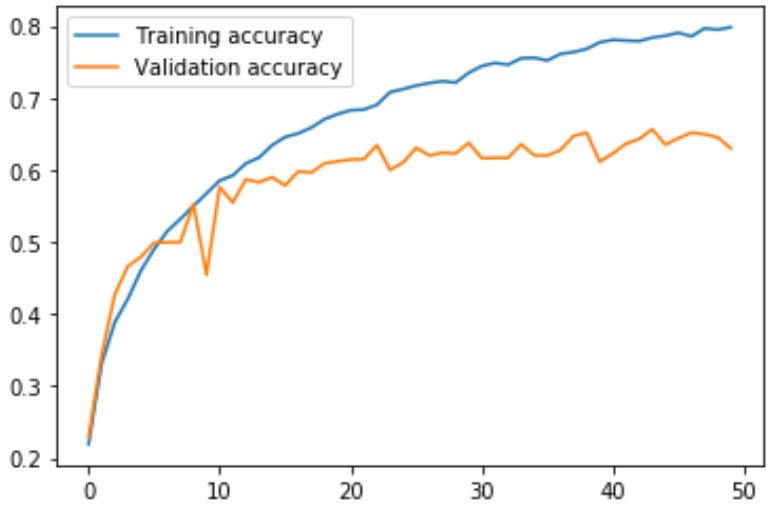
Una forma efectiva de prevenir el overfitting es añadiendo capas Dropout, como podemos ver en el paper *[Dropout: A Simple Way to Prevent Neural Networks from Overfitting Nitish Srivastava.]*

Loss	1.4528
Accuracy	0.63

Podemos ver como los datos han mejorado muy significativamente, y además hemos conseguido reducir el overfitting, aunque si bien no completamente.

Ya hemos realizado las mejoras propuestas en cuanto al modelo, veamos ahora las relativas al entrenamiento.





NORMALIZACIÓN DE DATOS

En ImageDataGenerator, incluímos los parámetros media=0 y desviación estándar 1.

*datagen = ImageDataGenerator(**featurewise_center = True,***

featurewise_std_normalization = True,

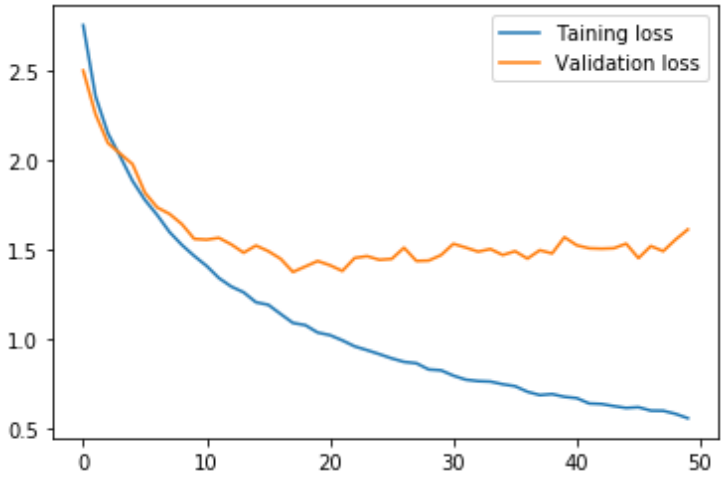
validation_split = 0.1

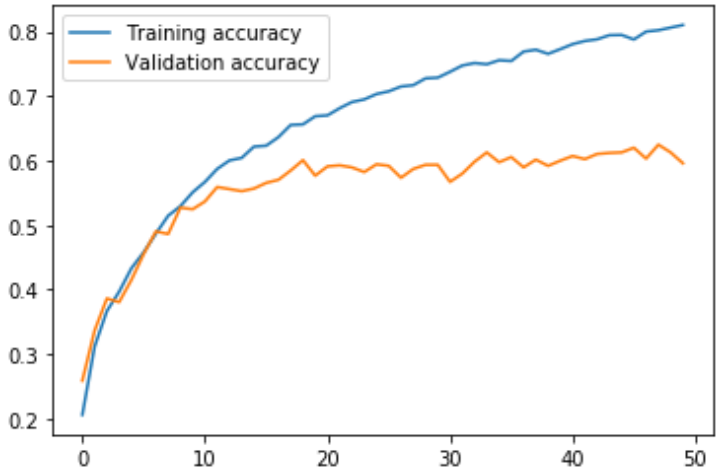
)

MEJORA

Loss	1.4679
Accuracy	0.6296

No parece haber cambios significativos en loss y accuracy, pero es muy interesante ver en las gráficas, como gracias a la normalización las curvas tienen muchos menos picos. Aún seguimos notando un overfitting importante.





AUMENTO DE DATOS

Utilizamos el aumento de datos para aumentar la generalidad de nuestro modelo, es decir, prevenir el *overfitting* que hemos visto en los resultados anteriores.

https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/

Para esto hay varias transformaciones que podemos aplicar a las imágenes, como traslaciones, rotaciones, cambios de escala, cizallamiento y volteados horizontales o verticales.

Estas transformaciones se definen con ImageDataGenerator, así quedaría la llamada a la función:

datagen = ImageDataGenerator(featurewise_center = True,

featurewise_std_normalization = True,

validation_split = 0.1,

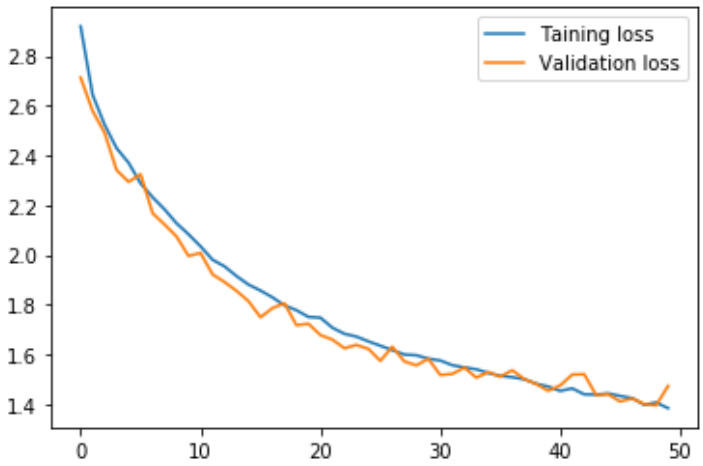
zoom_range=0.15,

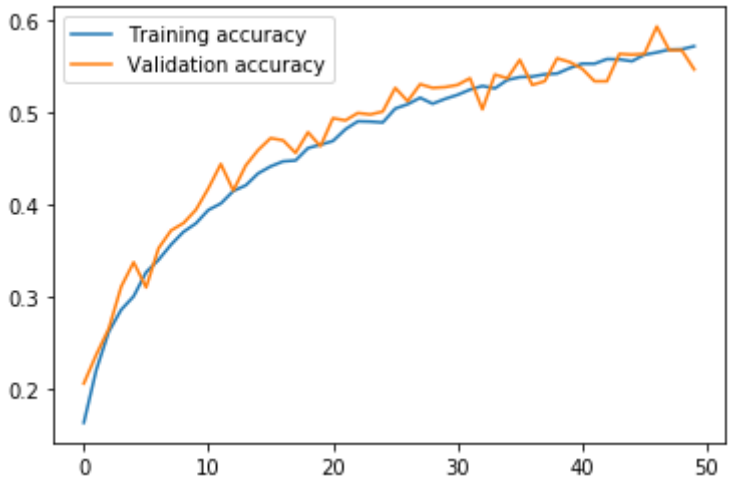
```
    horizontal_flip=True,
    fill_mode="nearest",
    width_shift_range=0.2,
    height_shift_range=0.2
)
```

MEJORA

Loss	1.2814
Accuracy	0.6

¡Hemos conseguido reducir el overfitting!





EARLY STOPPING

Como hemos podido ver en el ejemplo anterior, llega un momento en el que seguir entrenando más épocas deja de tener sentido. Para prevenir esto, se utiliza la técnica llamada Early Stopping, en la que se deja de entrenar cuando el rendimiento del modelo deja de mejorar.

Para esto usamos la función de keras EarlyStopping. En monitor especificamos la medida que queremos que cuando deje de mejorar, se deje de entrenar, en nuestro caso *accuracy*. En *patience* especificamos cuanto "margen" le damos en la que no se vea mejora. Especificamos patience=5 y ponemos *verbose=1* para que nos indique en qué época ha dejado de entrenar.

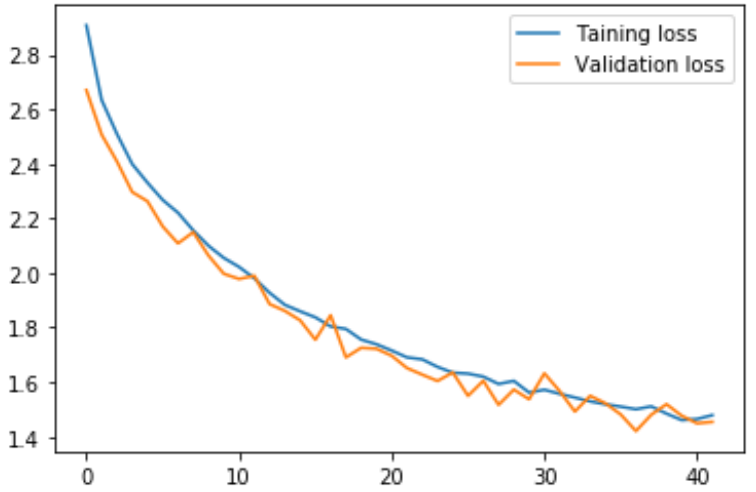
MEJORA

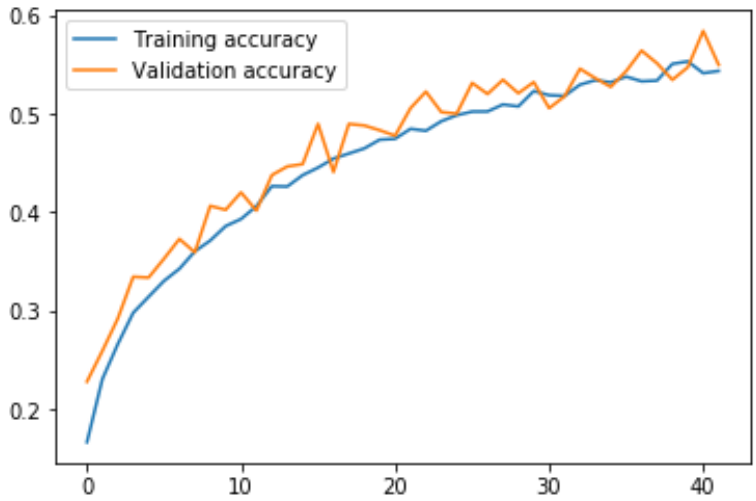
Loss	1.2689
Accuracy	0.6072

Restoring model weights from the end of the best epoch

Epoch 00042: early stopping

Aquí podemos ver como el early stopping ha servido para parar 6 épocas antes, aunque no se ha notado demasiado en los datos de los y accuracy.





Paula Iglesias Ahualli
Noviembre de 2019

3. EJERCICIO 3: TRANSFERENCIA DE MODELOS Y AJUSTE FINO CON RESNET50

3.1. RESNET 50 COMO EXTRACTOR DE CARACTERÍSTICAS

Usar ResNet como extractor de características preentrenado en ImageNet en el conjunto de datos Caltech-UCSD

ARQUITECTURA

Definimos el modelo que entrenaremos, muy simple, consistirá solamente en dos capas dense

modelo = Sequential()

modelo.add(Dense(1024,input_shape=(2048,), activation='relu'))

modelo.add(Dense(200, activation = 'softmax'))

Para realizar la extracción de características:

- Como vamos a usar resnet, debemos utilizar la función de preprocesamiento de datos adecuada para este modelo, *preprocess_input*
- Definición de resnet

modelo_base = ResNet50(include_top=False, pooling='avg', weights='imagenet')

No debemos incluir la ultima capa, pues no queremos clasficar imagenet, pero sí que incluimos los pesos resultado del entrenamiento con esta base de datos.

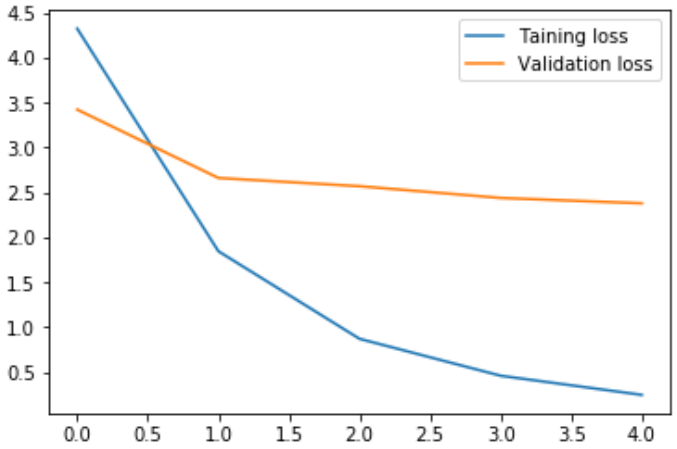
- Entrenaremos el modelo de dos capas dense con las características de resnet, para esto:

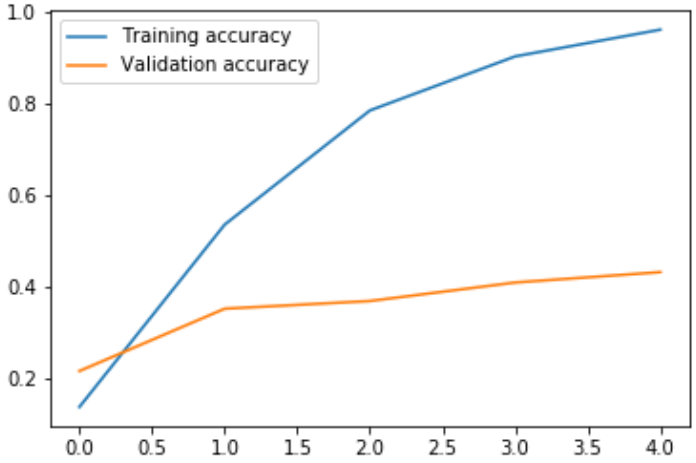
x_train_extractor = modelo_base.predict(x_train, batch_size=batch_size, verbose=1)
x_test_extractor = modelo_base.predict(x_test, batch_size=batch_size, verbose=1)

x_train_extractor y x_test_extractor son los conjuntos entrenamiento y test con los que entrenaremos gracias a fit

RESULTADOS

Loss	**2.376**
Accuracy	0.4309





Tan solo entrenando dos capas podemos llegar a una precisión del 40% gracias a la red preentrenada ResNet. Hemos entrenado diez épocas, aunque podríamos habernos limitado a aún menos, debido al overfitting que claramente se aprecia en las gráficas.

3.2. FINETUNING

Ajuste fino de toda la red ResNet50 también preentrenada en ImageNet en el conjunto de datos Caltech-UCSD

A diferencia de lo que hemos hecho en el apartado anterior, para proceder a realizar un ajuste fino no sólo debemos adaptar la red preentrenada, sino volverla a entrenar para que aprenda los nuevos datos.

https://www.pyimagesearch.com/2019/06/03/fine-tuning-with-keras-and-deep-learning/

Para esto

Definimos resnet como en el apartado anterior, quitando la última capa y con los pesos de imagenet

modelo_base = ResNet50(include_top=False, pooling='avg', weights='imagenet')

En este caso nuestro modelo tendrá como input el modelo base y el output serán dos capas densas. De este modo esta vez sí estaremos entrenando ResNet.

x=modelo_base.output

Añadimos una red totalmente conectada

x = Dense(1024, activation='relu')(x)

y como tenemos 200 clases, una capa para esta logística

predicciones = Dense(200, activation='softmax')(x)

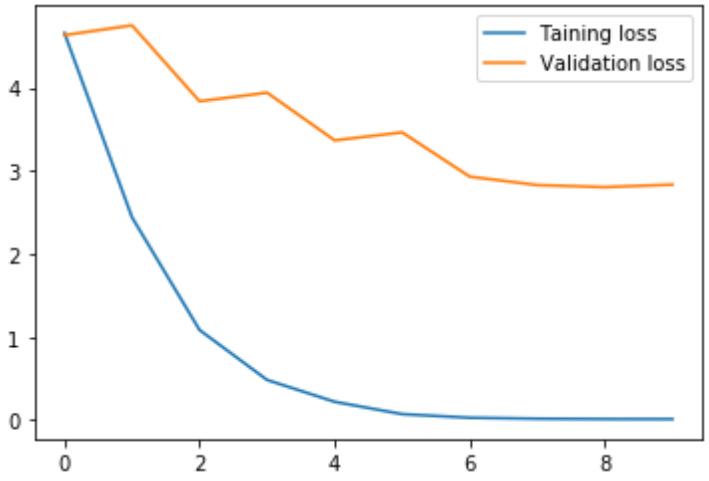
Modelo que entrenaremos:

modelo = Model(inputs=modelo_base.input, outputs = predicciones)

Entrenaremos 10 épocas, más que suficiente, pues tendremos overfitting, con fit().

Obtenemos el siguiente resultado:

Loss	2.8350
Accuracy	0.4026



Paula Iglesias Ahualli
Noviembre de 2019